Numerical Quality and High Performance In Interval Linear Algebra on Multi-Core Processors

Philippe Theveny

▶ To cite this version:

Philippe Theveny. Numerical Quality and High Performance In Interval Linear Algebra on Multi-Core Processors. Other [cs.OH]. Ecole normale supérieure de lyon - ENS LYON, 2014. English. NNT : 2014ENSL0941 . tel-01126973

HAL Id: tel-01126973
https://theses.hal.science/tel-01126973v1

Submitted on 6 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

en vue de l'obtention du grade de

Docteur de l'Université de Lyon, délivré par l'École Normale Supérieure de Lyon

Discipline : Informatique

Laboratoire de l'Informatique du Parallélisme

École Doctorale InfoMaths (ED 512)

présentée et soutenue publiquement le **31 octobre 2014**

par Monsieur Philippe Théveny

Numerical Quality and High Performance in Interval Linear Algebra on Multi-Core Processors

Directeur de thèse : Gilles Villard
Co-Encadrante : Nathalie Revol

Devant la commision d'examen composée de :
M. Marc Baboulin, LRI, Examinateur
M. Laurent Granvilliers, Université de Nantes, Rapporteur
Mme Mariana Kolberg, Universidade Federal do Rio Grande do Sul, Invitée
M. Julien Langou, University of Colorado Denver, Rapporteur
Mme Sylvie Putot, CEA LIST, Examinatrice
Mme Nathalie Revol, LIP, Co-Encadrante
M. Gilles Villard, LIP, Directeur

Contents

Introduction **3**

I Error Analysis **11**

Measuring error in interval linear algebra **13**

1 Metric on the set of real intervals **17**
1.1 Midpoint-radius representation 17
1.2 The choice of a metric 19
1.3 Relations between metrics 22
1.4 Conclusion 23

2 Accuracy of interval inner products **25**
2.1 Interval arithmetic variants 25
2.2 Interval inner product and variants 27
2.3 Arithmetic error of interval inner products 29
2.4 Arithmetic error analysis 31
2.5 A new approximate inner product 35
2.6 Conclusion 38

3 Accuracy of interval matrix products **39**
3.1 Floating-point model 39
3.2 Interval matrix product in three point matrix products 41
3.3 Interval matrix product in five point matrix products 44
3.4 A new algorithm in two point matrix products 47
3.5 Conclusion 49

4 Global error analysis **51**
4.1 Measuring the global error experimentally 51
4.2 Global error for `MMMul3` 53
4.3 Global error for `MMMul2` 60
4.4 Global error for `MMMul5` 63
4.5 Conclusion 66

II Parallel Implementation 71

Parallel interval linear algebra on multi-core processors 73

5 Implementation issues with regard to interval linear algebra 77
5.1 Rounding modes . . . 77
5.2 Execution order . . . 82
5.3 Conclusion . . . 83

6 Implementation methodology 85
6.1 Priority list of implementation goals . . . 85
6.2 Parallel linear algebra on multi-core processors . . . 87
6.3 Experimental protocols . . . 96
6.4 Conclusion . . . 99

7 Hardware model and blocking for single core computations 101
7.1 Hardware model for single core performance prediction . . . 101
7.2 Vector instruction set: Streaming SIMD Extensions . . . 104
7.3 Block computations . . . 106
7.4 Execution time of block kernels . . . 116
7.5 Conclusion . . . 122

8 Multi-core and multi-threading 123
8.1 Multi-threaded implementations of block algorithms . . . 123
8.2 Sequential execution time . . . 124
8.3 Measure of execution time for multi-threaded runs . . . 126
8.4 Conclusion . . . 130

Conclusion 133

Bibliography 135

List of Figures

1 Radius Overestimation – IIMul3 13
2 Global Error Decomposition 14
3 Part I Synopsis 15

1.1 Midpoint-Radius versus Infimum-Supremum Representation 18
1.2 Inclusion in Midpoint-Radius Representation 19
1.3 Relative Accuracy versus Maximum Relative Error 21
1.4 Relative Accuracy of Positive Intervals 22

2.1 Addition in Midpoint-Radius Representation 25
2.2 Product in Midpoint-Radius Representation 27
2.3 Exact and Approximate Inner Products 29
2.4 Upper Bounds on the Relative Arithmetic Error for $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ (Isolines) 32
2.5 Upper Bounds on the Relative Arithmetic Error for $\boldsymbol{z}_1$ $(e = f)$ 33
2.6 Upper Bounds on the Relative Arithmetic Error for $\boldsymbol{z}_2$ $(e = f)$ 33
2.7 Radius Overestimation for z_3 37
2.8 Upper Bounds on the Relative Arithmetic Error for z_3 38

4.1 Global Error Measurement 53
4.2 Bound on the Relative Radius Error for MMMul3 56
4.3 Random Dataset I – normal midpoints, fixed relative accuracy 56
4.4 Relative Hausdorff Error for MMMul3 – Dataset I 57
4.5 Random Dataset II – normal midpoints, bounded relative precision 59
4.6 Relative Hausdorff Error for MMMul3 – Dataset II 59
4.7 Bound on the Relative Radius Error for MMMul2 61
4.8 Bound on the Relative Hausdorff Error for MMMul2 62
4.9 Relative Hausdorff Error for MMMul2 – Dataset I 62
4.10 Relative Hausdorff Error for MMMul2 – Dataset II 63
4.11 Bound on the Relative Radius Error for MMMul5 65
4.12 Relative Hausdorff Error for MMul5 – Dataset I 65
4.13 Relative Hausdorff Error for MMul5 – Dataset II 66
4.14 Bounds on the Relative Radius Error for MMMul3 and MMMul5 67
4.15 Bounds on the Relative Hausdorff Error for MMMul3 and MMMul2 68
4.16 Decision Tree for Interval Matrix Multiplications 69
4.17 Part II Synopsis 75

6.1 NUMA machine 87
6.2 Multi-Core Multi-Processor Synopsis 89
6.3 Matrix Formats 93

7.1 Out-of-Order Execution Engine Model (Intel Sandy Bridge Micro-architecture). . 102
7.2 _m128d Variable 104
7.3 _mm_unpackhi_pd Intrinsics 105
7.4 Arithmetic and Logic Intrinsics 106
7.5 Row-Vector of Upper Bounds on Relative Accuracies 107
7.6 Column-Vector of Upper Bounds on Relative Accuracies 107

8.1 Execution time – Sequential. 125
8.2 Execution time – 8 threads. 127
8.3 Execution time – 32 threads. 128
8.4 MMMul2 – Scalability 129

List of Tables

4.1 Relative Hausdorff Error for MMMul3 – Bound and Measures 57
4.2 Relative Hausdorff Error of MMMul3 – Dataset I 58
4.3 Relative Hausdorff Error of MMMul3 – Dataset II 60
4.4 Relative Hausdorff Error for MMMul2 – Dataset I 62
4.5 Relative Hausdorff Error for MMMul2 – Dataset II 63
4.6 Relative Hausdorff Error for MMMul5 – Dataset I 66
4.7 Relative Hausdorff Error for MMMul5 – Dataset II 66
4.8 Error Bounds and Costs for Algorithms MMMul3 and MMMul5 67

6.1 Processor Description 87
6.2 Correspondence between Hardware Structure, Data Structure, and Algorithmic Structure. 94

7.1 Individual Use of Execution Units and Ports in mm_dmidmul2_bb 110
7.2 Total Use of Execution Units in mm_dmidmul2_bb 110
7.3 Individual Use of Execution Units and Ports in mm_raccrow_bb 112
7.4 Total Use of Execution Units and Ports in mm_raccrow_bb 112
7.5 Individual Use of Execution Units and Ports in mm_racccol_bb 115
7.6 Total Use of Execution Units and Ports in mm_racccol_bb 115
7.7 Individual Use of Execution Units and Ports in mm_draddmul2_bb 116
7.8 Total Use of Execution Units and Ports in mm_draddmul2_bb 116
7.9 GCC Optimizations and mm_dmul2_bb 117
7.10 ICC Floating-Point Model Option and mm_dmul2_bb 117
7.11 Automatic versus Manual Unrolling 118
7.12 Theoretical Minimum Execution Time for mm_dmul2_bb – Sandy Bridge 118
7.13 Execution Times and Block Size – Sandy Bridge 119
7.14 Automatic and Manual Vectorization: Execution Times (in cycles). 121

8.1 Measured Times and Efficiency 129

List of Algorithms

1 Classical . . . 4
2 IIMul4 . . . 41
3 InfsupToMidrad . . . 42
4 MMMul3 . . . 42
5 IIMul3 . . . 43
6 MMMul3 with explicit temporaries. . . . 44
7 MMMul5 . . . 45
8 MMMul5 with explicit temporaries. . . . 46
9 MMMul2 . . . 48

Listings

5.1 Interval Division. 78
5.2 Rounding Mode Violation. 82
6.1 Data Structures for Interval Matrices. 95
7.1 The mm_dmul2_bb Kernel. 108
7.2 The mm_dmidmul2_bb Function with Intrinsics. 109
7.3 The mm_raccrow_bb Function with Intrinsics. 111
7.4 The mm_racccol_bb Function with Intrinsics. 112
7.5 The mm_racccol_bb Inner Loop. 114
7.6 The mm_draddmul2_bb Function with Intrinsics. 115
7.7 The mm_dmidmul2_bb Function in plain C. 120
8.1 The mul_mmd_2000 Function with Outer Iterations on Rows. 123

Notations

$\mathbb{R}$	field of real numbers
$a \in \mathbb{R}$	scalar (lower-case)
$x \in \mathbb{R}^n$	vector (lower-case)
$A \in \mathbb{R}^{n\times n}$	matrix (upper-case)
dot	algorithm (sansserif)
$\mathbb{IR}$	set of real closed intervals
$\boldsymbol{a} \in \mathbb{IR}$	interval (bold)
$\boldsymbol{x} \in \mathbb{IR}^n$	interval vector (lower-case + bold)
$\boldsymbol{A} \in \mathbb{IR}^{n\times n}$	interval matrix (upper-case + bold)
$\boldsymbol{x} = [\underline{x}, \overline{x}]$	infimum-supremum representation
$\underline{x}$	left endpoint
$\overline{x}$	right endpoint
$\boldsymbol{x} = \langle \operatorname{mid} \boldsymbol{x}, \operatorname{rad} \boldsymbol{x} \rangle$	midpoint-radius representation
$\operatorname{mid} \boldsymbol{x}$	midpoint
$\operatorname{rad} \boldsymbol{x}$	radius
$\lvert\boldsymbol{x}\rvert = \lvert\operatorname{mid} \boldsymbol{x}\rvert + \operatorname{rad} \boldsymbol{x}$	magnitude, absolute value
$\mathbb{F}$	set of floating-point numbers
t	floating-point precision
$\mathtt{u}$	unit roundoff (monospace)
$\mathtt{realmin}$	smallest positive normal number (monospace)
$\operatorname{ulp}(\tilde{x})$	unit in the last place
$\tilde{a} \in \mathbb{F}$	floating-point scalar (lower-case + tilde)
$\tilde{x} \in \mathbb{F}^n$	floating-point vector (lower-case + tilde)
$\widetilde{M} \in \mathbb{F}^{n\times n}$	floating-point matrix (upper-case + tilde)
$\mathrm{fl}_{\triangle}$	evaluation with rounding toward $+\infty$
$\mathrm{fl}_{\triangledown}$	evaluation with rounding toward $-\infty$
$\mathrm{fl}_{\square}$	evaluation with rounding to nearest
$\mathbb{IF}$	set of floating-point intervals
$\tilde{\boldsymbol{a}} \in \mathbb{IF}$	floating-point interval (lower-case + bold + tilde)
$\tilde{\boldsymbol{x}} \in \mathbb{IF}^n$	floating-point interval vector (lower-case + bold + tilde)
$\widetilde{\boldsymbol{M}} \in \mathbb{IF}^{n\times n}$	interval matrix (upper-case + bold + tilde)
$\Diamond$	outward rounding
$\mathcal{N}$	nearest midpoint rounding

Introduction

The work presented in this document is a contribution to the questions: Is is possible to substitute interval matrix computations to numerical matrix computations? and what will be the cost?

More precisely, we will focus on the case of interval matrix multiplication and we will provide a proof of concept for a parallel implementation that is efficient on multi-core processors.

One of the advantages of interval computations over numerical computations is that the former can handle data that are not precisely known. The imprecision may come from several sources like the uncertainty of the measure, for experimental data, from the floating-point error due to the representations of numbers that are used by computers, or from roundoff errors due to the computation with an arithmetic in finite precision.

Another important advantage of interval computation is that the result computed with the interval arithmetic is guaranteed to be an enclosure of the exact result, provided that the so-called "inclusion property" is preserved all along the computation. This characteristic of interval computation makes it possible to prove mathematical properties on a result that is computed with the floating-point arithmetic.

However, interval computations also have drawbacks: more operations are needed for an interval enclosure of the exact result than for a floating-point approximate. And, in particular when the computation is performed with floating-point arithmetic, interval results tend to be overestimated. These two concerns have to be taken into account when implementing an interval algorithm.

Interval matrix multiplications

Numerical computations performed by computers produce only approximates of the mathematically exact values. This is due to the finite precision arithmetic and the finite precision representation, which only approximate the real operations and real numbers.

Interval arithmetic gives an indication on the accuracy of the computed result, by returning, not a mere value, but an interval that is guaranteed to contain the exact real value. The most stringent requirement for this goal is that all interval operations verify the inclusion property[1].

Definition (from [Neu90, p. 13])**.** Let $f : \mathbb{R}^n \to \mathbb{R}^p$ be a vector function and $\boldsymbol{F} : \mathbb{IR}^n \to \mathbb{IR}^p$ be an interval vector function.

The interval function $\boldsymbol{F}$ is said to be an *interval enclosure* of f if, and only if, we have $f(x) \in \boldsymbol{F}(\boldsymbol{x})$ for any box $\boldsymbol{x}$ in $\mathbb{IR}^n$ and any point $x \in \boldsymbol{x}$.

This property is fundamental: it implies the range inclusion $f(\boldsymbol{x}) \subseteq \boldsymbol{F}(\boldsymbol{x})$ and that the composition $\boldsymbol{F} \circ \boldsymbol{G}$ of two interval enclosures $\boldsymbol{F}$ of f and $\boldsymbol{G}$ of g is itself an interval enclosure of $f \circ g$.

[1]Interval quantities and functions are noted in bold font and $\mathbb{IR}$ is the set of real intervals. See the table of notations page 1.

Definition. With the notations of the previous definition, the interval function $\boldsymbol{F}$ verifies the *inclusion property* with respect to f if, and only if, $\boldsymbol{F}$ is an interval enclosure of f and $\boldsymbol{F}$ is inclusion monotonic: $\boldsymbol{x} \subseteq \boldsymbol{y} \implies \boldsymbol{F}(\boldsymbol{x}) \subseteq \boldsymbol{F}(\boldsymbol{y})$.

Again, the composition $\boldsymbol{F} \circ \boldsymbol{G}$ of two inclusion monotonic interval enclosures F and G of f and g, respectively, verifies the inclusion property with respect to $f \circ g$.

The addition and the multiplication have interval enclosures that are inclusion monotonic, and it is therefore possible to define an interval matrix multiplication based on the interval addition and multiplication that verifies the inclusion property. The simplest floating-point implementation of such an interval matrix product is given below[2] (Algorithm 1). The important point here is that it suffices to replace numerical types (respectively arithmetic) by interval types (respectively arithmetic) to transform the classical triple nested loops algorithm into a valid algorithm for the interval matrix product: at line 5, the infimum bound ($\min\{\dots\}$) of $\boldsymbol{A}_{ik}\boldsymbol{B}_{kj}$ is accumulated in the infimum bound $\underline{\boldsymbol{C}_{ij}}$ of the result, and, at line 6, the supremum bound ($\max\{\dots\}$) is accumulated in the supremum bound $\overline{\boldsymbol{C}_{ij}}$.

Algorithm 1 Classical

Input: $\boldsymbol{A} = [\underline{A}, \overline{A}] \in \mathbb{IF}^{m\times k}, \boldsymbol{B} = [\underline{B}, \overline{B}] \in \mathbb{IF}^{k\times n}$
Output: $\boldsymbol{C} \in \mathbb{IF}^{m\times n}, \boldsymbol{C} \supseteq \boldsymbol{AB}$
1: **for** $i = 1$ to m **do**
2:    **for** $j = 1$ to n **do**
3:       $\underline{C}_{ij} \leftarrow 0$; $\overline{C}_{ij} \leftarrow 0$
4:       **for** $l = 1$ to k **do**
5:          $\underline{C}_{ij} \leftarrow \mathrm{fl}_{\nabla}\left(\underline{C}_{ij} + \min\left\{\underline{A}_{il}\underline{B}_{lj}, \underline{A}_{il}\overline{B}_{lj}, \overline{A}_{il}\underline{B}_{lj}, \overline{A}_{il}\overline{B}_{lj}\right\}\right)$
6:          $\overline{C}_{ij} \leftarrow \mathrm{fl}_{\Delta}\left(\overline{C}_{ij} + \max\left\{\underline{A}_{il}\underline{B}_{lj}, \underline{A}_{il}\overline{B}_{lj}, \overline{A}_{il}\underline{B}_{lj}, \overline{A}_{il}\overline{B}_{lj}\right\}\right)$
7:       **end for**
8:    **end for**
9: **end for**
10: **return** $[\underline{C}, \overline{C}]$

However, this algorithm requires eight times more floating-point operations that the corresponding numerical matrix product. In 1999, Rump [Rum99a] proposed a more efficient but less accurate algorithm. It is based on the midpoint-radius interval representation and performs the *interval* matrix product by using several well-optimized *floating-point* matrix products. More recently, other interval matrix multiplication algorithms, which use the same idea, have been presented by several authors [OO05, OOO11, Ngu11, OORO12, Rum12]. The number of floating-point matrix multiplications depends on the algorithm, varying from 4 in the original paper to 7 in [Ngu11]. In 2012, Rump found how to reduce this number by using a clever error bound [Rum12]. When this method is applied to the algorithm proposed in 1999, the number of floating-point matrix multiplications drops from 4 to 3, and from 7 to 5, when applied to Nguyen's algorithm.

In this document, we focus on these two last improved algorithms, plus a new one designed in the same spirit. The choice of interval matrix multiplication is motivated by the following reasons:

[2]Floating-point notations are given in the table of notations page 1 and detailed in Section 3.1 page 39.

- First, it is commonly used for testing the regularity of an interval matrix $\boldsymbol{A}$ (are all floating-point matrices in $\boldsymbol{A}$ invertible?) or for the verification of the solution of system of linear equations with interval coefficients. In both cases, one may use a particular case of interval matrix multiplication, like in the computation of $R\boldsymbol{A} - I$, where $\boldsymbol{A}$ is an interval matrix, R is an approximate floating-point inverse of the matrix of midpoints of $\boldsymbol{A}$, and I is the identity matrix (see [Rum10]).
- Second, it could be used as a substitute to the `sgemm` or `dgemm` functions, which compute floating-point matrix products. Actually, the corresponding interval matrix product provides the same result as the classical floating-point matrix product plus componentwise bounds on the computation errors. Such a substitute has to present a small overhead if we want it to be adopted by users accustomed to highly optimized BLAS libraries.
- Third, algorithms for interval matrix multiplication are simple enough. In particular, the computation of a single component of the result matrix can be performed in such a way that a given input component is used exactly once. This removes from consideration the so-called effect of variable dependency that tends to dramatically increase the width of the computed intervals. Linear algebra operations at a higher level, like Gaussian elimination or matrix factorizations, undergo the variable dependency problem.
- Finally, algorithms for interval matrix multiplication are complex enough. The algorithms we have chosen use directed rounding modes, and this raises some constraints on the code as the inclusion property has to be verified. Moreover, it allows one to use variants of interval arithmetic operations that offer tradeoff between the accuracy of the result and the number of floating-point operations.

Multi-core processors

Besides the accuracy of the computed result, our other main goal is to demonstrate, through an actual implementation, that it is possible to perform an interval matrix multiplication in parallel for a cost, in terms of execution time, that is comparable to the cost of a numerical matrix product, for any matrix dimension and any number of threads. This implementation cannot be tested on all kind of platforms, however, and we will target only the current multi-core x86 processors.

Many reasons lead to the selection of this target. From a general standpoint, multi-core processors are the current direction taken by constructors to provide more computing power. Increasing the clock frequency at which processors operate has long been the favorite way to accelerate computing, but it has come to an end due to unsustainable power consumption and heat dissipation. The multi-core approach, which consists in multiplying the number of processing units, allows one to increase the computing power at a fixed clock frequency.

From the point of view of the usage of these processors, let us note that the current trend for scientific computing is to use machines based on commodity multi-core processors. For instance, 5 out of the 10 top supercomputers in the 43th list (June 2014) of the TOP500 project, which ranks some of the most powerful computers that are mainly used for academic research, are made up of x86 processors, sometimes backed by accelerators. The low cost of individual units affords the opportunity to assemble them by thousands in such supercomputers. In this document, we will study the efficiency of our implementations for a platform equipped with only one or a handful of such multi-core processors. This can be considered as the node level of the supercomputers mentioned above.

In terms of hardware architecture, multi-core processors can be described as a set of general-purpose processors, the *cores*, that share the same memory bus. This multiplication of processing units allows of more task parallelism. Devising and evaluating novel ways of expressing, in terms of tasks, the parallelism of internal operations in linear algebra algorithms is an ongoing work and an active field of research. One may cite, for instance, the Plasma [KLY+13] or the Flame [VCv+09] projects. One of our objectives is to determine how the ideas of this field can be adapted to the interval case.

Document summary

This document is divided in two parts. The first part deals with the problem of the error analysis of algorithms for interval matrix multiplication.

- Chapter 1 compares many ways of evaluating errors when computing with intervals.

  Motivation. Different authors of interval algorithms use different metrics for measuring the accuracy of their results. This makes it difficult to compare the results in different articles. Moreover, in theoretical papers, the numerical experiment is often considered as a simple verification step for the proposed algorithm, and the choice of the metric for interval error is seldom discussed.

  Our contribution. We settle a coherent framework for the measure and analysis of interval error. The different metrics are related together and their relative merits are made explicit. Finally, we motivate the choice of two of them, the Hausdorff distance and the ratio of radii, for the subsequent analyses of relative errors in interval matrix products. We also introduce the notion of *relative accuracy* for the quantification of the intrinsic accuracy of a given interval.

- In Chapter 2, we focus on three different algorithms for the inner product of interval vectors, which compute overestimates of the exact inner products with different accuracies and at different computational costs, and we analyze their error when exact arithmetic is used. For every instance, we check that these inner products verify the inclusion property.

  Motivation. The forward error in interval matrix multiplication can be bounded componentwise. The componentwise error analysis then reduces to the analysis of inner products of interval vectors.

  Our contribution. In contrast to previous error analyses, which restrict themselves to input interval with homogeneous accuracies, we show how the inhomogeneity of the input relative accuracy affects the accuracy of the result. We also detail how the relative Hausdorff error of the result varies when the relative accuracies of the input vary. Moreover, we introduce a new approximate inner product, which is computationally lighter than the existing ones and we determine the conditions where it is as accurate as some of them.

- Chapter 3 presents the algorithms for the interval matrix multiplication, including a new one derived from our new inner product. Their roundoff errors are also analyzed.

  Motivation. The arithmetic error presented in the previous chapter is not the only source of error for the computation of a matrix product. Roundoff errors are present as soon as the actual computation uses finite precision numbers and a finite precision arithmetic. Indeed, the roundoff error may dominates the arithmetic error for very

thin or very thick input intervals. Nevertheless, the literature on interval matrix products usually neglects the roundoff error analysis.

Our contribution. We establish upper bounds on the roundoff error for the floating-point implementations of three interval matrix products. This work demonstrates that only a few formulas are needed for bounding the roundoff error in interval matrix products.

- In Chapter 4, we conduct for each algorithm the analysis of the global error, which results from the interaction of the arithmetic error and the roundoff error.

  Motivation. When comparing algorithms for interval matrix multiplication, based on a uniform upper bound on the sole arithmetic error, one may conclude that there is a clear tradeoff between the accuracy of the result and the number of floating-point operations. However, an algorithm that performs more floating-point operations is likely to undergo a larger floating-point error.

  Our contribution. We define the *global error* as the sum of the arithmetic error and the roundoff error. Using the bounds that are established in the previous two chapters, we determine the important factors that affect the behavior of this global error: the value and homogeneity of relative accuracies of inputs, the matrix dimension, and the working precision.

  In addition, we also point out the limits of our analysis, which is based on error bounds. These limits are obvious when comparing the value of the bound with the actual error measured in numerical experiments. The difficulty of such a measurement is that the actual error is not directly accessible, it requires a symbolic computation to determine the exact interval result. Nevertheless, the symbolic computation of all components would take too much time for matrices of large dimension. In order to conduct the numerical experiments in a reasonable time, we introduce the notion of *nearest midpoint rounding* of a real interval, we show how this representation can be computed with an arbitrary precision library in the case of interval inner product, and we determine the conditions under which the substitution of the exact value by its nearest floating-point representation gives meaningful results. To the best of our knowledge, it is the first time that such an experimental protocol is described and implemented.

  The conclusions of the experiment are also new. These experimental measures show that the computed bound is a good estimate of the actual error when the relative accuracies of the input are homogeneous and not too small. On the contrary, the relative Hausdorff error for accurate inputs, that is intervals with small radii compared to their midpoints, is usually far better than the value of the bound. The situation is even worse when the input matrices have components with a large variety of relative accuracies. In that case, the bound on the global error tends to infinity whereas the observed behavior with the chosen random dataset stays close to its behavior in the case of moderate inhomogeneity.

  The outcome of this work is a clear understanding of the domains where each algorithm may produce a more accurate enclosure than its competitors.

The second part deals with the parallel implementation of algorithms for interval matrix multiplication.

- Chapter 5 lists the technical problems one may encounter when implementing interval algorithms.

Motivation. The interval algorithms we want to implement use intensively one of the directed rounding modes in order to guarantee that the inclusion property is verified. Nowadays, the directed rounding modes are supported by the commodity processors we are targeting, and high level languages, like C or FORTRAN, also provide portable means to switch between the different rounding modes. Unfortunately, several different layers of software interact between the source code and the executed program that actually computes the results. And most of them do not handle correctly or even support other rounding modes than the default rounding to nearest.

Our contribution. Most of the implementation issues detailed in this chapter are already mentioned in the literature. In many articles about interval matrix computation, they are too hastily dismissed, however. To the best of our knowledge, it is the first time that a violation of the directed rounding mode by an actual BLAS library is exemplified. Moreover, the fact that the non-determinism of multi-threaded execution may affect the correctness of implementation of interval matrix multiplication has never been reported before.

We also make clear the fact that the language and compiler designers, on the one hand, and interval algorithm implementers, on the other hand, do not have the same interpretation of the switching to a directed rounding mode. The first ones simply interpret it as a command for computing subsequent operations with the newly set rounding mode whereas the others expect to compute under- or over-estimates.

Considering the obstacles listed in this chapter, we come to the conclusion that it is not safe to implement the algorithms for interval matrix multiplication in the way that was intended by their authors, that is, by calling an external BLAS library to compute the numerical matrix products.

- We detail a formal approach to the efficient, and correct, parallel implementation of interval matrix multiplications in Chapter 6.

Motivation. The previous chapter shows that an extreme care is needed if we want to guarantee that the inclusion property is verified at each step of the computation. Moreover, the underlying hardware has to be taken into account from the start of the implementation if a high level of performance is expected.

Our contribution. We state the implementation goals we want to reach, motivate them and prioritize them in a different order than other authors of algorithms for interval matrix multiplication. Our most important objectives are the correctness, the sequential efficiency and the strong scalability even at the expense of the ease of implementation. Actually, the efficient implementation of numerical algorithms for multi-core targets is an active field of research. Several possibilities are currently explored for dense linear algebra. We explain here how Gustavson's ideas about block matrix and blocking algorithms can be adapted to the case of interval matrices. We also quantify the implementation goals, when possible, and describe the experimental protocol that is used to determine whether our implementation reaches the prescribed objectives.

- We exemplify the application of the methodology described in the previous chapter with the implementation of the simple, yet complex enough, `MMMul2` algorithm for matrix multiplication. Chapter 7 deals with the implementation of the computing kernels that process block sub-matrices.

Motivation. When a numerical algorithm is given, the high performance of the program that implements this algorithm comes from the efficient use of the processing units and the efficient use of the memory. Current multi-core processors involve a complex hierarchy of processing units built onto a complex hierarchy of memory units.

Our contribution. Having chosen to implement block algorithms, we can distinguish two levels in the computation: the inner block level and the super-block level. Fortunately, this distinction matches the inner and inter-core levels. So, we describe in this chapter the implementation of the code performing the inner block computation on a single core in a single execution thread. We show how to exploit the instruction parallelism of the algorithm. To this end, we build a simple model of the core architecture, from which we compute a lower bound on the execution time. The comparison of the measured execution times against this lower bound allows us to check that the implementation reaches our prescribed threshold of sequential efficiency. Our implementation shows that common optimization techniques may be employed in our context, but most of them cannot be automatically applied by current compilers. Additionally, we guarantee the correctness of the implementation with respect to the rounding mode by systematically setting the correct rounding mode at the beginning of each block computation.

- Chapter 8 deals with the measure of the performance of the multi-threaded execution of our implementation on a multi-core platform.

Motivation. The competition in the field of dense numerical linear algebra is intense, and many processor vendors (for instance, Intel MKL, AMD ACML, or IBM ESSL) or academic competitors propose highly optimized BLAS libraries. In the interval field, however, interval linear algebra libraries are less common. One can cite the discontinued Profil-BIAS [Knü94], as well as INTLAB [Rum99b]. The latter relying on calls to numerical BLAS libraries for the matrix computations, it is thus subject to the correctness concerns listed in Chapter 5. Implementing an interval matrix product that does not use an external BLAS library and whose parallel performances are on par with the performances of up-to-date numerical BLAS library is challenging.

Our contribution. We choose to implement the classical triple nested loop algorithm at the super-block level and to control the multithreaded execution with OpenMP annotations. This approach combines the simplicity of the algorithm with the portability of OpenMP. The experimental measures show that the chosen approach provides sequential performance and scalability that are comparable with the ones of the `dgemm` function from the Intel's MKL library.

Finally, the main results of this work are recalled and some possible extensions and applications are detailed in the last chapter.

Part I

Error Analysis

Measuring error in interval linear algebra

The first part of this document states the underlying principles that make possible midpoint-radius algorithms for interval matrix multiplication. Two such algorithms are studied thoroughly. The principles being clear, we exploit them and propose a new interval matrix product that requires a smaller amount of computation.

At the same time, we address the problem of the accuracy of these algorithms. The error analyses presented in the literature assume exact arithmetic and neglect floating-point error. For example, Rump establishes in [Rum99a] that the radius of the product $\widetilde{R_3}$ computed by his algorithm (`IIMul3`, presented page 43) is less than 1.5 times larger than the exact radius R. This agrees with numerical experiments for sufficiently large intervals (see Figure 1). Rump also notices that the bound no longer holds if the inputs are too tight (left part of the graph). As detailed in depth in this part, this effect is due to roundoff errors.

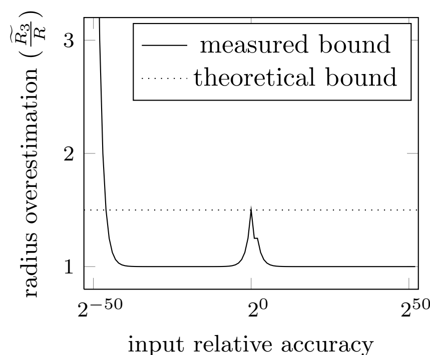

Figure 1: Radius Overestimation of the `IIMul3` Algorithm.

Our main goal is to present a complete analysis of the forward error, taking into account all sources of error. In fact, given an interval matrix multiplication algorithm and given its floating-point implementation, we can decompose the global error on the computed result into two distinct parts. One part represents the algorithmic overestimation of the product and the second part is the roundoff error of the floating-point calculation. Both parts can be studied separately, the global error being their mere sum. Figure 2 illustrates this idea. In the global error analysis, we expose the relevant parameters, and we use them to define criteria for choosing the best algorithm for a given input.

This part on error analysis is organized as follows. Since the error analysis depends on the metric and since several ones are possible, Chapter 1 discusses the advantages of the metrics

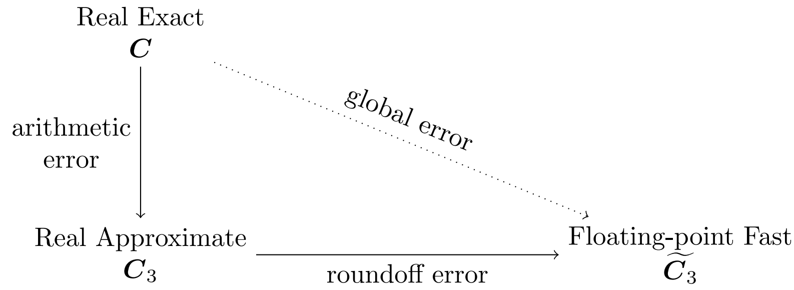

Figure 2: Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be two interval matrices of compatible sizes. The exact product $\boldsymbol{C} = \boldsymbol{AB}$ is an interval matrix that is mathematically defined, $\boldsymbol{C}_3$ is an approximate product like those defined in Chapter 3, and $\widetilde{\boldsymbol{C}_3}$ is the approximate result computed with floating-point arithmetic. The computation is such that $\boldsymbol{C} \subseteq \boldsymbol{C}_3 \subseteq \widetilde{\boldsymbol{C}_3}$. The global error between $\boldsymbol{C}$ and the computed floating-point interval $\widetilde{\boldsymbol{C}_3}$ is the sum of the arithmetic error between $\boldsymbol{C}$ and $\boldsymbol{C}_3$ and the roundoff error between $\boldsymbol{C}_3$ and $\widetilde{\boldsymbol{C}_3}$.

used in this document, and how they relate to each others. Next, the formulas for computing an enclosure of an interval inner product are presented in Chapter 2, along with an analysis of their arithmetic error. Chapter 3 contains a floating-point error analysis of interval matrix products based on these approximate inner products. Taking advantage of the possibility to return an interval enclosure of the result larger than the minimal enclosure, we also present a new and computationally lighter algorithm for interval matrix product. Finally in Chapter 4, we assemble the arithmetic and floating-point error analyses into a global error analysis, and we conduct numerical experiments to assess the pertinence of the theoretical bounds with respect to the actual error.

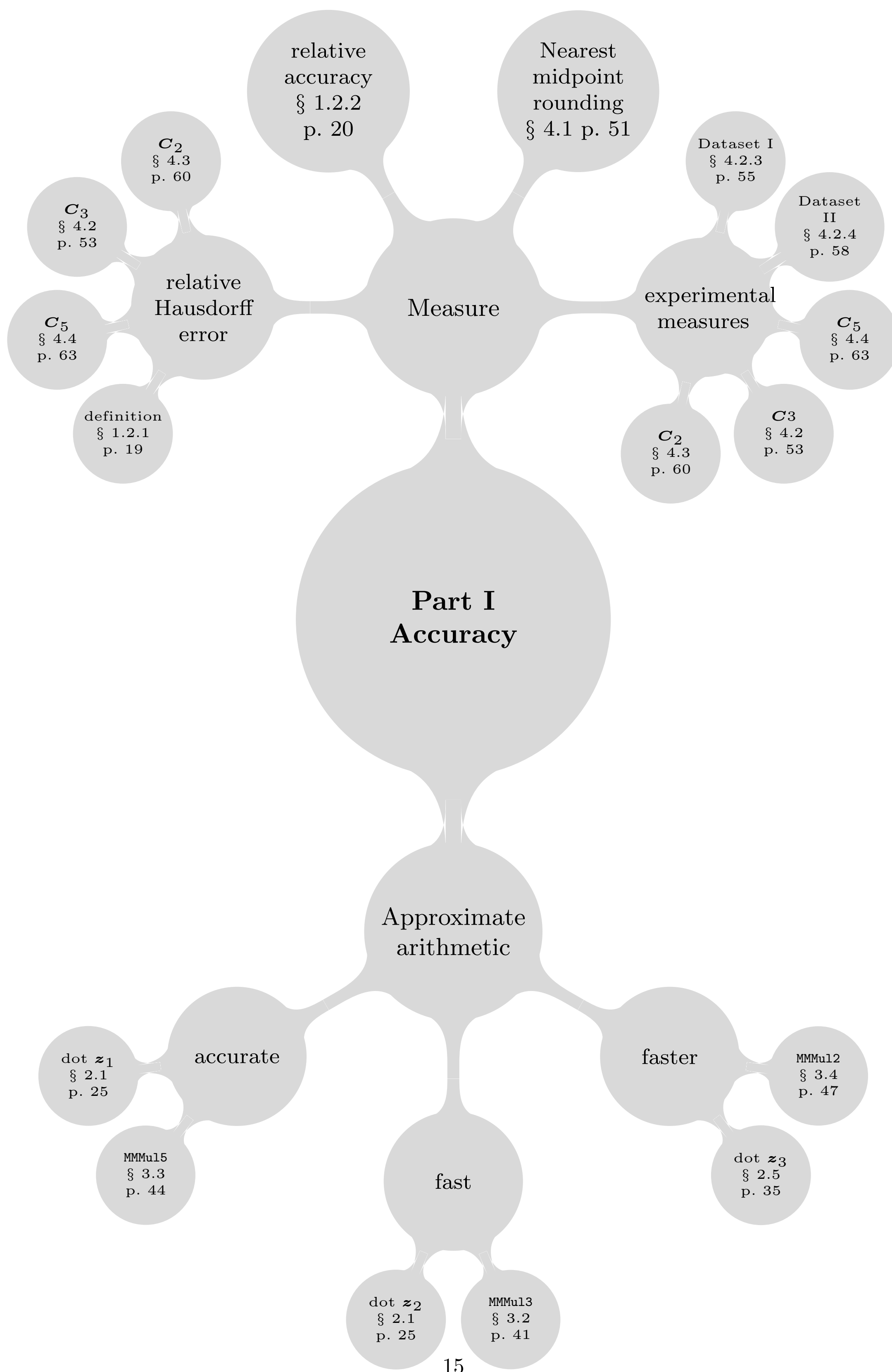

Figure 3: Synopsis of Part I.

Chapter 1

Metric on the set of real intervals

The correctness of algorithms that compute with intervals depends on the respect of the inclusion property. So, for a given problem, different algorithms may give different results, and each output is an acceptable solution as long as it contains the mathematically exact result. This raises the problem of comparing the computed approximates. When the exact solution is a real point, several measures of the distance to the exact result have been proposed: for instance, Kulpa and Markov define relative extent [KM03], Rump defines relative precision [Rum99a]. Another possibility is the relative approximation error proposed by Kreinovich [Kre13]. When the exact solution itself is an interval, the ratio of radii is often used. In this chapter, we discuss the possible choices for such metrics and we introduce the ones that will be used in the next chapters.

We advocate, for its precision and its mathematical properties, the use of the *Hausdorff metric* when measuring absolute error as well as relative error between two intervals. We also show how the much commonly used ratios of radii simplify the analysis with the Hausdorff distance in the case of nested intervals. But as will be seen in the analysis of experimental global error in Chapter 4, this simplicity may mask some interesting phenomena that the analysis with the Hausdorff distance reveals.

We also introduce the notion of *relative accuracy* for quantifying the amount of information that an interval conveys with respect to an unknown exact value it encloses. This measure is similar, yet not equivalent, to the relative precision, the relative extent, or the relative approximation error.

The first section below presents the midpoint-radius representation of an interval as well as useful relations in the case of inclusion. The next section defines and gives some characteristics of the Hausdorff distance and relative accuracy. The third section expresses the relations between the different possible alternatives.

1.1 Midpoint-radius representation

An interval $\boldsymbol{x}$ is traditionally represented by its infimum and supremum bounds $\boldsymbol{x} = [\underline{x}, \overline{x}]$. Following the standardized notation[1] for interval analysis [KNN+10], we note $\operatorname{mid} \boldsymbol{x} = (\overline{x} + \underline{x})/2$ the midpoint of $\boldsymbol{x}$ and $\operatorname{rad} \boldsymbol{x} = (\overline{x} - \underline{x})/2$ its radius. The midpoint-radius pair $\langle \operatorname{mid} \boldsymbol{x}, \operatorname{rad} \boldsymbol{x} \rangle$ is an alternative representation for $\boldsymbol{x}$.

While the infimum-supremum representation of an interval corresponds graphically to a mere line segment, we may describe a interval in midpoint-radius representation by a point in a plane

[1] Notations are summed up in the table page 1.

frame, as in Figure 1.1. In such a display, midpoints are reported along the abscissa axis and radii along the ordinate axis. So, intervals are represented by points lying in the upper half-plane. Let us consider the isosceles triangle whose base is on the abscissa axis and whose vertex is the point representing a given interval $\boldsymbol{x}$. The base of this triangle is the segment corresponding to the infimum-supremum representation of $\boldsymbol{x}$. Moreover, any point in this triangle represents a sub-interval of $\boldsymbol{x}$. Similarly, if we extend the edges of this triangle above $\boldsymbol{x}$, we obtain an infinite triangular sector, whose points correspond to intervals that contain $\boldsymbol{x}$ (Proposition 1.1 below).

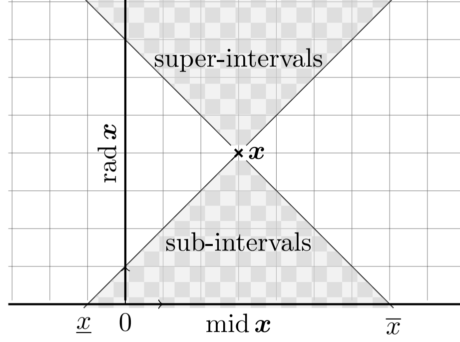

Figure 1.1: Interval Representations: Midpoint-Radius versus Infimum-Supremum.

As for scalar intervals, a matrix $\boldsymbol{X}$ with interval coefficients can be represented by the pair $\langle \operatorname{mid} \boldsymbol{X}, \operatorname{rad} \boldsymbol{X} \rangle$ of the numerical matrices of the midpoints $\operatorname{mid} \boldsymbol{X}$ and radii $\operatorname{rad} \boldsymbol{X}$ of its coefficients. Let m, n, and k be three integers, we note $\boldsymbol{A} = \langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle$ an m-by-k interval matrix, $\boldsymbol{B} = \langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}} \rangle$ a k-by-n interval matrix and $\boldsymbol{C} = \langle M, R \rangle$ the product $\boldsymbol{AB} = \{AB : A \in \mathbb{R}^{m \times k}$ and $A \in \boldsymbol{A}, B \in \mathbb{R}^{k \times n}$ and $B \in \boldsymbol{B}\}$.

We will use the following characterizations throughout the text. These relations are well-known; they appear, for instance, in [Neu90, Proposition 1.6.3].

Proposition 1.1. *Let x be a real number and let $\boldsymbol{x}$ and $\boldsymbol{y}$ be real intervals.*
The point x is in $\boldsymbol{x}$ if, and only if,

$$|x - \operatorname{mid} \boldsymbol{x}| \leq \operatorname{rad} \boldsymbol{x}. \tag{1.1}$$

The interval $\boldsymbol{x}$ is a sub-interval of $\boldsymbol{y}$ if, and only if,

$$|\operatorname{mid} \boldsymbol{y} - \operatorname{mid} \boldsymbol{x}| \leq \operatorname{rad} \boldsymbol{y} - \operatorname{rad} \boldsymbol{x}. \tag{1.2}$$

The situation $\boldsymbol{x} \subseteq \boldsymbol{y}$ is represented in Figure 1.2.

Proof. Inequality (1.1) translates to $|2x - (\overline{x} + \underline{x})| \leq \overline{x} - \underline{x}$, which is equivalent to $\underline{x} \leq x \leq \overline{x}$.

If $\boldsymbol{x} \subseteq \boldsymbol{y}$, then the points $\underline{x} = \operatorname{mid} \boldsymbol{x} - \operatorname{rad} \boldsymbol{x}$ and $\overline{x} = \operatorname{mid} \boldsymbol{x} + \operatorname{rad} \boldsymbol{x}$ are in $\boldsymbol{y}$, and applying (1.1), we have

$$|\underline{x} - \operatorname{mid} \boldsymbol{y}| \leq \operatorname{rad} \boldsymbol{y}$$
$$|\overline{x} - \operatorname{mid} \boldsymbol{y}| \leq \operatorname{rad} \boldsymbol{y}.$$

Moreover, $\max\{|\underline{x} - \operatorname{mid} \boldsymbol{y}|, |\overline{x} - \operatorname{mid} \boldsymbol{y}|\} = |\operatorname{mid} \boldsymbol{y} - \operatorname{mid} \boldsymbol{x}| + \operatorname{rad} \boldsymbol{x}$, so (1.2) holds.

Conversely, for all $x \in \boldsymbol{x}$, (1.1), (1.2), and a triangular inequality imply $|x - \operatorname{mid} \boldsymbol{y}| \leq \operatorname{rad} \boldsymbol{y}$. Thus, $x \in \boldsymbol{y}$. □

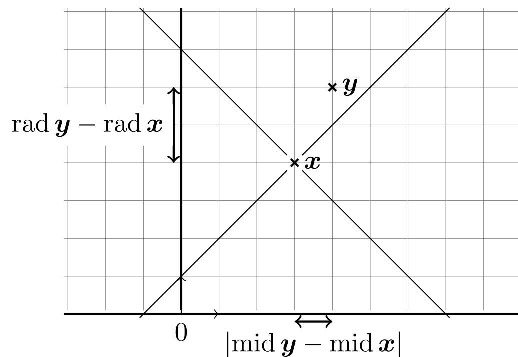

Figure 1.2: Inclusion in Midpoint-Radius Representation.

1.2 The choice of a metric

In order to compare several algorithms and implementations for interval matrix multiplication, we first have to choose a metric to quantify their numerical quality. No consensus has been reached about this matter, and different authors employ different metrics. For example, in [Knü94], Knüppel simply compares the width of the numerical results obtained with different libraries. Similarly, tables of radius values are displayed in [OOO11]. However, such an absolute quantification is relevant only when all midpoint components are of the same order of magnitude. In [Rum99a, OO05, OORO12], the overestimation of the computed intervals is measured with ratio of radii. Likewise, Nguyen uses equivalent ratios of width in [Ngu11]. As a last example, Rump uses in [Rum12] both ratio of radii and ratio of relative precision (see Definition 1.3 below). We will see that all these relative metrics are somewhat equivalent, but their diversity hinders direct comparison.

1.2.1 Hausdorff distance

For a given problem that has an exact solution x in a given normed space X and a given numerical algorithm that computes an approximate solution $y \in X$, the classical measure for the accuracy of the computation is the *relative forward error*, defined as $\frac{||x-y||}{||x||}$. Unfortunately, the set $\mathbb{IR}$ of real intervals and the set $\mathbb{IR}^n$ of rectangular boxes are not vector spaces[2]. For instance, only tight intervals, that is intervals that contain a unique real number, have an inverse for the addition. However, $\mathbb{IR}$ endowed with the Hausdorff distance is a metric space[3] [Neu90, Proposition 1.7.2]. It is therefore possible to study convergence and limits of sequences of intervals. Moreover, the Hausdorff distance has a very simple expression on intervals in midpoint-radius representation.

Definition 1.1. The *Hausdorff distance* between two intervals $\boldsymbol{x}$ and $\boldsymbol{y}$ is

$$\mathrm{d}(\boldsymbol{x}, \boldsymbol{y}) = |\mathrm{mid}\, \boldsymbol{x} - \mathrm{mid}\, \boldsymbol{y}| + |\mathrm{rad}\, \boldsymbol{x} - \mathrm{rad}\, \boldsymbol{y}|. \tag{1.3}$$

The measure[4] $\mathrm{d}(\boldsymbol{x}, 0)$ associated with the Hausdorff distance is the *absolute value* $|\boldsymbol{x}|$ (named also *magnitude*) of an interval. It is defined by $|\boldsymbol{x}| = |\mathrm{mid}\, \boldsymbol{x}| + \mathrm{rad}\, \boldsymbol{x}$ and is the maximum

[2]However, we still call *interval vectors* the elements of $\mathbb{IR}^n$ and *interval matrices* those of $\mathbb{IR}^{m\times n}$.

[3]This is the only occurrence of the term in its mathematical sense. In the rest of the document, *metric* simply denotes a system of measurement.

[4]Here, in the mathematical sense. Everywhere else, *measure* means measurement.

of the absolute values of the points in $\boldsymbol{x}$. For algorithms that compute with interval matrices and interval vectors, we will analyze computational errors componentwise with the Hausdorff distance.

In the case where $\boldsymbol{y}$ is a approximation of an interval value $\boldsymbol{x}$, the absolute forward error measured with the Hausdorff distance can be seen as the sum of the midpoint drift plus the radius overestimation (see Figure 1.2). We also define the *relative Hausdorff error* for a computed approximate interval $\boldsymbol{y}$ of an exact interval value $\boldsymbol{x}$ as the following relative forward error:

$$\frac{\mathrm{d}(\boldsymbol{x}, \boldsymbol{y})}{\mathrm{d}(\boldsymbol{x}, 0)} = \frac{|\boldsymbol{x} - \boldsymbol{y}|}{|\boldsymbol{x}|}.$$

If the computation of $\boldsymbol{y}$ satisfies the inclusion principle, then $\boldsymbol{x} \subseteq \boldsymbol{y}$ and Inequality (1.2) allows us to simplify the error analysis with this metric by using the following upper bound:

$$\frac{\mathrm{d}(\boldsymbol{x}, \boldsymbol{y})}{\mathrm{d}(\boldsymbol{x}, 0)} \leq 2\left(\frac{\operatorname{rad} \boldsymbol{y}}{\operatorname{rad} \boldsymbol{x}} - 1\right). \tag{1.4}$$

1.2.2 Relative accuracy

Suppose now that $x \neq 0$ is an unknown exact real value and that we can compute two different interval enclosures $\boldsymbol{y}$ and $\boldsymbol{z}$ of x. If $\boldsymbol{y}$ contains both positive and negative numbers, then we cannot conclude about the sign of x. On the contrary, if $\boldsymbol{z}$ does not contain 0, it guarantees the sign of x. In this case, we may say that $\boldsymbol{z}$ contains more information about x than $\boldsymbol{y}$. The notion of relative accuracy is a measure of (the lack of) this information.

Definition 1.2. The *relative accuracy* of an interval $\boldsymbol{x}$ is the quantity

$$\operatorname{racc}(\boldsymbol{x}) = \frac{\operatorname{rad} \boldsymbol{x}}{|\operatorname{mid} \boldsymbol{x}|}$$

if $\operatorname{mid} \boldsymbol{x} \neq 0$. If $\operatorname{mid} \boldsymbol{x} = 0$, we note $\operatorname{racc}(\boldsymbol{x}) = +\infty$.

The relative accuracy provides a means to characterize intervals that do or do not contain zero. It can also be used to bound the absolute value of the ratio between any point of $\boldsymbol{x}$ and $\operatorname{mid} \boldsymbol{x}$. These assertions are made explicit by the next proposition.

Proposition 1.2. *Let $\boldsymbol{x}$ be a real interval. The relative accuracy of $\boldsymbol{x}$ has the following properties.*

1. $0 \notin \boldsymbol{x}$ *if, and only if,* $\operatorname{racc}(\boldsymbol{x}) < 1$.
2. *If* $\operatorname{mid} \boldsymbol{x} \neq 0$*, then* $\mathrm{d}(\boldsymbol{x}, 0) = (1 + \operatorname{racc}(\boldsymbol{x}))|\operatorname{mid} \boldsymbol{x}|$.

Proof. Property 1 is a consequence of (1.1). Property 2 is a consequence of (1.3). □

The previous proposition implies that when $\operatorname{racc}(\boldsymbol{x}) \ll 1$, the elements of $\boldsymbol{x}$ do not differ much from $\operatorname{mid} \boldsymbol{x}$. Conversely, if $1 < \operatorname{racc}(\boldsymbol{x})$, some elements in $\boldsymbol{x}$ have a sign opposite to the sign of $\operatorname{mid} \boldsymbol{x}$. However, even in that case, $|x| \leq (1 + \operatorname{racc}(\boldsymbol{x}))|\operatorname{mid} \boldsymbol{x}|$ for all $x \in \boldsymbol{x}$.

If $\boldsymbol{x}$ is an interval enclosure of a real point x, the following property holds.

Proposition 1.3. *Let $\boldsymbol{x}$ be an interval such that $0 \notin \boldsymbol{x}$. Let x be a point of $\boldsymbol{x}$. We have, for all $\tilde{x}$ in $\boldsymbol{x}$*

$$\frac{|x - \tilde{x}|}{|x|} \leq \frac{2 \operatorname{racc}(\boldsymbol{x})}{1 - \operatorname{racc}(\boldsymbol{x})}. \tag{1.5}$$

Proof. Since x and $\tilde{x}$ are two points of $\boldsymbol{x}$, their distance is bounded by the width of $\boldsymbol{x}$, and we have

$$\frac{|x - \tilde{x}|}{|\operatorname{mid} \boldsymbol{x}|} \leq 2 \operatorname{racc}(\boldsymbol{x}). \tag{1.6}$$

Moreover, $|x| \geq |\operatorname{mid} \boldsymbol{x}| - \operatorname{rad} \boldsymbol{x}$, so

$$\frac{|x|}{|\operatorname{mid} \boldsymbol{x}|} \geq 1 - \operatorname{racc}(\boldsymbol{x}). \tag{1.7}$$

Dividing (1.6) by (1.7) gives (1.5). □

So when the relative accuracy of the enclosure is known, we can bound the relative error of an interval enclosure with respect to the unknown real value being approximated. For instance, if $\operatorname{racc}(\boldsymbol{x}) \leq \frac{1}{5}$ then $\frac{|x-\tilde{x}|}{|x|} \leq \frac{1}{2}$, so the relative error between any element of $\boldsymbol{x}$ and the exact unknown value x is less than 50% (see Figure 1.3). In that case, we have $\frac{2}{3}|\operatorname{mid} \boldsymbol{x}| \leq |x| \leq 2|\operatorname{mid} \boldsymbol{x}|$, which

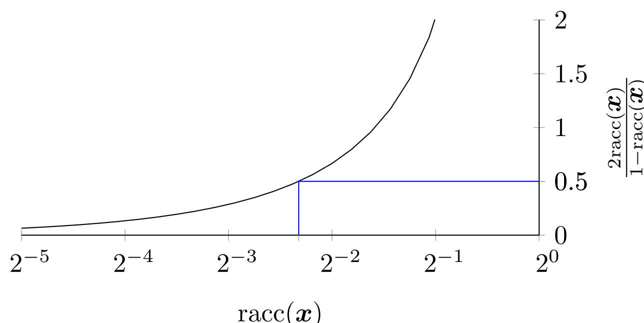

Figure 1.3: Relative Accuracy versus Maximum Relative Error.

means that $\operatorname{mid} \boldsymbol{x}$ and x are of the same order of magnitude. Again, if $\operatorname{racc}(\boldsymbol{x}) \ll 1$, then $\boldsymbol{x}$ determines the sign of the unknown value x and, according to (1.5), the relative error between x and any element of $\boldsymbol{x}$ cannot be much greater than $2 \operatorname{racc}(\boldsymbol{x})$.

When comparing nested intervals, we have the following property.

Proposition 1.4. *Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two real intervals.*
If $\boldsymbol{x} \subseteq \boldsymbol{y}$ and $\operatorname{racc}(\boldsymbol{x}) < 1$, then $\operatorname{racc}(\boldsymbol{x}) \leq \operatorname{racc}(\boldsymbol{y})$.

Proof. This is evident if $1 \leq \operatorname{racc}(\boldsymbol{y})$. Let assume that $\operatorname{racc}(\boldsymbol{y}) < 1$. As $\operatorname{racc}(\langle \operatorname{mid} \boldsymbol{z}, \operatorname{rad} \boldsymbol{z} \rangle) = \operatorname{racc}(\langle -\operatorname{mid} \boldsymbol{z}, \operatorname{rad} \boldsymbol{z} \rangle)$ for any interval $\boldsymbol{z}$, we can assume that $0 < \operatorname{mid} \boldsymbol{y}$, and therefore $0 < y$ for all $y \in \boldsymbol{y}$, without loss of generality.

Then, $\boldsymbol{x} \subseteq \boldsymbol{y}$ implies $\operatorname{rad} \boldsymbol{x} \leq \operatorname{rad} \boldsymbol{y}$. Thus, the property is true if $\operatorname{mid} \boldsymbol{y} \leq \operatorname{mid} \boldsymbol{x}$.

Let assume that $\operatorname{mid} \boldsymbol{x} < \operatorname{mid} \boldsymbol{y}$. Since $\boldsymbol{x} \subseteq \boldsymbol{y}$, we have $\underline{y} \leq \underline{x}$, thus

$$\operatorname{rad} \boldsymbol{x} + \operatorname{mid} \boldsymbol{y} - \operatorname{mid} \boldsymbol{x} \leq \operatorname{rad} \boldsymbol{y}. \tag{1.8}$$

Moreover, we know that $0 \notin \boldsymbol{x}$, so $0 < \operatorname{rad} \boldsymbol{x} < \operatorname{mid} \boldsymbol{x}$, and, for all positive r

$$\frac{\operatorname{rad} \boldsymbol{x}}{\operatorname{mid} \boldsymbol{x}} \leq \frac{\operatorname{rad} \boldsymbol{x} + r}{\operatorname{mid} \boldsymbol{x} + r}. \tag{1.9}$$

Setting $r = \operatorname{mid} \boldsymbol{y} - \operatorname{mid} \boldsymbol{x}$ in (1.9) and using (1.8), we have

$$\frac{\operatorname{rad} \boldsymbol{x}}{\operatorname{mid} \boldsymbol{x}} \leq \frac{\operatorname{rad} \boldsymbol{y}}{\operatorname{mid} \boldsymbol{y}},$$

which proves the proposition. □

Proposition 1.4 shows that the relative accuracy is non-decreasing on intervals that do not contain zero. Figure 1.4 illustrates this behavior for positive intervals, that is, intervals $\boldsymbol{x}$ such that $0 < x$ for all $x \in \boldsymbol{x}$. Consequently, for two computed intervals $\boldsymbol{y}$ and $\boldsymbol{z}$ not containing zero

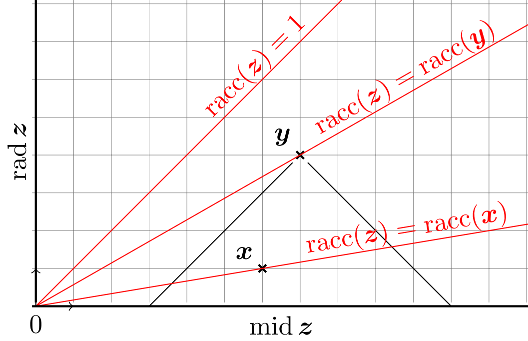

Figure 1.4: Relative Accuracy is Non-Decreasing on Positive Intervals.

and such that their computations respect the inclusion monotonicity, we will consider $\boldsymbol{z}$ as a better approximation of the exact unknown value than $\boldsymbol{y}$ if $\operatorname{racc}(\boldsymbol{z}) < \operatorname{racc}(\boldsymbol{y})$.

1.3 Relations between metrics

We now have two possibilities to quantify the numerical quality of an interval approximation. The relative accuracy is a relative measure dedicated to interval enclosures of exact *real* values, and the Hausdorff distance is an absolute measure for *interval* overestimation of interval quantities. We examine below the relations of these two metrics with some other measures employed in the literature.

The quantity $\operatorname{racc}(\boldsymbol{x})$ is closely related to what Rump calls the relative precision.

Definition 1.3 (from [Rum99a, Definition 2.3])**.** An interval $\boldsymbol{x}$ not containing 0 is said to be *of relative precision* e, $0 \leq e \in \boldsymbol{R}$, if $\operatorname{rad} \boldsymbol{x} \leq e|\operatorname{mid} \boldsymbol{x}|$. An interval containing 0 is said to be of relative precision 1.

The relative accuracy $\operatorname{racc}(\boldsymbol{x})$ is the minimum of such e when the interval does not contain zero. However, the two notions differ in several ways. First, the relative accuracy is a definite quantity, whereas to be *of relative precision* e is a property, as the number e is not unique. Second, the relative precision is bounded from above by 1, while the relative accuracy distinguishes between some intervals containing zero. In [Rum12], the above definition of the relative precision property is used for an interval but, with some ambiguity, the relative precision of interval matrices is implicitly defined as the quantity named here as the relative accuracy. So, the ratios of relative precision displayed in numerical tables are ratios of relative accuracy. Moreover, we will deal later with floating-point analysis, where the word *precision* is related to the number of digits used for floating-point numbers. This is why we choose to employ the word *accuracy* when

denoting the relative width of an interval. In doing so, we follow Higham [Hig02, § 1.4 Precision Versus Accuracy].

The fact that the relative accuracy becomes unbounded for symmetric intervals, that is intervals centered in 0, may be annoying in the computation. The above definition of the relative precision avoids this problem by bounding it from above by 1. Kreinovich proposes in [Kre13] an alternative definition for the relative error that remains bounded if the interval is not reduced to 0.

Definition 1.4. The *relative approximation error* r of an interval estimate $[\underline{x}, \overline{x}]$ is defined as

$$r = \min_{\tilde{x}\in[\underline{x},\overline{x}]} \max_{x\in[\underline{x},\overline{x}]} \frac{|x - \tilde{x}|}{|\tilde{x}|}$$

The relation between this quantity and the relative accuracy is clearer if we distinguish two cases.

Proposition 1.5. *When $\underline{x}$ and $\overline{x}$ have the same sign, then*

$$r = \frac{\overline{x} - \underline{x}}{|\underline{x} + \overline{x}|}.$$

When $\underline{x}$ and $\overline{x}$ are of different signs, i.e. when $\underline{x} < 0 < \overline{x}$, we have

$$r = \frac{\overline{x} - \underline{x}}{\max(|\underline{x}|, \overline{x})}.$$

The first case is equivalent to the relative accuracy, the second case is obviously finite.

Another related metric is the function $\chi : \mathbb{IR}\backslash 0 \to [-1, 1]$ defined by Ratschek [Rat72, RS82] as follows:

$$\chi(\boldsymbol{x}) = \begin{cases} \underline{x}/\overline{x}, & \text{if } |\underline{x}| \leq |\overline{x}|, \\ \overline{x}/\underline{x}, & \text{otherwise.} \end{cases}$$

The relative accuracy and Ratschek's metric verify $\operatorname{racc}(\boldsymbol{x}) = \frac{1-\chi(\boldsymbol{x})}{1+\chi(\boldsymbol{x})}$ and $\chi(\boldsymbol{x}) = \frac{1-\operatorname{racc}(\boldsymbol{x})}{1+\operatorname{racc}(\boldsymbol{x})}$.

However, we choose to have a single formula that applies to all cases. This simplifies the analysis and allows for computations without conditional branches. Moreover, the fact that the relative accuracy of an interval becomes unbounded when its midpoint tends to zero may be an indication of catastrophic cancellation. Anyway, when a zero result is expected, an absolute quantity such as the radius or the absolute value of the interval provides more relevant information.

Finally, concerning the relative forward error measured with the Hausdorff distance, note that the right-hand side of Inequality (1.4) is the ratio of radii, which is used in several papers, translated from $[1, +\infty)$ to $[0, +\infty)$, then scaled by 2. So, the ratio of radii gives a simple bound on the relative forward error and we will use it as such. However, such a simplification is based upon Inequality (1.2), which may overestimate the error on the computed midpoint. As exemplified in Chapter 4, this may lead to a pessimistic error bound.

1.4 Conclusion

In this chapter, we introduce two metrics for intervals. In the following, we use the relative accuracy for quantifying the numerical precision of the input intervals and the relative Hausdorff error as a measure of the error on the computed result.

Chapter 2

Accuracy of interval inner products

The main constraint when defining interval algorithms is to preserve the inclusion property. Besides this requirement, the accuracy of the computed result and a short computation time are other goals. Those are indeed common to all numerical algorithms. Having set how to measure the accuracy of the result in the previous chapter, we will now study how to trade accuracy for efficiency by using some approximate arithmetic operators.

In this chapter, we will state relations for real quantities and exact arithmetic. Algorithms dealing with floating-point quantities and their error analysis are not addressed until the next chapter.

2.1 Interval arithmetic variants

Most of the formulas below are well-known: (2.1) and (2.2) are in [Neu90, § 1.6 Rules for midpoint, radius, and absolute value], (2.4) is in [Rum99a, Definition 2.1], and (2.3) is Equation (2.12) with dimension one in [Ngu11]. We collect and present them here for completeness and in a form that will be useful for error analysis and further developments in Chapter 3.

In this section, $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ are real intervals.

The addition of intervals is straightforward (see Figure 2.1) and not much can be done to accelerate its computation.

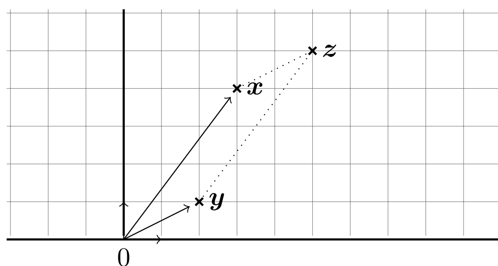

Figure 2.1: Interval Addition $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$ in Midpoint-Radius Representation.

Proposition 2.1. *Let $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$, the midpoint and radius of the sum are computed with:*

$$\operatorname{mid} \boldsymbol{z} = \operatorname{mid} \boldsymbol{x} + \operatorname{mid} \boldsymbol{y}, \qquad \operatorname{rad} \boldsymbol{z} = \operatorname{rad} \boldsymbol{x} + \operatorname{rad} \boldsymbol{y}. \tag{2.1}$$

Proof. Let $x \in \boldsymbol{x}$ and $y \in \boldsymbol{y}$. Using (1.1) for x and y and the triangular inequality, we have $|x + y - \operatorname{mid} \boldsymbol{x} - \operatorname{mid} \boldsymbol{y}| \leq \operatorname{rad} \boldsymbol{x} + \operatorname{rad} \boldsymbol{y}$. Thus, $x + y \in \boldsymbol{z}$.

Conversely, suppose there is $z \in \boldsymbol{z}$ such that $z \neq x + y$ for all $x \in \boldsymbol{x}$ and $y \in \boldsymbol{y}$. In other words, $z - x \notin \boldsymbol{y}$ for all $x \in \boldsymbol{x}$.

Suppose first that $\operatorname{mid} \boldsymbol{x} + \operatorname{mid} \boldsymbol{y} \leq z$, then for $x = \overline{x} = \operatorname{mid} \boldsymbol{x} + \operatorname{rad} \boldsymbol{x}$, we have $z - \overline{x} \notin \boldsymbol{y}$. Thus, using (1.1), $\operatorname{rad} \boldsymbol{x} + \operatorname{rad} \boldsymbol{y} \leq z - \operatorname{mid} \boldsymbol{x} - \operatorname{mid} \boldsymbol{y}$.

Suppose now that $z \leq \operatorname{mid} \boldsymbol{x} + \operatorname{mid} \boldsymbol{y}$, then for $x = \underline{x} = \operatorname{mid} \boldsymbol{x} - \operatorname{rad} \boldsymbol{x}$, we have $z - \underline{x} \notin \boldsymbol{y}$. Thus, using (1.1), $\operatorname{rad} \boldsymbol{x} + \operatorname{rad} \boldsymbol{y} \leq \operatorname{mid} \boldsymbol{x} + \operatorname{mid} \boldsymbol{y} - z$.

So, in both cases, $\operatorname{rad} \boldsymbol{z} \leq |z - \operatorname{mid} \boldsymbol{z}|$, which contradicts the hypothesis that z is in $\boldsymbol{z}$. □

The interval multiplication is more complex, but it can be replaced by simpler operations that compute an overestimated result.

Proposition 2.2. *Let $\boldsymbol{z} = \boldsymbol{x}\boldsymbol{y}$, the following formulas compute $\boldsymbol{z}$, $\boldsymbol{z}_1$, and $\boldsymbol{z}_2$ such that $\boldsymbol{z} \subseteq \boldsymbol{z}_1 \subseteq \boldsymbol{z}_2$*

$$\operatorname{mid} \boldsymbol{z} = \alpha + \mu, \qquad \operatorname{rad} \boldsymbol{z} = \beta + \gamma + \delta - |\mu|, \tag{2.2}$$

$$\operatorname{mid} \boldsymbol{z}_1 = \alpha + \nu, \qquad \operatorname{rad} \boldsymbol{z}_1 = \beta + \gamma + \delta - |\nu|, \tag{2.3}$$

$$\operatorname{mid} \boldsymbol{z}_2 = \alpha, \qquad \operatorname{rad} \boldsymbol{z}_2 = \beta + \gamma + \delta, \tag{2.4}$$

where

$$\alpha = \operatorname{mid} \boldsymbol{x} \operatorname{mid} \boldsymbol{y}, \qquad \alpha' = |\operatorname{mid} \boldsymbol{x}||\operatorname{mid} \boldsymbol{y}|,$$
$$\beta = |\operatorname{mid} \boldsymbol{x}| \operatorname{rad} \boldsymbol{y}, \qquad \gamma = \operatorname{rad} \boldsymbol{x} |\operatorname{mid} \boldsymbol{y}|,$$
$$\delta = \operatorname{rad} \boldsymbol{x} \operatorname{rad} \boldsymbol{y},$$
$$\mu = \operatorname{sign}(\alpha) \min\{\beta, \gamma, \delta\}, \qquad \nu = \operatorname{sign}(\alpha) \min\{\alpha', \beta, \gamma, \delta\},$$

and

$$\operatorname{sign}(\alpha) = \begin{cases} +1 & \text{if } \alpha > 0 \\ 0 & \text{if } \alpha = 0 \\ -1 & \text{if } \alpha < 0 \end{cases}.$$

Proof. First, we prove that $\boldsymbol{z} = \boldsymbol{x}\boldsymbol{y}$. Using (2.2), we have

$$\underline{\boldsymbol{z}} = \alpha + (\operatorname{sign}(\alpha) + 1)|\mu| - \beta - \gamma - \delta,$$
$$\overline{\boldsymbol{z}} = \alpha + (\operatorname{sign}(\alpha) - 1)|\mu| + \beta + \gamma + \delta.$$

Let $x \in \boldsymbol{x}$, we can write $x = \operatorname{mid} \boldsymbol{x} + \sigma_{\boldsymbol{x}}\ \lambda_x \operatorname{rad} \boldsymbol{x}$ with $\lambda_x \in [-1, 1]$, $\sigma_{\boldsymbol{x}} = +1$ if $\operatorname{mid} \boldsymbol{x} \geq 0$, and $\sigma_{\boldsymbol{x}} = -1$ if $\operatorname{mid} \boldsymbol{x} < 0$. Similarly, if we set $\sigma_{\boldsymbol{y}}$ to +1 when $0 \leq \operatorname{mid} \boldsymbol{y}$ and to -1 otherwise, then any y in $\boldsymbol{y}$ can be written $y = \operatorname{mid} \boldsymbol{y} + \sigma_{\boldsymbol{y}}\ \lambda_y \operatorname{rad} \boldsymbol{y}$, with $\lambda_y \in [-1, 1]$. So, the product may be written as follows

$$xy = \alpha + \sigma_{\boldsymbol{x}}\sigma_{\boldsymbol{y}}(\lambda_y\beta + \lambda_x\gamma + \lambda_x\lambda_y\delta).$$

If $0 \leq \alpha$, then $\sigma_{\boldsymbol{x}}\sigma_{\boldsymbol{y}} = +1$. Choosing x and y such that $\lambda_x = \lambda_y = +1$, we have $\max\{xy : x \in \boldsymbol{x}, y \in \boldsymbol{y}\} = \overline{z}$. The right choice for (λ_x, λ_y) in $\{(1, -1), (-1, 1), (-1, -1)\}$ gives the minimum value $\min\{xy : x \in \boldsymbol{x}, y \in \boldsymbol{y}\} = \underline{z}$. So, $\boldsymbol{x}\boldsymbol{y} = \boldsymbol{z}$ when $0 \leq \alpha$.

If $\alpha < 0$, then $\sigma\boldsymbol{x}\sigma\boldsymbol{y} = -1$. Choosing $\lambda_x = \lambda_y = +1$, we have $\min\{xy : x \in \boldsymbol{x}, y \in \boldsymbol{y}\} = \underline{z}$. Similarly, the right choice for (λ_x, λ_y) in $\{(1,-1), (-1,1), (-1,-1)\}$ leads to $\max\{xy : x \in \boldsymbol{x}, y \in \boldsymbol{y}\} = \overline{z}$. So, $\boldsymbol{xy} = \boldsymbol{z}$ when $\alpha < 0$.

Let us see now that $\boldsymbol{z} \subseteq \boldsymbol{z}_1 \subseteq \boldsymbol{z}_2$. Using the formulas (2.2–2.4), we have

$$|\mathrm{mid}\,\boldsymbol{z}_1 - \mathrm{mid}\,\boldsymbol{z}| = \mathrm{rad}\,\boldsymbol{z}_1 - \mathrm{rad}\,\boldsymbol{z} \tag{2.5}$$

and

$$|\mathrm{mid}\,\boldsymbol{z}_2 - \mathrm{mid}\,\boldsymbol{z}_1| = \mathrm{rad}\,\boldsymbol{z}_2 - \mathrm{rad}\,\boldsymbol{z}_1. \tag{2.6}$$

These equalities prove the inclusions by (1.2). □

In fact, a consequence of equalities (2.5) and (2.6) is that the three intervals share one of their extremal points for the absolute value. Indeed, since $\mathrm{mid}\,\boldsymbol{z}$, $\mathrm{mid}\,\boldsymbol{z}_1$, $\mathrm{mid}\,\boldsymbol{z}_2$, and α have the same sign, we have $\underline{z} = \underline{z_1} = \underline{z_2} = \alpha - (\beta + \gamma + \delta)$ if $\mathrm{mid}\,\boldsymbol{z} \leq 0$ (see Figure 2.2), and $\overline{z} = \overline{z_1} = \overline{z_2} = \alpha + (\beta + \gamma + \delta)$ if $0 \leq \mathrm{mid}\,\boldsymbol{z}$.

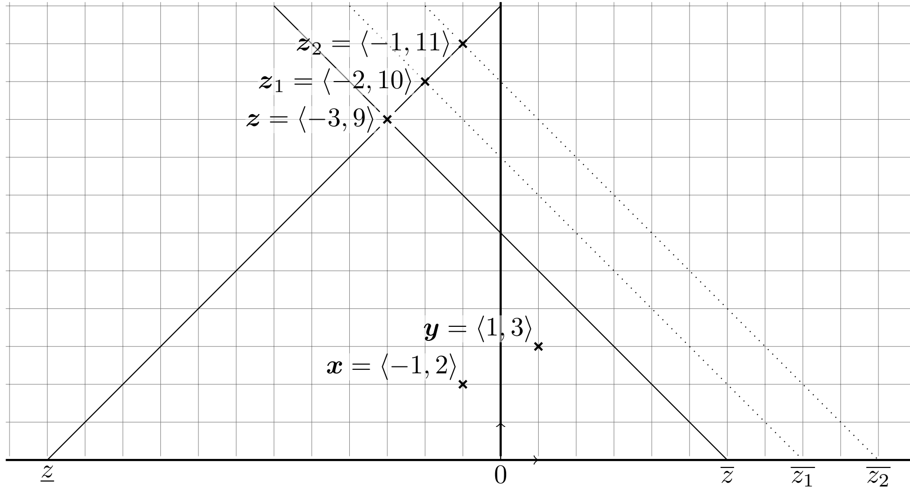

Figure 2.2: Products in Midpoint-Radius Representation.

Note that if $0 \notin \boldsymbol{z}$, then $0 \notin \boldsymbol{z}_1$ but $0 \in \boldsymbol{z}_2$ is possible. If fact, if $0 \notin \boldsymbol{z}$ then neither $\boldsymbol{x}$ nor $\boldsymbol{y}$ contains 0, so $\mathrm{rad}\,\boldsymbol{x} < |\mathrm{mid}\,\boldsymbol{x}|$ and $\mathrm{rad}\,\boldsymbol{y} < |\mathrm{mid}\,\boldsymbol{y}|$. Then, $\alpha = \alpha'$ and $\boldsymbol{z} = \boldsymbol{z}_1$. However, the following example shows that the situation may be different for z_2.

Example 2.1. Let $\boldsymbol{x} = \boldsymbol{y} = \langle 2, 1 \rangle$. Then $\boldsymbol{z} = \boldsymbol{z}_1 = \langle 5, 4 \rangle$ and $z_2 = \langle 4, 5 \rangle$. So $0 \notin \boldsymbol{z}$ and $0 \notin \boldsymbol{z}_1$, while $0 \in \boldsymbol{z}_2$.

In any case, the midpoints $\mathrm{mid}\,\boldsymbol{z}$, $\mathrm{mid}\,\boldsymbol{z}_1$, and $\mathrm{mid}\,\boldsymbol{z}_2$ have the same sign as the product $\mathrm{mid}\,\boldsymbol{x}\ \mathrm{mid}\,\boldsymbol{y}$.

2.2 Interval inner product and variants

We define now exact and approximate inner products of interval vectors that reduce to (2.2–2.4) when the dimension is 1.

Proposition 2.3. *Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two interval vectors of dimension k. The following formulas compute the inner product $\boldsymbol{z} = \boldsymbol{x}^T\boldsymbol{y}$, $\boldsymbol{z}_1$, and $\boldsymbol{z}_2$ such that $\boldsymbol{z} \subseteq \boldsymbol{z}_1 \subseteq \boldsymbol{z}_2$:*

$$\operatorname{mid}\boldsymbol{z} = \sum_{i=1}^{k}(\alpha_i + \mu_i), \qquad \operatorname{rad}\boldsymbol{z} = \sum_{i=1}^{k}(\beta_i + \gamma_i + \delta_i - |\mu_i|), \tag{2.7}$$

$$\operatorname{mid}\boldsymbol{z}_1 = \sum_{i=1}^{k}(\alpha_i + \nu_i), \qquad \operatorname{rad}\boldsymbol{z}_1 = \sum_{i=1}^{k}(\beta_i + \gamma_i + \delta_i - |\nu_i|), \tag{2.8}$$

$$\operatorname{mid}\boldsymbol{z}_2 = \sum_{i=1}^{k}\alpha_i, \qquad \operatorname{rad}\boldsymbol{z}_2 = \sum_{i=1}^{k}(\beta_i + \gamma_i + \delta_i), \tag{2.9}$$

where

$$\alpha_i = \operatorname{mid}\boldsymbol{x}_i \operatorname{mid}\boldsymbol{y}_i, \qquad \beta_i = |\operatorname{mid}\boldsymbol{x}_i|\operatorname{rad}\boldsymbol{y}_i,$$
$$\gamma_i = \operatorname{rad}\boldsymbol{x}_i|\operatorname{mid}\boldsymbol{y}_i|, \qquad \delta_i = \operatorname{rad}\boldsymbol{x}_i \operatorname{rad}\boldsymbol{y}_i,$$
$$\mu_i = \operatorname{sign}(\alpha_i)\min(\beta_i, \gamma_i, \delta_i), \qquad \nu_i = \operatorname{sign}(\alpha_i)\min(|\alpha_i|, \beta_i, \gamma_i, \delta_i),$$

for all $i = 1, \ldots, k$.

Proof. First, we prove $\boldsymbol{z} = \boldsymbol{x}^T\boldsymbol{y}$ by induction on the dimension of the vectors. If $\boldsymbol{x}$ and $\boldsymbol{y}$ are of dimension one, (2.7) reduces to (2.2). The induction step is true because, according to Proposition 2.1, midpoints and radii sum separately to give the midpoint and the radius of the sum.

Similarly, (2.8) and (2.9) extend (2.3) and (2.4), respectively, in higher dimension.

Let us see now that $\boldsymbol{z} \subseteq \boldsymbol{z}_1 \subseteq \boldsymbol{z}_2$. Using the formulas (2.7–2.9), we have

$$|\operatorname{mid}\boldsymbol{z}_1 - \operatorname{mid}\boldsymbol{z}| \le \operatorname{rad}\boldsymbol{z}_1 - \operatorname{rad}\boldsymbol{z} \tag{2.10}$$

and

$$|\operatorname{mid}\boldsymbol{z}_2 - \operatorname{mid}\boldsymbol{z}_1| \le \operatorname{rad}\boldsymbol{z}_2 - \operatorname{rad}\boldsymbol{z}_1. \tag{2.11}$$

These equalities prove the inclusions by (1.2). □

The definitions of the exact and approximate interval inner products in terms of components, as in the previous proposition, are useful for the error analysis, which is done in Section 2.3. The next corollary presents a formulation of the inner products defined in Proposition 2.3 in terms of inner products of real vectors. These formulas extend naturally to formulas for interval matrix products, which are presented in Chapter 3.

Corollary 2.4. *Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two interval vectors of dimension k. The following formulas compute the inner product $\boldsymbol{z} = \boldsymbol{x}^T\boldsymbol{y}$, $\boldsymbol{z}_1$, and $\boldsymbol{z}_2$ such that $\boldsymbol{z} \subseteq \boldsymbol{z}_1 \subseteq \boldsymbol{z}_2$:*

$$\operatorname{mid}\boldsymbol{z} = \operatorname{mid}\boldsymbol{x}^T\operatorname{mid}\boldsymbol{y} + \sum_{i=1}^{k}\mu_i, \qquad \operatorname{rad}\boldsymbol{z} = |\operatorname{mid}\boldsymbol{x}|^T\operatorname{rad}\boldsymbol{y} + \operatorname{rad}\boldsymbol{x}^T|\operatorname{mid}\boldsymbol{y}| + \operatorname{rad}\boldsymbol{x}^T\operatorname{rad}\boldsymbol{y} - \sum_{i=1}^{k}|\mu_i|, \tag{2.12}$$

$$\operatorname{mid}\boldsymbol{z}_1 = \operatorname{mid}\boldsymbol{x}^T\operatorname{mid}\boldsymbol{y} + \rho_{\boldsymbol{x}}^T\rho_{\boldsymbol{y}}, \qquad \operatorname{rad}\boldsymbol{z}_1 = |\operatorname{mid}\boldsymbol{x}|^T\operatorname{rad}\boldsymbol{y} + \operatorname{rad}\boldsymbol{x}^T|\operatorname{mid}\boldsymbol{y}| + \operatorname{rad}\boldsymbol{x}^T\operatorname{rad}\boldsymbol{y} - |\rho_{\boldsymbol{x}}|^T|\rho_{\boldsymbol{y}}|, \tag{2.13}$$

$$\operatorname{mid}\boldsymbol{z}_2 = \operatorname{mid}\boldsymbol{x}^T\operatorname{mid}\boldsymbol{y}, \qquad \operatorname{rad}\boldsymbol{z}_2 = |\operatorname{mid}\boldsymbol{x}|^T\operatorname{rad}\boldsymbol{y} + \operatorname{rad}\boldsymbol{x}^T|\operatorname{mid}\boldsymbol{y}| + \operatorname{rad}\boldsymbol{x}^T\operatorname{rad}\boldsymbol{y} \tag{2.14}$$

where μ_i are numbers defined by

$$|\mu_i| = \min\{|\operatorname{mid}\boldsymbol{x}_i|\operatorname{rad}\boldsymbol{y}_i, \operatorname{rad}\boldsymbol{x}_i|\operatorname{mid}\boldsymbol{y}_i|, \operatorname{rad}\boldsymbol{x}_i \operatorname{rad}\boldsymbol{y}_i\}, \tag{2.15}$$

$$\mu_i = \operatorname{sign}(\operatorname{mid}\boldsymbol{x}_i \operatorname{mid}\boldsymbol{y}_i)|\mu_i|, \tag{2.16}$$

and $\rho\boldsymbol{x}$ *and* $\rho\boldsymbol{y}$ *are interval vectors such that*

$$\rho\boldsymbol{x}_i = \operatorname{sign}(\operatorname{mid}\boldsymbol{x}_i)\min\{|\operatorname{mid}\boldsymbol{x}_i|, \operatorname{rad}\boldsymbol{x}_i\}, \tag{2.17}$$

$$\rho\boldsymbol{y}_i = \operatorname{sign}(\operatorname{mid}\boldsymbol{y}_i)\min\{|\operatorname{mid}\boldsymbol{y}_i|, \operatorname{rad}\boldsymbol{y}_i\}, \tag{2.18}$$

for all $i = 1, \ldots, k$*.*

Proof. Since $\operatorname{mid}\boldsymbol{x}^T\operatorname{mid}\boldsymbol{y} = \sum_{i=1}^k \alpha_i$, where the α_i's are defined as in Proposition 2.3, the expression of $\operatorname{mid}\boldsymbol{z}$ in (2.7) and (2.12) are equivalent. In the same way, the expression of $\operatorname{rad}\boldsymbol{z}$ in (2.12) is equivalent to the corresponding one in (2.7), the components β_i, γ_i, δ_i, and $|\mu_i|$ being summed separately.

Similarly, the midpoint and radius formulas in (2.13) and (2.14) are trivial rewriting of their expressions in (2.8) and (2.9), respectively. □

Note that equations (2.14) are the simplest ones. In equations (2.13), the corrections to $\operatorname{mid}\boldsymbol{z}_2$ and $\operatorname{rad}\boldsymbol{z}_2$ are simpler to compute compared to the corresponding terms in the exact formulas (2.12). Moreover, $\mu_i = \rho\boldsymbol{x}_i\rho\boldsymbol{y}_i$ unless $|\operatorname{mid}\boldsymbol{x}_i| < \operatorname{rad}\boldsymbol{x}_i$ and $|\operatorname{mid}\boldsymbol{y}_i| < \operatorname{rad}\boldsymbol{y}_i$. So, $\boldsymbol{z}_1$ computes the exact inner product $\boldsymbol{z}$ when interval coefficients of $\boldsymbol{x}$ and $\boldsymbol{y}$ do not contain zero, when $\operatorname{rad}\boldsymbol{x} = 0$, or when $\operatorname{rad}\boldsymbol{y} = 0$.

However, $\operatorname{mid}\boldsymbol{z}$ and $\operatorname{mid}\boldsymbol{z}_2$ (or even $\operatorname{mid}\boldsymbol{z}_1$ if some thick components contain zero) may have different signs, as shown by the following example.

Example 2.2. Let $\boldsymbol{x} = (\langle 1, 4\rangle, \langle -1, 2\rangle)^T$ and $\boldsymbol{y} = (\langle 1, 4\rangle, \langle 2, 2\rangle)^T$. Applying Proposition 2.4, we have $\boldsymbol{z} = \langle 1, 28\rangle$, $\boldsymbol{z}_1 = \langle -2, 31\rangle$, and $\boldsymbol{z}_2 = \langle -1, 34\rangle$ (see Figure 2.3). So, $\operatorname{mid}\boldsymbol{x}^T\operatorname{mid}\boldsymbol{y} = -1$ while $\operatorname{mid}\boldsymbol{z} = 1$.

This means that $\operatorname{mid}\boldsymbol{x}^T\operatorname{mid}\boldsymbol{y}$ is not always a good approximation of $\operatorname{mid}\boldsymbol{z}$.

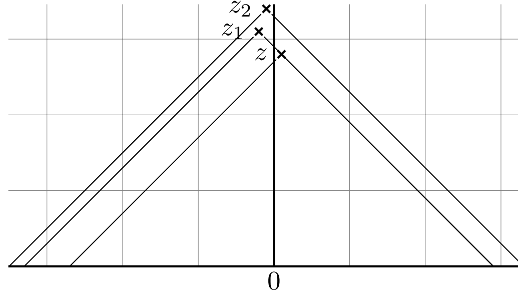

Figure 2.3: Exact and Approximate Inner Products.

2.3 Arithmetic error of interval inner products

Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two k-vectors of interval components. Let e, f, and λ such that

$$e = \max\{\operatorname{racc}(\boldsymbol{x}_i) : 1 \le i \le k\},$$
$$f = \max\{\operatorname{racc}(\boldsymbol{y}_i) : 1 \le i \le k\},$$
$$\lambda = \min\{\operatorname{racc}(\boldsymbol{x}_i)/e, \operatorname{racc}(\boldsymbol{y}_i)/f : 1 \le i \le k\}.$$

Then $\operatorname{racc}(\boldsymbol{x}_i) \in e[\lambda, 1]$ and $\operatorname{racc}(\boldsymbol{y}_i) \in f[\lambda, 1]$, for all $i = 1, \ldots, k$.

In order to express the relative arithmetic error of inner products, we need to bound relations between radii of exact and approximate inner products.

Proposition 2.5. *The exact inner product* $\boldsymbol{z} = \boldsymbol{x}^T\boldsymbol{y}$*, defined by* (2.12)*, the approximate inner product* $\boldsymbol{z}_1 \supseteq \boldsymbol{x}^T\boldsymbol{y}$*, defined by* (2.13)*, and the approximate inner product* $\boldsymbol{z}_2 \supseteq \boldsymbol{x}^T\boldsymbol{y}$*, defined by* (2.14) *verify*

$$a|\mathrm{mid}\,\boldsymbol{x}|^T|\mathrm{mid}\,\boldsymbol{y}| \le \mathrm{rad}\,\boldsymbol{z}, \tag{2.19}$$

$$\mathrm{rad}\,\boldsymbol{z}_1 - \mathrm{rad}\,\boldsymbol{z} \le b|\mathrm{mid}\,\boldsymbol{x}|^T|\mathrm{mid}\,\boldsymbol{y}|, \tag{2.20}$$

$$\mathrm{rad}\,\boldsymbol{z}_2 - \mathrm{rad}\,\boldsymbol{z} \le c|\mathrm{mid}\,\boldsymbol{x}|^T|\mathrm{mid}\,\boldsymbol{y}|, \tag{2.21}$$

where

$$a = \lambda\max\{e,f\} + \lambda\max\{\min\{e,f\},\lambda ef\}$$

$$b = \max\{\min\{e,f,ef\},1\} - 1 \qquad\qquad c = \min\{e,f,ef\}$$

Proof. We can rewrite the definition of $\mathrm{rad}\,\boldsymbol{z}$ in (2.7) in the following way:

$$\mathrm{rad}\,\boldsymbol{z} = \sum\nolimits_{i=1}^{k}(\epsilon_i + \zeta_i) \tag{2.22}$$

where, with the definitions of Proposition 2.3, $\epsilon_i = \max\{\beta_i,\gamma_i\}$ and $\zeta_i = \max\{\min\{\beta_i,\gamma_i\},\delta_i\}$. In fact, the quantities ϵ_i, ζ_i, and $|\mu_i|$ are the quantities β_i, γ_i, and δ_i in another order. So, $\epsilon_i + \zeta_i = \beta_i + \gamma_i + \delta_i - |\mu_i|$.

Then, using the definitions of e, f, and λ, we have for any $i = 1,\ldots,k$

$$\max\{\lambda e,\lambda f\}|\alpha_i| \le \epsilon_i,$$

$$\max\{\min\{\lambda e,\lambda f\},\lambda^2 ef\}|\alpha_i| \le \zeta_i.$$

Summing the previous inequalities for all i gives (2.19).

Let us prove (2.21). From (2.9) and (2.7), we derive $\mathrm{rad}\,\boldsymbol{z}_2 - \mathrm{rad}\,\boldsymbol{z} = \sum_{i=1}^k |\mu_i|$. From the definitions of e, f, and μ_i, we have $|\mu_i| \le \min\{e,f,ef\}|\alpha_i|$. Summing the last inequality for all $i \le k$ gives (2.21).

From (2.8) and (2.7), we derive $\mathrm{rad}\,\boldsymbol{z}_1 - \mathrm{rad}\,\boldsymbol{z} = \sum_{i=1}^k(|\mu_i| - |\nu_i|)$. First, note that, by definition, $c - 1 \le b$. If $\nu_i \ne \mu_i$, then $|\nu_i| = |\alpha_i|$. So, we have $|\mu_i| - |\nu_i| < (c-1)|\alpha_i|$. If $\nu_i = \mu_i$, then $|\mu_i| - |\nu_i| = 0$. In both cases, we have $|\mu_i| - |\nu_i| \le b|\alpha_i|$. Then, summing for i over $1,\ldots,k$ gives (2.20). ☐

Note that the upper bounds are tight, in the sense that they can be reached by $\mathrm{rad}\,\boldsymbol{z}_1 - \mathrm{rad}\,\boldsymbol{z}$, and $\mathrm{rad}\,\boldsymbol{z}_2 - \mathrm{rad}\,\boldsymbol{z}$. Figure 2.2 illustrates such a case. The lower bound on $\mathrm{rad}\,\boldsymbol{z}$ is also attained when $\lambda = 1$. However, it cannot be reached when $\lambda < 1$. Actually, by definition of e and f, there exists an index i_0 such that $\max\{\beta_{i_0},\gamma_{i_0}\} = \max\{e,f\}|\mathrm{mid}\,\boldsymbol{x}_{i_0}||\mathrm{mid}\,\boldsymbol{y}_{i_0}|$. So, $\lambda\max\{e,f\}|\alpha_{i_0}| < \epsilon_{i_0}$. According to (2.22), ϵ_{i_0} appears in the sum $\mathrm{rad}\,\boldsymbol{z}$, so $a|\mathrm{mid}\,\boldsymbol{x}|^T|\mathrm{mid}\,\boldsymbol{y}|$ is strictly less than $\mathrm{rad}\,\boldsymbol{z}$.

It is now possible to exhibit an upper bound for the relative arithmetic error of both approximate inner products.

Corollary 2.6. *Let* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *be two nonzero* k*-vectors of interval components. The exact inner product* $\boldsymbol{z} = \boldsymbol{x}^T\boldsymbol{y}$*, defined by* (2.12)*, the approximate inner product* $\boldsymbol{z}_1 \supseteq \boldsymbol{x}^T\boldsymbol{y}$*, defined by* (2.13)*, and the approximate inner product* $\boldsymbol{z}_2 \supseteq \boldsymbol{x}^T\boldsymbol{y}$*, defined by* (2.14) *verify*

$$\frac{\mathrm{d}(\boldsymbol{z}_1,\boldsymbol{z})}{\mathrm{d}(\boldsymbol{z},0)} \le \frac{2b}{a}, \tag{2.23}$$

$$\frac{\mathrm{d}(\boldsymbol{z}_2,\boldsymbol{z})}{\mathrm{d}(\boldsymbol{z},0)} \le \frac{2c}{a}, \tag{2.24}$$

where

$a = \lambda \max\{e, f\} + \lambda \max\{\min\{e, f\}, \lambda ef\},$

$b = \max\{\min\{e, f, ef\}, 1\} - 1, \qquad c = \min\{e, f, ef\}.$

Proof. Since $\boldsymbol{x}$ and $\boldsymbol{y}$ are different from 0, so is $\boldsymbol{z}$. Thus, $\mathrm{d}(\boldsymbol{z}, 0) \neq 0$.

Inequalities (1.4) and (2.20) imply $\frac{\operatorname{rad} \boldsymbol{z}_1}{\operatorname{rad} \boldsymbol{z}} - 1 \leq \frac{b}{a}$. Inequalities (1.4) and (2.21) imply $\frac{\operatorname{rad} \boldsymbol{z}_2}{\operatorname{rad} \boldsymbol{z}} - 1 \leq \frac{c}{a}$. Remember that, according to (1.4), $\frac{\mathrm{d}(\boldsymbol{z}_1, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)} \leq 2\left(\frac{\operatorname{rad} \boldsymbol{z}_1}{\operatorname{rad} \boldsymbol{z}} - 1\right)$ and a similar inequality holds for $\frac{\mathrm{d}(\boldsymbol{z}_2, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)}$. Thus, the previous inequalities imply respectively (2.23) and (2.24). □

2.4 Arithmetic error analysis

In this section, we analyze the arithmetic error of the approximate inner products $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$. Indeed, much knowledge can be extracted from the bounds (2.23) and (2.24) given by Corollary 2.6. As it is apparent from the expression of the bounds, the arithmetic error is sensible to the variation of several parameters: the upper bounds e and f on the relative accuracies of the input vectors and the dispersion of these relative accuracies, measured by the λ parameter. We show here that the worst cases are located on the line $e = f$ and that it is possible to simplify the upper bounds to different levels of detail, leading to more or less fine-grained analysis.

In the following analysis, we assume that $\lambda \neq 0$. The first two points deal with the variation of the bounds with e and f, the next one analyzes the behavior with respect to λ and the following ones relate to worst cases.

- First, if $e \leq 1$ or $f \leq 1$, then $b = 0$, so $\boldsymbol{z}_1 = \boldsymbol{z}$. In other words, $\boldsymbol{z}_1$ computes the exact inner product if one input vector, at least, does not intersect the axes, i.e. 0 does not belong to any of its components. This phenomenon was already noticed in Section 2.2 and is faithfully reflected by the bound (2.23). The situation is quite different for $\boldsymbol{z}_2$. The numerator c of the bound is zero if, and only if, $e = 0$ or $f = 0$.

  Therefore, for small relative accuracies, one can expect exact results when using inner product $\boldsymbol{z}_1$ and some overestimation when using inner product $\boldsymbol{z}_2$.

- Second, as $b \leq c$, the upper bound in (2.23) is always smaller than the upper bound in (2.24). Their difference $2(c - b)/a$ tends to zero when e or f grows and λ is a non-zero constant. Thus, for large relative accuracies, the bounds become indistinguishable.

- Third, notice that the parameter λ appears only in the denominator of the bounds. So, the bounds are at least one order of magnitude larger when λ losses one order of magnitude. For the limit case $\lambda = 0$, the upper bounds end up infinite, that is to say we have no upper bound at all.

- Fourth, the quantities $a = a(e, f, \lambda)$, $b = b(e, f)$, and $c = c(e, f)$ that appear in Corollary 2.6 depend on both parameters e and f. In Chapter 4, we will use the following simpler bounds that depend only on one of them.

  Proposition 2.7. *Let a, b, and c be defined as in Corollary 2.6. Assume that $f \leq e$ and $\lambda \neq 0$, we have*

$$\frac{b(e,f)}{a(e,f,\lambda)} \leq \frac{\max\{0, e-1\}}{\lambda e(1 + \max\{1, \lambda e\})} = \frac{b(e,e)}{a(e,e,\lambda)}, \tag{2.25}$$

$$\frac{c(e,f)}{a(e,f,\lambda)} \leq \frac{\min\{1, e\}}{\lambda(1 + \max\{1, \lambda e\})} = \frac{c(e,e)}{a(e,e,\lambda)}. \tag{2.26}$$

Proof. If $f \leq 1$, then $b(e, f) = 0$ and Inequality (2.25) holds. If $1 < f \leq e$, then $b(e, f) = f - 1$ and $a(e, f, \lambda) = \lambda e + \lambda f \max\{1, \lambda e\}$. So,

$$b(e, f)\ a(e, e, \lambda) = \lambda ef + \lambda ef \max\{1, \lambda e\} - \lambda e - \lambda e \max\{1, \lambda e\}$$

is less than

$$b(e, e)\ a(e, f, \lambda) = \lambda e^2 + \lambda ef \max\{1, \lambda e\} - \lambda e - \lambda f \max\{1, \lambda e\}.$$

All involved quantities being positive, this implies Inequality (2.25).

Likewise, if $f = 0$, then $c(e, f) = 0$ and Inequality (2.26) holds. Under the hypothesis $0 < f \leq e$, we have $c(e, f) = f \min\{1, e\}$. So,

$$c(e, f)\ a(e, e, \lambda) = \lambda ef \min\{1, e\} + \lambda ef \min\{1, e\} \max\{1, \lambda e\}$$

is less than

$$c(e, e)\ a(e, f, \lambda) = \lambda e^2 \min\{1, e\} + \lambda ef \min\{1, e\} \max\{1, \lambda e\}).$$

All involved quantities being positive, this implies Inequality (2.26). □

When $e \leq f$, it suffices to exchange e and f in the above inequalities.

We can interpret this result as follows. Given two interval vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of relative accuracies bounded by e and f respectively, inequality (2.25) shows that the bound $2b(e, f)/a(e, f, \lambda)$ on the relative Hausdorff error of the approximate inner product $\boldsymbol{z}_1$ is always less than the bound $2b(e, e)/a(e, e, \lambda)$ on the corresponding error if the maximum relative accuracies of the two vectors were both equal to $\max\{e, f\}$. The same is true for the inner product $\boldsymbol{z}_2$. In other words, for a given $E > 0$, the maxima of b/a and c/a under the constraints $e \leq f = E$ or $f \leq e = E$ are attained on the diagonal at the point $e = f = E$. This conclusion is also apparent in the isoline graphs displayed in Figure 2.4[1].

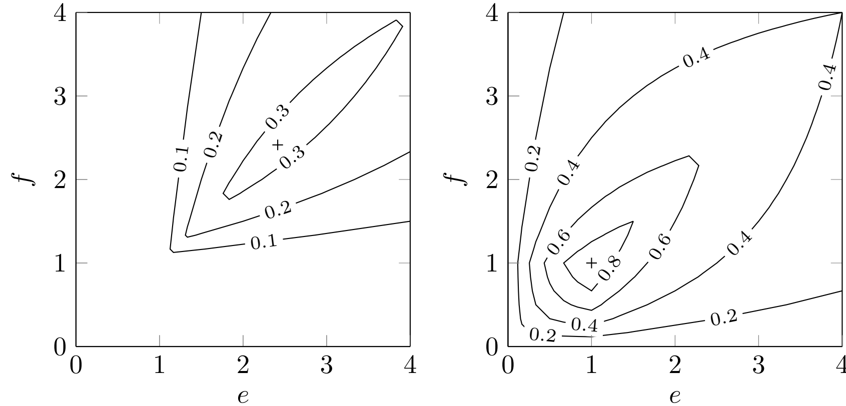

Figure 2.4: Upper Bounds on the Relative Arithmetic Error of Approximate Inner Products: (2.23) for $\boldsymbol{z}_1$ (left) and (2.24) for $\boldsymbol{z}_2$ (right) with $\lambda = 1$.

[1]The graph for $\boldsymbol{z}_2$ already appears in [KM03].

- Fifth, we focus now on the special case where $e = f$, that is on the diagonals of Figure 2.4. Rewriting (2.25) and (2.26), we have

$$\frac{\mathrm{d}(\boldsymbol{z}_1, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)} = 0, \qquad \frac{\mathrm{d}(\boldsymbol{z}_2, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)} \le \frac{e}{\lambda}, \qquad \text{if } e \le 1. \tag{2.27}$$

$$\frac{\mathrm{d}(\boldsymbol{z}_1, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)} \le \frac{e-1}{\lambda e}, \qquad \frac{\mathrm{d}(\boldsymbol{z}_2, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)} \le \frac{1}{\lambda}, \qquad \text{if } \lambda e \le 1 \le e. \tag{2.28}$$

$$\frac{\mathrm{d}(\boldsymbol{z}_1, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)} \le \frac{2(e-1)}{\lambda e(1+\lambda e)}, \qquad \frac{\mathrm{d}(\boldsymbol{z}_2, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)} \le \frac{2}{\lambda(1+\lambda e)}, \qquad \text{if } 1 \le \lambda e. \tag{2.29}$$

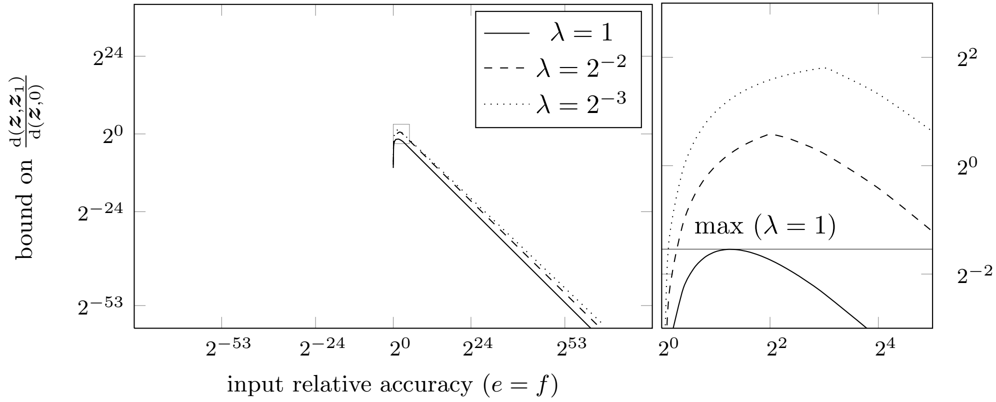

Figure 2.5: Upper Bounds on the Relative Arithmetic Error for $\boldsymbol{z}_1$ Inner Product in Exact Arithmetic (left) with a Close-up around Maximum (right).

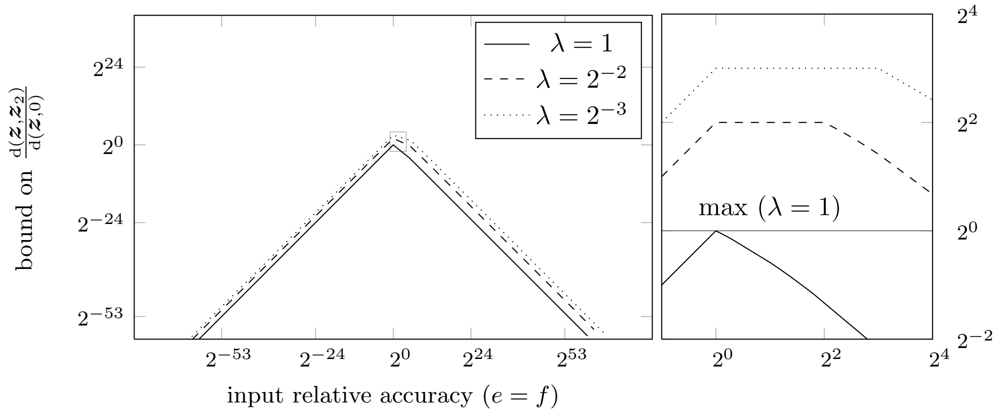

Figure 2.6: Upper Bounds on the Relative Arithmetic Error for $\boldsymbol{z}_2$ Inner Product in Exact Arithmetic (left) with a Close-up around Maximum (right).

The graphs of the corresponding upper bound are displayed in Figures 2.5 and 2.6. The maximum relative accuracy $\mathrm{racc}(\boldsymbol{x}_i) = \mathrm{racc}(\boldsymbol{y}_i) = e$ of all input components is displayed along the abscissa axis in a logarithmic scale. The corresponding bound, namely $\frac{2b}{a}$ for Figure 2.5 and $\frac{2c}{a}$ for Figure 2.6, is reported along the y-axis, also in a logarithmic scale.

The abscissa and ordinate ranges are chosen so that these figures are easily comparable with those in the next chapters. The region around the global maximum that is represented by a small frame is magnified in the right part of each figure. A horizontal line indicates the maximum value reached by the bound when all components are of the same relative accuracy ($\lambda = 1$).

From (2.27–2.29), we note that the bounds vary differently with respect to e in three different areas.

1. In the first area defined by $e \leq 1$, the bound for $\boldsymbol{z}_1$ is zero and the bound for $\boldsymbol{z}_2$ increases with e.
2. In the second area, where $1 \leq e \leq 1/\lambda$, the bound for $\boldsymbol{z}_1$ increases with e, but is bounded by $1/\lambda$, and the bound for $\boldsymbol{z}_2$ is constant and equal to $1/\lambda$.
3. In the third area, where $1/\lambda \leq e$, the bound for $\boldsymbol{z}_2$ decreases when e grows. The situation is more complicated for the bound for $\boldsymbol{z}_1$. If λ is sufficiently close to 1, then this bound first increases with e before decreasing for large e (see the graph for $\lambda = 1$ in Figure 2.5 and see Corollary 2.6 below). Otherwise, it decreases monotonically when e increases.
   Because the ratio of the bound $\frac{e-1}{e}$ is less than 1, the bound on $\boldsymbol{z}_1$ is dominated by the bound on z_2.

So, in all cases, the bound on the relative arithmetic error is worse for $\boldsymbol{z}_2$ than for $\boldsymbol{z}_1$.

Moreover, according to (2.27), the relative arithmetic error for $\boldsymbol{z}_1$ is zero and the relative arithmetic error for $\boldsymbol{z}_2$ tends to zero when e tends to 0. According to (2.29), the relative arithmetic errors for $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ tend to zero when e tends to $+\infty$.

- Finally, we can go one step further and bound from above uniformly, with respect to e and f, the relative arithmetic errors of Corollary 2.6.

Corollary 2.8. *With the hypotheses of Corollary 2.6,*

$$\frac{\mathrm{d}(\boldsymbol{z}_1, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)} \leq \frac{2\sqrt{\lambda^2 + \lambda}}{\lambda(\lambda + \sqrt{\lambda^2 + \lambda})(1 + \lambda + \sqrt{\lambda^2 + \lambda})}. \tag{2.30}$$

Proof. Note that the bounds on $\frac{\mathrm{d}(\boldsymbol{z}_1, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)}$ from (2.28) and (2.29) differ by a factor $2/(1+\lambda e)$, which is greater than 1 if $\lambda e \leq 1$. Consequently, the arithmetic error on $\boldsymbol{z}_1$ is bounded by $g_\lambda(e) = \frac{2(e-1)}{\lambda e(1+\lambda e)}$ when $1 \leq e$, regardless of the value of λe. Its derivative $g'_\lambda(e) = -2(\lambda e^2 - 2\lambda e - 1)/\lambda e^2(1 + \lambda e)^2$ is positive when $0 < e < e_{max} = 1 + \sqrt{1 + 1/\lambda}$ and negative if $e_{max} < e$. The maximum $g_\lambda(e_{max})$ is given by the right-hand side of (2.30). $\square$

Inequality (2.30) applies in all cases because g_λ is a bound on the arithmetic error on $\boldsymbol{z}_1$ when $1 \leq e$, according to the above proof, and the arithmetic error is zero if $e \leq 1$. Nonetheless, it can be improved for small λ. Actually, if $\lambda e_{max} \leq 1$, that is if $\lambda \leq \frac{1}{3}$, then, according to Inequality (2.28), the relative arithmetic error of $\boldsymbol{z}_1$ is bounded by a smaller quantity $\frac{e-1}{\lambda e}$ and the maximum of g_λ is not reached.

Corollary 2.9. *With the hypotheses of Corollary 2.6 and assuming $\lambda \leq \frac{1}{3}$,*

$$\frac{\mathrm{d}(\boldsymbol{z}_1, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)} \leq \frac{1}{\lambda} - 1. \tag{2.31}$$

Proof. When $\lambda e \leq 1$, the relative arithmetic error is bounded by $\frac{e-1}{\lambda e}$, which is monotonically increasing. When $1 \leq \lambda e$, it is bounded by $\frac{2(e-1)}{\lambda e(1+\lambda e)}$, which is monotonically decreasing if $\lambda \leq \frac{1}{3}$, according to the proof of Corollary 2.8. These two quantities cross at $e = 1/\lambda$, where they both equal the maximum value on their range, which is $\frac{1}{\lambda} - 1$. □

The following corollary gives a uniform upper bound on the relative arithmetic error for $\boldsymbol{z}_2$.

Corollary 2.10. *With the hypotheses of Corollary 2.6,*

$$\frac{\mathrm{d}(\boldsymbol{z}_2, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)} \leq \frac{1}{\lambda}. \tag{2.32}$$

Proof. Inequality (2.32) is an immediate consequence of (2.27), (2.28), and (2.29). □

Let us discuss now the extreme cases: $\lambda = 0$ and $\lambda = 1$.

When λ tends to 0, the bound for $\boldsymbol{z}_1$ in (2.31) and the bound for $\boldsymbol{z}_2$ in (2.32) are both asymptotically equivalent to $1/\lambda$.

When $\lambda = 1$, the bounds given by (2.30) and (2.32) are equal to $2(3 - 2\sqrt{2})$ and 1 respectively. These values are compatible respectively with the one given by Nguyen in [Ngu11, Theorem 2.4.3]:

$$\frac{\operatorname{rad} \boldsymbol{z}_1}{\operatorname{rad} \boldsymbol{z}} \leq 4 - 2\sqrt{2}$$

and the one given by Rump [Rum99a, Theorem 2.4]:

$$\frac{\operatorname{rad} \boldsymbol{z}_2}{\operatorname{rad} \boldsymbol{z}} \leq \frac{3}{2}.$$

Indeed, according to the proof of Corollary 2.6, $2b/a$ and $2c/a$ are also upper bounds for $2\left(\frac{\operatorname{rad} \boldsymbol{z}_1}{\operatorname{rad} \boldsymbol{z}} - 1\right)$ and $2\left(\frac{\operatorname{rad} \boldsymbol{z}_2}{\operatorname{rad} \boldsymbol{z}} - 1\right)$. Consequently, the bounds in (2.30) and (2.32), which are derived from Corollary 2.6, verify

$$2\left(\frac{\operatorname{rad} \boldsymbol{z}_1}{\operatorname{rad} \boldsymbol{z}} - 1\right) \leq 2(3 - 2\sqrt{2})$$

and

$$2\left(\frac{\operatorname{rad} \boldsymbol{z}_2}{\operatorname{rad} \boldsymbol{z}} - 1\right) \leq 1.$$

However, these authors use another presentation and their analyses are restricted to the case where $\lambda = 1$ in the documents cited above.

2.5 A new approximate inner product

The notion of relative accuracy introduced for the error analysis can be used to compute directly an overestimated radius. This idea leads to the following approximate inner product, which needs less computation than $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$.

Proposition 2.11. *Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two interval vectors of dimension k. Let e and f such that*

$$e = \max\{\mathrm{racc}(\boldsymbol{x}_i) : 1 \le i \le k\}, \qquad f = \max\{\mathrm{racc}(\boldsymbol{y}_i) : 1 \le i \le k\}.$$

The following formulas compute the approximate inner product $\boldsymbol{z}_3 \supseteq \boldsymbol{x}^T \boldsymbol{y}$

$$\mathrm{mid}\,\boldsymbol{z}_3 = \mathrm{mid}\,\boldsymbol{x}^T \mathrm{mid}\,\boldsymbol{y}, \qquad \mathrm{rad}\,\boldsymbol{z}_3 = (e + f + ef)|\mathrm{mid}\,\boldsymbol{x}|^T |\mathrm{mid}\,\boldsymbol{y}|. \quad (2.33)$$

The approximate inner product $\boldsymbol{z}_3$ always encloses $\boldsymbol{z}_2$, as defined by Proposition 2.3.

Proof. Let us show that $\boldsymbol{z}_3 \supseteq \boldsymbol{z}_2$. From the definition (2.14), we have $\mathrm{mid}\,\boldsymbol{z}_3 = \mathrm{mid}\,\boldsymbol{z}_2$. Using α_i, β_i, γ_i, and δ_i defined in Proposition 2.3, we have $\beta_i \le f|\alpha_i|$, $\gamma_i \le e|\alpha_i|$, and $\delta_i \le ef|\alpha_i|$. So, writing the radius expression as $\mathrm{rad}\,\boldsymbol{z}_3 = \sum_{i=1}^{k}(e|\alpha_i| + f|\alpha_i| + ef|\alpha_i|)$ and using (2.9), we have $0 \le \mathrm{rad}\,\boldsymbol{z}_3 - \mathrm{rad}\,\boldsymbol{z}_2$. This last inequality proves that $\boldsymbol{z}_3 \supseteq \boldsymbol{z}_2$ by (1.2).

Since the approximate inner product $\boldsymbol{z}_2$ contains the exact inner product $\boldsymbol{z}$, we have $\boldsymbol{z}_3 \supseteq \boldsymbol{x}^T \boldsymbol{y}$. □

As in Section 2.3, we study now the radius overestimation of the approximate product $\boldsymbol{z}_3$ in exact arithmetic.

Proposition 2.12. *Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two nonzero k-vectors of interval components. Let e, f, and λ be the three real numbers that verify*

$$\begin{aligned} e &= \max\{\mathrm{racc}(\boldsymbol{x}_i) : 1 \le i \le k\}, \\ f &= \max\{\mathrm{racc}(\boldsymbol{y}_i) : 1 \le i \le k\}, \\ \lambda &= \min\{\mathrm{racc}(\boldsymbol{x}_i)/e, \mathrm{racc}(\boldsymbol{y}_i)/f : 1 \le i \le k\}. \end{aligned}$$

The approximate inner product $\boldsymbol{z}_3 \supseteq \boldsymbol{x}^T \boldsymbol{y}$, defined by (2.33) verifies

$$\mathrm{rad}\,\boldsymbol{z}_3 - \mathrm{rad}\,\boldsymbol{z} \le (c + d)|\mathrm{mid}\,\boldsymbol{x}|^T |\mathrm{mid}\,\boldsymbol{y}| \quad (2.34)$$

where

$$c = \min\{e, f, ef\}, \qquad d = (1 - \lambda)\left(e + f + (1 + \lambda)ef\right).$$

Proof. By definition of α_i, β_i, γ_i, and δ_i given in Proposition 2.3, we have for any $i \le k$,

$$\begin{aligned} f|\alpha_i| - \beta_i &\le (1 - \lambda)f|\alpha_i|, \\ e|\alpha_i| - \gamma_i &\le (1 - \lambda)e|\alpha_i|, \\ ef|\alpha_i| - \delta_i &\le (1 - \lambda^2)ef|\alpha_i|. \end{aligned}$$

Summing the previous inequalities for $i = 1, \ldots, k$ gives (2.34). □

In the special case where all interval components of $\boldsymbol{x}$, on the one hand, and all interval components of $\boldsymbol{y}$, on the other hand, have the same fixed relative accuracy, respectively e for $\boldsymbol{x}$ and f for $\boldsymbol{y}$, we have $\lambda = 1$ and $d = 0$. In that case, Propositions 2.5 and 2.12 show that the arithmetic error of the radius of $\boldsymbol{z}_2$ and $\boldsymbol{z}_3$ can be bounded by the same quantity $\min\{e, f, ef\}|\mathrm{mid}\,\boldsymbol{x}|^T |\mathrm{mid}\,\boldsymbol{y}|$.

The following corollary establishes an upper bound on the relative arithmetic error for the new approximate interval inner product.

Corollary 2.13. *Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two nonzero k-vectors of interval components. Let e, f, and λ the three real numbers that verify*

$$e = \max\{\mathrm{racc}(\boldsymbol{x}_i) : 1 \le i \le k\},$$
$$f = \max\{\mathrm{racc}(\boldsymbol{y}_i) : 1 \le i \le k\},$$
$$\lambda = \min\{\mathrm{racc}(\boldsymbol{x}_i)/e, \mathrm{racc}(\boldsymbol{y}_i)/f : 1 \le i \le k\}.$$

The approximate inner product $\boldsymbol{z}_3 \supseteq \boldsymbol{x}^T\boldsymbol{y}$, defined by (2.33)*, verifies*

$$\frac{\mathrm{d}(\boldsymbol{z}_3, \boldsymbol{z})}{\mathrm{d}(\boldsymbol{z}, 0)} \le 2\,\frac{c+d}{a} \tag{2.35}$$

where

$$a = \lambda \max\{e, f\} + \lambda \max\{\min\{e, f\}, \lambda e f\},$$
$$c = \min\{e, f, ef\}, \qquad d = (1-\lambda)\,(e + f + (1+\lambda)ef).$$

Proof. Inequality (2.35) is a consequence of (1.4), (2.19), and (2.34). □

Since $d \ge 0$, it is clear that the upper bound $\frac{c+d}{a}$ of (2.35) is greater than the upper bound $\frac{c}{a}$ of (2.24) for any $\lambda < 1$ and that they are equal when $\lambda = 1$.

Assume, without loss of generality, that $f \le e$. As for the bounds on the radius overestimation of $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ given in Corollary 2.6, the maximum of $(c+d)/a$ is reached for $f = e$ (see Figure 2.7). Thus in the following, we study the behavior of the bound on the diagonal $e = f$. In that case,

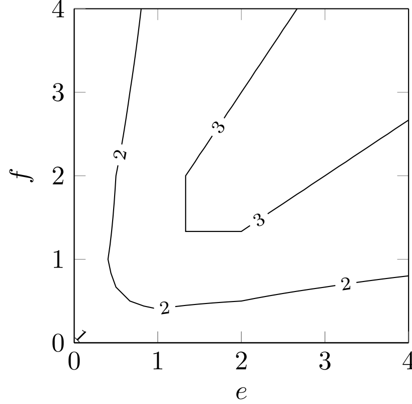

Figure 2.7: Radius Overestimation for z_3 Inner Product ($\lambda = 2^{-1}$).

the situation is noticeably different from those described in Section 2.3: the presence of a factor $1-\lambda$ in the numerator $c+d$ changes the bound trends for small and large e. When $e = f$, the upper bound on the arithmetic error given by Corollary 2.13 is

$$2\,\frac{c+d}{a} = 2\,\frac{\min\{1, e\} + 2(1-\lambda) + (1-\lambda^2)e}{\lambda(1 + \max\{1, \lambda e\})}.$$

When $\lambda = 1$, the previous quantity tends to 0 when e tends to 0 or to $+\infty$, which are the same limits as for $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$.

When $\lambda < 1$, the previous quantity tends to $2\left(\frac{1}{\lambda} - 1\right)$ when e tends to 0, and to $2\left(\frac{1}{\lambda^2} - 1\right)$ when e tends to $+\infty$. Hence, (2.35) does not ensure that the relative arithmetic error is small when e tends to zero or to infinity and $\lambda \ne 1$, in contrast with the cases of (2.23) and (2.24) (see Figure 2.8).

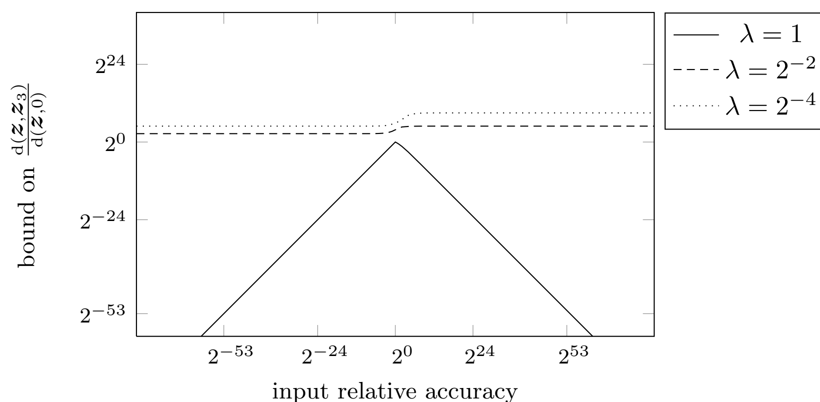

Figure 2.8: Upper Bounds on the Relative Arithmetic Error for z_3 Inner Product in Exact Arithmetic.

2.6 Conclusion

In this chapter, we have seen that it is possible to use approximate arithmetic operators for the definition of approximate inner products while preserving the inclusion property.

It has been shown that the relative arithmetic error of each inner product depends on the bounds on relative accuracies of each input. In particular, we have shown that the worst cases occur when the input vectors have the same bound on their relative accuracies. This remark allows us to focus further analyses on the case where $e = f$. It also depends on the homogeneity of the relative accuracies in the interval components of each input vector. Homogeneous components, that is when λ is close to 1, lead to a smaller bound on the relative arithmetic error.

Besides its utility for the arithmetic error analysis of the inner products, the relative accuracy notion is also the source of inspiration for a new approximate inner product. The different inner products provide different levels of tradeoff between accuracy and ease of computation and serve in turn as building blocks for the interval matrix products of the following chapters.

Chapter 3

Accuracy of interval matrix products

This chapter presents algorithms for the interval matrix product, that are based upon the inner products detailed in the previous chapter. The framework for the computation is now the floating-point arithmetic, we assume that the input matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ below are matrices with interval coefficients that are represented by floating-point numbers.

The main purpose here is to merely bound the roundoff error of the implementation of three algorithms for interval matrix multiplication. The complete analysis and comparison between the algorithms will be conducted in the next chapter.

The first section introduces the floating-point notions that are needed for the error analysis. The next two sections deal with two algorithms for interval matrix multiplication that are proposed by Rump in [Rum12]. We present the underlying ideas and establish a bound on the roundoff error of these algorithms. The same principles that lead to the previous algorithms guided us to the new algorithm introduced and analyzed in Section 3.4. As a conclusion, the last section discusses the conditions and limits of our method for roundoff error analysis.

3.1 Floating-point model

We first introduce some notations specific to the floating-point error analysis (for a more formal description of the corresponding notions see [MBdD$^+$10, § 2.6 Tools for Manipulating Floating-point Errors]). In the following, we assume that all floating-point numbers are written in radix 2 with precision t. The set of floating-point numbers is denoted by $\mathbb{F}$. We assume that the working precision for the computation is the same as for the representation of numbers, so as to avoid double rounding. For simplicity, we use the term *floating-point* as an adjective to indicate that the involved numbers have a floating-point representation. For instance, a *floating-point midpoint-radius interval* designates an interval whose midpoint value and radius value are represented by floating-point numbers. The set of floating-point intervals is denoted by $\mathbb{IF}$. It will be clear from the context if the intervals are in infimum-supremum or in midpoint-radius representation.

For any arithmetic expression E, we note $\mathrm{fl}_{\square}(E)$ the result of an evaluation of E in floating-point arithmetic with all intermediate results rounded to nearest. Likewise, $\mathrm{fl}_{\Delta}(E)$ denotes a floating-point evaluation of E with all intermediate results rounded toward $+\infty$ ($\mathrm{fl}_{\nabla}(E)$ for rounding toward $-\infty$). In any case, we do not assume any particular evaluation order when several orders are compatible with the priority of operations. For arithmetic operations, we

assume that we can employ the *standard model* of floating-point arithmetic as it is defined in [Hig02, § 2.2 Model of arithmetic]: for any finite floating-point numbers $\tilde{x}$ and $\tilde{y}$, we have

$$\mathrm{fl}_{\square}(\tilde{x} * \tilde{y}) = (\tilde{x} * \tilde{y})(1 + \delta)$$

where $*$ is either the addition or the multiplication, $|\delta| \leq \mathtt{u}$, and $\mathtt{u} = 2^{-t}$ is the *unit roundoff* for rounding to nearest. For directed rounding, the unit roundoff is $2\mathtt{u}$.

As we will deal principally with non-negative quantities, we can specialize the standard model to the two following cases. Suppose $0 \leq \tilde{x} + \tilde{y}$, then

$$\mathrm{fl}_{\square}(\tilde{x} + \tilde{y}) \leq (1 + \mathtt{u})(\tilde{x} + \tilde{y}),$$
$$\mathrm{fl}_{\Delta}(\tilde{x} + \tilde{y}) \leq (1 + 2\mathtt{u})(\tilde{x} + \tilde{y}).$$

Suppose $0 \leq \tilde{x}\tilde{y}$, then

$$\mathrm{fl}_{\square}(\tilde{x}\tilde{y}) \leq (1 + \mathtt{u})\tilde{x}\tilde{y},$$
$$\mathrm{fl}_{\Delta}(\tilde{x}\tilde{y}) \leq (1 + 2\mathtt{u})\tilde{x}\tilde{y}.$$

Following [Rum12], we note `realmin` the least positive normal floating-point number. Finally, $\mathrm{ulp}(\tilde{x})$ denotes the *unit in the last place* of a nonzero floating-point number $\tilde{x}$ defined as $\mathrm{ulp}(\tilde{x}) = 2^{\max(e, e_{min}) - t + 1}$ where e is the exponent in the floating-point representation of $\tilde{x}$ and e_{min} is the minimal possible exponent in the set of floating-point numbers. Note that $\mathrm{ulp}(\tilde{x}) \leq \mathtt{u}|\tilde{x}|$.

We will use the following classical result for forward error analysis.

Theorem 3.1 (from [Hig02, § 3.5 Matrix Multiplication])**.** *Let $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$ two matrices. Let $\widetilde{C}$ the computed product AB with the classical triple loop algorithm and all operations performed with rounding to nearest mode. The forward error is bounded as follows*

$$|\widetilde{C} - AB| \leq \gamma_k |A||B| \tag{3.1}$$

where $\gamma_k = k\mathtt{u}/(1 - k\mathtt{u})$.

Note that (3.1) is independent of the three-loop ordering. Furthermore, with the same algorithm but with rounding toward $+\infty$, the computed result $\widetilde{D}$ is an overestimate of the real product, and the forward error is obtained by substituting $2\mathtt{u}$ for $\mathtt{u}$. That is

$$0 \leq \widetilde{D} - AB \leq \gamma_{2k}|A||B|.$$

The following presentation derives directly from the previous inequalities.

Corollary 3.2. *Let $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$ two matrices. Let $\widetilde{C}$ the computed product AB with the classical triple loop algorithm and rounding to nearest mode. Let $\widetilde{D}$ the computed product AB with the classical triple loop algorithm and rounding toward $+\infty$ mode. Then,*

$$|\widetilde{C}| \leq (1 + \gamma_k)|A||B|, \tag{3.2}$$
$$|\widetilde{D}| \leq (1 + \gamma_{2k})|A||B|. \tag{3.3}$$

Next, the following lemma will help us to simplify the error analysis.

Lemma 3.3. *Let n, k, and p three positive integers such that $(n + kp)\mathtt{u} < 1$. Then*

$$(1 + \gamma_n)(1 + p\mathtt{u})^k \leq 1 + \gamma_{n+pk}. \tag{3.4}$$

Proof. Assume first that $k = 1$. From the hypothesis on n, p, and k, we know that $p\mathtt{u} < 1$. Thus, $p\mathtt{u} < \gamma_p$. Then, $(1 + \gamma_n)(1 + p\mathtt{u}) \leq 1 + \gamma_n + \gamma_p + \gamma_n\gamma_p$. Since $\gamma_n + \gamma_p + \gamma_n\gamma_p \leq \gamma_{n+p}$, according to [Hig02, Lemma 3.3], we have $(1 + \gamma_n)(1 + p\mathtt{u}) \leq 1 + \gamma_{n+p}$.

Assume now that $(1 + \gamma_n)(1 + p\mathtt{u})^{k-1} \leq 1 + \gamma_{n+p(k-1)}$. Applying in turn the last two inequalities proves (3.4) for any k. □

3.2 Interval matrix product in three point matrix products

In this section, we present the evolution of the first midpoint-radius algorithm, originally proposed by Rump in 1999, then improved by Rump himself in 2012. Actually, the formulas for the approximate inner products presented in the previous chapter allow one to use directly Level-3 BLAS functions for the most computationally intensive part of the algorithm. As very efficient implementations of the BLAS are widespread, the burden of the performance endeavor, like memory access management or parallelization, is put on the BLAS library programmer. The user simply reuses this work, with profit, for implementing the interval matrix operations.

Thus, this approach leads to an elegant and efficient way to perform interval matrix multiplication, barring some implementation issues, which are addressed in Chapter 5. This algorithm was a great progress compared to the classical matrix product with interval arithmetic (Algorithm 1, page 4). Since 1999, many algorithms for interval matrix multiplication based on the same approach have appeared [OO05, OOO11, Ngu11, OORO12].

We describe first the evolution from the algorithm in four point matrix products to the one in three point matrix products. Then, we bound the roundoff error of the computed radius.

3.2.1 The algorithm: underlying ideas and historical background

From the inner product $\boldsymbol{z}_2$ of interval vectors defined by (2.14), Rump [Rum99a] derived the following formulas for an overestimate $\boldsymbol{C}_3 = \langle M_3, R_3 \rangle$ of the interval matrix product $\boldsymbol{AB}$:

$$M_3 = M_{\boldsymbol{A}} M_{\boldsymbol{B}}, \tag{3.5}$$

$$R_3 = R_{\boldsymbol{A}}(|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}}) + |M_{\boldsymbol{A}}| R_{\boldsymbol{B}}. \tag{3.6}$$

In Equation (3.6), factoring $R_{\boldsymbol{A}}$ saves one costly matrix product. Thus, computing $\boldsymbol{C}_3$ with (3.5) and (3.6) requires only three numerical matrix products in exact arithmetic.

To implement the previous formulas in floating-point arithmetic, we have to take the roundoff errors into account. This can be achieved in two different ways.

The first possibility, which was initially proposed by Rump in 1999, is to compute the supremum endpoint matrix of the product by adding an overestimate of its radius matrix to an overestimate of its midpoint matrix. Symmetrically, the infimum endpoint is computed by subtracting an overestimate of the radius to an underestimate of the midpoint. This can be done easily with directed rounding modes and leads to Algorithm 2.

Algorithm 2 IIMul4

Input: $\boldsymbol{A} = [\underline{A}, \overline{A}], \boldsymbol{B} = [\underline{B}, \overline{B}]$
Output: $\boldsymbol{C} \supseteq \boldsymbol{AB}$
1: $\langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle \leftarrow \mathsf{InfsupToMidrad}(\boldsymbol{A})$
2: $\langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}} \rangle \leftarrow \mathsf{InfsupToMidrad}(\boldsymbol{B})$
3: $R_{\boldsymbol{C}} \leftarrow \mathrm{fl}_{\triangle}(|M_{\boldsymbol{A}}| R_{\boldsymbol{B}} + R_{\boldsymbol{A}}(|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}}))$
4: $\overline{C} \leftarrow \mathrm{fl}_{\triangle}(M_{\boldsymbol{A}} M_{\boldsymbol{B}} + R_{\boldsymbol{C}})$
5: $\underline{C} \leftarrow \mathrm{fl}_{\nabla}(M_{\boldsymbol{A}} M_{\boldsymbol{B}} - R_{\boldsymbol{C}})$
6: **return** $[\underline{C}, \overline{C}]$

The inputs being in infimum-supremum representation, we need to convert them into midpoint-radius representation in order to apply formula (3.6). This is done at lines 1 and 2 of Algorithm 2

with the conversion function `InfsupToMidrad` defined by Algorithm 3.

Algorithm 3 InfsupToMidrad

Input: $\boldsymbol{A} = [\underline{A}, \overline{A}] \in \mathbb{IF}^{m \times k}$
Output: $\langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle \supseteq \boldsymbol{A}$
1: $M_{\boldsymbol{A}} \leftarrow \mathrm{fl}_{\triangle}((\overline{A} + \underline{A})/2)$
2: $R_{\boldsymbol{A}} \leftarrow \mathrm{fl}_{\triangle}(M_{\boldsymbol{A}} - \underline{A})$
3: **return** $\langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle$

The main advantage of Algorithm 2 over Algorithm 1 is that the number of rounding mode changes is limited to 2 and the main part of the computation is expressed as floating-point matrix computations. The cost of Algorithm 2 is dominated by the cost of the four floating-point matrix-matrix products, which can be performed by well-optimized numerical libraries that attain near peak performance.

However, Algorithm 2 requires to compute twice the product $M_{\boldsymbol{A}} M_{\boldsymbol{B}}$, because of the need of one overestimate and one underestimate of this quantity. Rump recently found a means to remove this double computation by using a clever bound on the product error, as presented below.

Indeed, the second possibility to allow for roundoff errors is to accumulate several error bounds in the result radius. The first roundoff error occurs in the computation of $M_{\boldsymbol{A}} M_{\boldsymbol{B}}$. The classical upper bound given by (3.1) is not a floating-point quantity, so Rump stated the following bound, which is defined by floating-point operations.

Theorem 3.4 (from [Rum12, Theorem 2.1])**.** *Let $A \in \mathbb{F}^{m \times k}$ and $B \in \mathbb{F}^{k \times n}$ with $2(k+2)\mathtt{u} \leq 1$ be given, and let $C = \mathrm{fl}_{\square}(AB)$ and $\Gamma = \mathrm{fl}_{\square}(|A||B|)$. Here C may be computed in any order, and we assume that Γ is computed in the same order. Then*

$$|\mathrm{fl}_{\square}(AB) - AB| \leq \mathrm{fl}_{\square}\left(\frac{k+2}{2}\mathrm{ulp}(\Gamma) + \mathtt{realmin}\right). \tag{3.7}$$

Other roundoff errors may appear in the computation of (3.6), but they are automatically taken into account when we evaluate this formula in rounding toward $+\infty$. Of course, $\mathrm{fl}_{\triangle}\left(\frac{k+2}{2}\mathrm{ulp}(\Gamma) + \mathtt{realmin}\right)$ is also a bound on $|\mathrm{fl}_{\square}(AB) - AB|$.

This leads to the algorithm `MMMul3`[1] shown below (Algorithm 4).

Algorithm 4 MMMul3

Input: an m-by-k interval matrix $\boldsymbol{A} = \langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle$,
a k-by-n interval matrix $\boldsymbol{B} = \langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}} \rangle$
Output: $\widetilde{\boldsymbol{C}}_3 \supseteq \boldsymbol{AB}$
1: $\widetilde{M}_3 \leftarrow \mathrm{fl}_{\square}\left(M_{\boldsymbol{A}} M_{\boldsymbol{B}}\right)$
2: $\widetilde{R}_3 \leftarrow \mathrm{fl}_{\triangle}\left(R_{\boldsymbol{A}}(|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}}) + |M_{\boldsymbol{A}}|(R_{\boldsymbol{B}} + \frac{k+2}{2}\mathrm{ulp}|M_{\boldsymbol{B}}|) + \mathtt{realmin}\right)$
3: **return** $\widetilde{\boldsymbol{C}}_3 = \langle \widetilde{M}_3, \widetilde{R}_3 \rangle$

[1] We name the algorithms with the following convention: `II` refers to inputs in infimum-supremum representation and `MM` to midpoint-radius representation, `Mul` refers to matrix multiplication, and the last digit indicates the number of numerical matrix multiplications involved in the algorithm.

Algorithm 4 requires one matrix product less than Algorithm 2. It computes an overestimate of the interval matrix product in no more than the three products needed by formulas (3.5) and (3.6).

Yet, contrary to Algorithm 2, `MMMul3` (Algorithm 4) uses inputs in midpoint-radius representation. In general, such algorithms can be turned into algorithms that accept and return matrices in infimum-supremum representation by converting to and fro this representation, as in Algorithm 5. These conversions add some roundoff errors more and require twice the memory space, but the overhead in execution time is negligible compared to the cost of the matrix products. They are also subject to overflow, but alternative algorithms solve this problem [Gou14]. In the following, we will study only midpoint-radius algorithms like `MMMul3`, setting aside conversions between representations.

Algorithm 5 IIMul3

Input: $\boldsymbol{A} = [\underline{A}, \overline{A}], \boldsymbol{B} = [\underline{B}, \overline{B}]$

Output: $\boldsymbol{C} \supseteq \boldsymbol{AB}$

1: $\langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle \leftarrow$ InfsupToMidrad$(\boldsymbol{A})$

2: $\langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}} \rangle \leftarrow$ InfsupToMidrad$(\boldsymbol{B})$

3: $\langle M_{\boldsymbol{C}}, R_{\boldsymbol{C}} \rangle \leftarrow$ MMMul3$(\langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle, \langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}} \rangle)$

4: $\overline{C} \leftarrow \mathrm{fl}_{\triangle}\left(M_{\boldsymbol{C}} + R_{\boldsymbol{C}}\right)$

5: $\underline{C} \leftarrow \mathrm{fl}_{\nabla}\left(M_{\boldsymbol{C}} - R_{\boldsymbol{C}}\right)$

6: **return** $[\underline{C}, \overline{C}]$

3.2.2 Roundoff error bound

We can now analyze the roundoff errors in Algorithm 4. In this error analysis and in all the following ones, we will assume that no underflow occurs during the computation. However, the algorithms do allow for the occurrence of underflow, preserving the inclusion property even in this exceptional case.

The roundoff error depends on the actual implementation. So, let us first rewrite `MMMul3` in a form where all temporary matrix results are explicit (see Algorithm 6).

The componentwise roundoff error of the computed midpoint matrix is given directly by Theorem 3.4. For the computed radius matrix, we prove the following bound.

Proposition 3.5. *Let $\boldsymbol{A} \in \mathbb{IF}^{m \times k}$ and $\boldsymbol{B} \in \mathbb{IF}^{k \times n}$ with $(4k+12)\mathtt{u} \le 1$.*

The roundoff error of radius $\widetilde{R_3}$ computed by Algorithm 6 verifies

$$\begin{aligned}\widetilde{R_3} - R_3 \le &\gamma_{2k+6}(|M_{\boldsymbol{A}}|R_{\boldsymbol{B}} + R_{\boldsymbol{A}}|M_{\boldsymbol{B}}| + R_{\boldsymbol{A}}R_{\boldsymbol{B}}) + \\ &(2k+4)\mathtt{u}|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + \\ &2\mathtt{realmin}. \end{aligned} \tag{3.8}$$

Proof. Note that the T_i's are non-negative, so they equal their absolute value. We apply (3.2) to line 1, and (3.3) to lines 3 and 5, which gives, respectively,

$$\begin{aligned}|\widetilde{M_3}| &\le (1+\gamma_k)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|,\\ \widetilde{T_2} &\le (1+\gamma_{2k})R_{\boldsymbol{A}}\widetilde{T_1},\\ \widetilde{T_4} &\le (1+\gamma_{2k})|M_{\boldsymbol{A}}|\widetilde{T_3}.\end{aligned}$$

Algorithm 6 MMMul3 with explicit temporaries.

Input: an m-by-k interval matrix $\boldsymbol{A} = \langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle$,
a k-by-n interval matrix $\boldsymbol{B} = \langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}} \rangle$

Output: $\widetilde{\boldsymbol{C}}_3 \supseteq \boldsymbol{AB}$

1: $\widetilde{M_3} \leftarrow \mathrm{fl}_\square \left(M_{\boldsymbol{A}} M_{\boldsymbol{B}}\right)$
2: $\widetilde{T_1} \leftarrow \mathrm{fl}_\triangle \left(|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}}\right)$
3: $\widetilde{T_2} \leftarrow \mathrm{fl}_\triangle \left(R_{\boldsymbol{A}} \widetilde{T_1}\right)$
4: $\widetilde{T_3} \leftarrow \mathrm{fl}_\triangle \left(R_{\boldsymbol{B}} + (k+2)\mathrm{ulp}|M_{\boldsymbol{B}}|\right)$
5: $\widetilde{T_4} \leftarrow \mathrm{fl}_\triangle \left(|M_{\boldsymbol{A}}| \widetilde{T_3}\right)$
6: $\widetilde{T_5} \leftarrow \mathrm{fl}_\triangle \left(\widetilde{T_2} + \widetilde{T_4}\right)$
7: $\widetilde{R_3} \leftarrow \mathrm{fl}_\triangle \left(\widetilde{T_5} + \texttt{realmin}\right)$

8: **return** $\widetilde{\boldsymbol{C}}_3 = \langle \widetilde{M_3}, \widetilde{R_3} \rangle$

For the additions in lines 2, 4, 6, and 7, we have, since $\mathrm{ulp}(M_{\boldsymbol{B}}) \le \mathtt{u}|M_{\boldsymbol{B}}|$:

$$\begin{aligned}
\widetilde{T_1} &\le (1+2\mathtt{u})(|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}}),\\
\widetilde{T_3} &\le (1+2\mathtt{u})(R_{\boldsymbol{B}} + (k+2)\mathtt{u}|M_{\boldsymbol{B}}|),\\
\widetilde{T_5} &\le (1+2\mathtt{u})(\widetilde{T_2} + \widetilde{T_4}),\\
\widetilde{R_3} &\le (1+2\mathtt{u})(\widetilde{T_5} + \texttt{realmin}).
\end{aligned}$$

Starting from the last inequality, substituting in turn $\widetilde{T_i}$ by the corresponding upper bound from the previous inequalities, and using (3.4) with the hypothesis $(2k+6)\mathtt{u} \le 1$, we get

$$\begin{aligned}
\widetilde{R_3} \le &(1+\gamma_{2k+6})(|M_{\boldsymbol{A}}|R_{\boldsymbol{B}} + R_{\boldsymbol{A}}|M_{\boldsymbol{B}}| + R_{\boldsymbol{A}}R_{\boldsymbol{B}}) +\\
&(1+\gamma_{2k+6})(k+2)\mathtt{u}|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| +\\
&(1+2\mathtt{u})\texttt{realmin}.
\end{aligned}$$

If $(2k+6)\mathtt{u} \le 1/2$, then $1+2\mathtt{u} \le 1+\gamma_{2k+6} \le 2$ and the previous inequality implies (3.8). □

The previous proof demonstrates that the roundoff error analysis, which is always neglected in articles about interval matrix multiplication that we are aware of, is indeed tractable. Actually, a small number of technical results, namely the three inequalities of Corollary 3.2 and Lemma 3.3 and the classical inequalities of the standard model of floating-point arithmetic, are sufficient to bound the roundoff error of Algorithm 6.

3.3 Interval matrix product in five point matrix products

We present below another algorithm for interval matrix multiplication. The version first introduced by Nguyen has a cost dominated by seven point matrix products. Rump improved it the same way he did for `MMMul3`, reducing the number of calls to point matrix products to five.

The next section details the steps of the Rump's version, and the following one gives a bound on its roundoff error.

3.3.1 Algorithm

With a reasoning analog to the one leading to (3.5) and (3.6), Nguyen [Ngu11] proposed the use of the approximate inner product $\boldsymbol{z}_1$ defined by (2.13) for a simplified interval matrix product $\boldsymbol{C}_5 = \langle M_5, R_5 \rangle$, such that $\boldsymbol{C}_5 \supseteq \boldsymbol{AB}$. The overestimate $\boldsymbol{C}_5$ is computed as follows

$$M_5 = M_{\boldsymbol{A}} M_{\boldsymbol{B}} + P_{\boldsymbol{A}} P_{\boldsymbol{B}}, \tag{3.9}$$

$$R'_5 = R_{\boldsymbol{A}}(|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}}) + |M_{\boldsymbol{A}}| R_{\boldsymbol{B}} - |P_{\boldsymbol{A}}||P_{\boldsymbol{B}}| \tag{3.10}$$

with

$$P_{\boldsymbol{A}} = \text{sign}(M_{\boldsymbol{A}}) .* \min(|M_{\boldsymbol{A}}|, R_{\boldsymbol{A}}),$$
$$P_{\boldsymbol{B}} = \text{sign}(M_{\boldsymbol{B}}) .* \min(|M_{\boldsymbol{B}}|, R_{\boldsymbol{B}}),$$

where the operator $.*$ denotes, as in MATLAB, the componentwise product of matrices (also known as Hadamard matrix product or Schur product).

In floating-point arithmetic, an approximate midpoint $\widetilde{M_5}$ can be computed with (3.9) and rounding to nearest. Then, provided that $2(2k+2)\mathtt{u} \leq 1$, Rump bounds the roundoff error using Theorem 3.4 as follows (from [Rum12]). The row-block matrix $(M_{\boldsymbol{A}}\, P_{\boldsymbol{A}})$ is chosen for A and the column-block matrix $(M_{\boldsymbol{B}}\, P_{\boldsymbol{B}})^T$ for B. The common dimension of A and B is $2k$, and, by (3.7), the roundoff error of $\widetilde{M_5} = \text{fl}_\square\left(M_{\boldsymbol{A}} M_{\boldsymbol{B}} + P_{\boldsymbol{A}} P_{\boldsymbol{B}}\right)$ is bounded by $\text{fl}_\square\left(\frac{2k+2}{2}\text{ulp}(\widetilde{\Gamma}) + \mathtt{realmin}\right)$, where $\widetilde{\Gamma} = \text{fl}_\square\left(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + |P_{\boldsymbol{A}}||P_{\boldsymbol{B}}|\right)$.

Next, note that the radius of $\boldsymbol{C}_5$ is also given by the following expression:

$$R_5 = (|M_{\boldsymbol{A}}| + R_{\boldsymbol{A}})(|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}}) - \Gamma, \tag{3.11}$$

where $\Gamma = |M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + |P_{\boldsymbol{A}}||P_{\boldsymbol{B}}|$. The formulas (3.10) and (3.11) are equivalent in exact arithmetic. In floating-point arithmetic, (3.11) is subject to a more important cancellation than (3.10). However, the former requires one matrix product less because it reuses Γ, which also appears in the error bound of M_5. So, an overestimate of R'_5 is obtained by evaluating (3.11) in rounding toward $+\infty$.

This leads to the following implementation of `MMMul5` (Algorithm 7), proposed by Rump ([Rum12, Algorithm 4.9]).

Algorithm 7 MMMul5

Input: an m-by-k interval matrix $\boldsymbol{A} = \langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle$,
a k-by-n interval matrix $\boldsymbol{B} = \langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}} \rangle$

Output: $\widetilde{\boldsymbol{C}_5} \supseteq \boldsymbol{AB}$

1: $P_{\boldsymbol{A}} \leftarrow \text{sign}(M_{\boldsymbol{A}}) .* \min(|M_{\boldsymbol{A}}|, R_{\boldsymbol{A}})$

2: $P_{\boldsymbol{B}} \leftarrow \text{sign}(M_{\boldsymbol{B}}) .* \min(|M_{\boldsymbol{B}}|, R_{\boldsymbol{B}})$

3: $\widetilde{M_5} \leftarrow \text{fl}_\square\left(M_{\boldsymbol{A}} M_{\boldsymbol{B}} + P_{\boldsymbol{A}} P_{\boldsymbol{B}}\right)$

4: $\widetilde{\Gamma} \leftarrow \text{fl}_\square\left(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + |P_{\boldsymbol{A}}||P_{\boldsymbol{B}}|\right)$

5: $\widetilde{R_5} \leftarrow \text{fl}_\Delta\left((|M_{\boldsymbol{A}}| + R_{\boldsymbol{A}})(|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}}) - \widetilde{\Gamma} + (2k+2)\text{ulp}(\widetilde{\Gamma}) + 2\mathtt{realmin}\right)$

6: **return** $\widetilde{C_5} = \langle \widetilde{M_5}, \widetilde{R_5} \rangle$

The cost of this algorithm is dominated by the cost of the five matrix products.

The last two terms in the computation of $\widetilde{R_5}$ (line 5) come from the sum of the two following error bounds

$$|\widetilde{M_5} - M_5| \leq (k+1)\mathrm{ulp}(\widetilde{\Gamma}) + \texttt{realmin} \tag{3.12}$$

and

$$|\widetilde{\Gamma} - \Gamma| \leq (k+1)\mathrm{ulp}(\widetilde{\Gamma}) + \texttt{realmin}. \tag{3.13}$$

Inequality (3.12) is proved above and the proof of (3.13) is similar. Indeed, an underestimate of Γ has to be subtracted to $(|M_{\boldsymbol{A}}| + R_{\boldsymbol{A}})(|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}})$ in order to to get an overestimate of R_5 and (3.13) implies that $\widetilde{\Gamma} - (k+1)\mathrm{ulp}(\widetilde{\Gamma}) - \texttt{realmin} \leq \Gamma$.

3.3.2 Roundoff error bound

We rewrite `MMMul5` (Algorithm 7) so as to make explicit the needed temporaries (Algorithm 8).

Algorithm 8 MMMul5 with explicit temporaries.

Input: an m-by-k interval matrix $\boldsymbol{A} = \langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}}\rangle$,
a k-by-n interval matrix $\boldsymbol{B} = \langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}}\rangle$

Output: $\widetilde{\boldsymbol{C}_5} \supseteq \boldsymbol{AB}$

1: $P_{\boldsymbol{A}} \leftarrow \mathrm{sign}(M_{\boldsymbol{A}}).*\min(|M_{\boldsymbol{A}}|, R_{\boldsymbol{A}})$
2: $P_{\boldsymbol{B}} \leftarrow \mathrm{sign}(M_{\boldsymbol{B}}).*\min(|M_{\boldsymbol{B}}|, R_{\boldsymbol{B}})$
3: $\widetilde{T_1} \leftarrow \mathrm{fl}_{\square}(M_{\boldsymbol{A}} M_{\boldsymbol{B}})$; $\widetilde{T_2} \leftarrow \mathrm{fl}_{\square}(P_{\boldsymbol{A}} P_{\boldsymbol{B}})$
4: $\widetilde{M_5} \leftarrow \mathrm{fl}_{\square}\left(\widetilde{T_1} + \widetilde{T_2}\right)$
5: $\widetilde{T_3} \leftarrow \mathrm{fl}_{\square}(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|)$; $\widetilde{T_4} \leftarrow \mathrm{fl}_{\square}(|P_{\boldsymbol{A}}||P_{\boldsymbol{B}}|)$
6: $\widetilde{\Gamma} \leftarrow \mathrm{fl}_{\square}\left(\widetilde{T_3} + \widetilde{T_4}\right)$
7: $\widetilde{T_5} \leftarrow \mathrm{fl}_{\triangle}(|M_{\boldsymbol{A}}| + R_{\boldsymbol{A}})$; $\widetilde{T_6} \leftarrow \mathrm{fl}_{\triangle}(|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}})$
8: $\widetilde{T_7} \leftarrow \mathrm{fl}_{\triangle}\left(\widetilde{T_5}\widetilde{T_6}\right)$
9: $\widetilde{T_8} \leftarrow \mathrm{fl}_{\triangle}\left(2\texttt{realmin} + (2k+2)\mathtt{u}\widetilde{\Gamma}\right)$
10: $\widetilde{T_9} \leftarrow \mathrm{fl}_{\triangle}\left(\widetilde{T_7} + \widetilde{T_8}\right)$
11: $\widetilde{R_5} \leftarrow \mathrm{fl}_{\triangle}\left(\widetilde{T_9} - \widetilde{\Gamma}\right)$

12: **return** $\widetilde{\boldsymbol{C}_5} = \langle \widetilde{M_5}, \widetilde{R_5}\rangle$

A bound on the roundoff error for the computed midpoint is given by (3.12). The next proposition states a bound on the radius roundoff error.

Proposition 3.6. *Let $\boldsymbol{A} \in \mathbb{IF}^{m\times k}$ and $\boldsymbol{B} \in \mathbb{IF}^{k\times n}$ with $4(k+1)\mathtt{u} \leq 1$.*
The roundoff error of radius $\widetilde{R_5}$ computed by Algorithm 8 verifies

$$\begin{aligned}\widetilde{R_5} - R_5 \leq &\gamma_{2k+8}(|M_{\boldsymbol{A}}| + R_{\boldsymbol{A}})(|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}}) + \\ &(4k+2)\mathtt{u}(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + |P_{\boldsymbol{A}}||P_{\boldsymbol{B}}|) + \\ &4\ \texttt{realmin}.\end{aligned} \tag{3.14}$$

Proof. First, we have simple upper bounds for the temporary sums and products $\widetilde{T_3}, \ldots, \widetilde{T_9}$, and $\widetilde{\Gamma}$, which are are non-negative. The computation of $P_{\boldsymbol{A}}$ and $P_{\boldsymbol{B}}$ in lines 1 and 2 yields exact results. Applying (3.2) and (3.3) to matrix products, we have

$$\widetilde{T_3} \le (1+\gamma_k)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|, \qquad \widetilde{T_4} \le (1+\gamma_k)|P_{\boldsymbol{A}}||P_{\boldsymbol{B}}|,$$
$$\widetilde{T_7} \le (1+\gamma_{2k})\widetilde{T_5}\widetilde{T_6}.$$

The temporary sums are bounded as follows:

$$\widetilde{\Gamma} \le (1+\mathtt{u})(\widetilde{T_3}+\widetilde{T_4}),$$
$$\widetilde{T_5} \le (1+2\mathtt{u})(|M_{\boldsymbol{A}}|+R_{\boldsymbol{A}}), \qquad \widetilde{T_6} \le (1+2\mathtt{u})(|M_{\boldsymbol{B}}|+R_{\boldsymbol{B}}),$$
$$\widetilde{T_8} \le (1+2\mathtt{u})(2\mathtt{realmin}+(2k+2)\mathtt{u}\widetilde{\Gamma}), \qquad \widetilde{T_9} \le (1+2\mathtt{u})(\widetilde{T_7}+\widetilde{T_8}).$$

Second, we need an upper bound for the last quantity $\widetilde{R_5}$. Let us show that $\widetilde{R_5}$ is non-negative too. Note that $\widetilde{R_5}$ is computed with rounding toward $+\infty$, so $\widetilde{T_9} - \widetilde{\Gamma} \le \widetilde{R_5}$. Likewise, since the computation of $\widetilde{T_5}$, $\widetilde{T_6}$, $\widetilde{T_7}$, $\widetilde{T_8}$, and $\widetilde{T_9}$ uses rounding toward $+\infty$, we have $(|M_{\boldsymbol{A}}|+R_{\boldsymbol{A}})(|M_{\boldsymbol{B}}|+R_{\boldsymbol{B}}) + (2k+2)\mathtt{u}\widetilde{\Gamma} + 2\mathtt{realmin} \le \widetilde{T_9}$. The last inequality and (3.11) imply

$$R_5 + \Gamma - \widetilde{\Gamma} + (2k+2)\mathtt{u}\widetilde{\Gamma} + 2\mathtt{realmin} \le \widetilde{T_9} - \widetilde{\Gamma}.$$

The left-hand side is non-negative because the exact radius R_5 is non-negative and (3.13) implies $0 \le \Gamma - \widetilde{\Gamma} + (k+1)\mathtt{u}\widetilde{\Gamma} + \mathtt{realmin}$. Consequently, $0 \le \widetilde{R_5}$ and we can write

$$\widetilde{R_5} \le (1+2\mathtt{u})(\widetilde{T_9} - \widetilde{\Gamma}).$$

Third, in order to deal with the subtraction in the previous inequality, we bound $\widetilde{\Gamma}$ from below: (3.13) and $1-(k+1)\mathtt{u} \le 1/(1+(k+1)\mathtt{u})$ imply $(1-(k+1)\mathtt{u})\Gamma - \mathtt{realmin} \le \widetilde{\Gamma}$. Using the last inequality in the bound for $\widetilde{R_5}$, we have

$$\widetilde{R_5} \le (1+2\mathtt{u})\widetilde{T_9} - (1+2\mathtt{u})\,(1-(k+1)\mathtt{u})\,\big(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + |P_{\boldsymbol{A}}||P_{\boldsymbol{B}}|\big) + (1+2\mathtt{u})\mathtt{realmin}.$$

Finally, starting from the last inequality, we substitute in turn the $\widetilde{T_i}$'s and $\widetilde{\Gamma}$ by the corresponding upper bounds. The procedure leads to

$$\widetilde{R_5} \le a(|M_{\boldsymbol{A}}|+R_{\boldsymbol{A}})(|M_{\boldsymbol{B}}|+R_{\boldsymbol{B}}) + b(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + |P_{\boldsymbol{A}}||P_{\boldsymbol{B}}|) + c\ \mathtt{realmin} \tag{3.15}$$

with $a = (1+2\mathtt{u})^4(1+\gamma_{2k})$, $b = 2(1+\mathtt{u})(1+2\mathtt{u})^3(1+\gamma_k)(k+1)\mathtt{u} + (1+2\mathtt{u})(k+1)\mathtt{u} - 2\mathtt{u} - 1$, and $c = 2(1+2\mathtt{u})^3 + (1+2\mathtt{u})$. Let us bound from above these coefficients.

Using (3.4), we have $a \le 1 + \gamma_{2k+8}$.

Again, $(1+\mathtt{u})(1+2\mathtt{u})^3(1+\gamma_k) \le 1 + \gamma_{k+7}$ by (3.4). With the hypothesis $4(k+1)\mathtt{u} \le 1$ and assuming $6 \le t$, as it is the case with usual working precisions, we have $2(1+\gamma_{k+7}) + 1 + 2\mathtt{u} \le 4$ and $1 + 2\mathtt{u} \le (1+\mathtt{u})(1+2\mathtt{u})^3 \le 2$. Then, $b \le (4k+2)\mathtt{u} - 1$ and $c \le 4$.

With the previous upper bounds, (3.11) and (3.15) imply (3.14). □

3.4 A new algorithm in two point matrix products

In the spirit of the previous approximate inner products, it is possible to devise an interval multiplication algorithm with a larger radius overestimation but with a cost of only two point matrix product.

3.4.1 Algorithm

Using the approximate inner product (2.33), we define the following approximate interval matrix product $\boldsymbol{C}_2 = \langle M_2, R_2 \rangle$, such that $\boldsymbol{C}_2 \supseteq \boldsymbol{AB}$:

$$M_2 = M_{\boldsymbol{A}} M_{\boldsymbol{B}}, \tag{3.16}$$

$$R_2 = (e + f + ef)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|. \tag{3.17}$$

The formulas (3.16) and (3.17) lead to the computation of an overestimated interval matrix product (Algorithm 9[2]) in two numerical matrix products.

Algorithm 9 MMMul2

Input: an m-by-k interval matrix $\boldsymbol{A} = \langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle$,
a k-by-n interval matrix $\boldsymbol{B} = \langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}} \rangle$

Output: $\widetilde{\boldsymbol{C}_2} \supseteq \boldsymbol{AB}$

1: $\widetilde{e} \leftarrow \max\{\mathrm{fl}_\Delta(R_{\boldsymbol{A}ij}/|M_{\boldsymbol{A}ij}|) : i = 1, \ldots, m \text{ and } j = 1, \ldots, k\}$
2: $\widetilde{f} \leftarrow \max\{\mathrm{fl}_\Delta(R_{\boldsymbol{B}ij}/|M_{\boldsymbol{B}ij}|) : i = 1, \ldots, k \text{ and } j = 1, \ldots, n\}$
3: $\widetilde{M_2} \leftarrow \mathrm{fl}_\Box\left(M_{\boldsymbol{A}} M_{\boldsymbol{B}}\right)$
4: $\widetilde{\Gamma} \leftarrow \mathrm{fl}_\Box\left(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|\right)$
5: $\widetilde{R_2} \leftarrow \mathrm{fl}_\Delta\left((\widetilde{e} + \widetilde{f} + \widetilde{e}\widetilde{f} + \frac{k+2}{2}\mathtt{u})\widetilde{\Gamma} + \mathtt{realmin}\right)$

6: **return** $\widetilde{\boldsymbol{C}_2} = \langle \widetilde{M_2}, \widetilde{R_2} \rangle$

The cost of `MMMul2` is dominated by the cost of the two matrix products.

The factor $\frac{k+2}{2}\mathtt{u}\widetilde{\Gamma}$ that appears at line 5 comes from the roundoff error bound (3.7) for the product of the midpoints $\widetilde{M_2}$ (line 3).

[2] **ERRATUM**: The computation of an approximate interval matrix product $\widetilde{\boldsymbol{C}_2}$ as in Algorithm 9 is *incorrect*. Actually, the floating-point radius matrix $\widetilde{R_2}$, computed at line 5, is not less than the radius matrix R_2 when $|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| \le \widetilde{\Gamma}$. Yet, since $\widetilde{\Gamma}$ is $|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|$ computed with rounding to nearest (line 4), we have no guaranty that the previous relation holds, and the substitution of $|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|$ by $\widetilde{\Gamma}$ may be incorrect.

The correction is straightforward. A correct $\widetilde{R_2}$ may be computed as

$$\widetilde{R_2} \leftarrow \mathrm{fl}_\Delta\left((\widetilde{e} + \widetilde{f} + \widetilde{e}\widetilde{f})((1 + \frac{k+2}{2}\mathtt{u})\widetilde{\Gamma} + \mathtt{realmin}) + \frac{k+2}{2}\mathtt{u}\widetilde{\Gamma} + \mathtt{realmin}\right)$$

Proof. Using Theorem 3.4 with $A = M_{\boldsymbol{A}}$ and $B = M_{\boldsymbol{B}}$, we have $\left|\Gamma - |M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|\right| \le \mathrm{fl}_\Box\left(\frac{k+2}{2}\mathrm{ulp}(\Gamma) + \mathtt{realmin}\right)$, which implies, by triangular inequality, $|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| \le \Gamma + \mathrm{fl}_\Box\left(\frac{k+2}{2}\mathrm{ulp}(\Gamma) + \mathtt{realmin}\right)$ and the right-hand side can be bounded from above by $\mathrm{fl}_\Delta\left((1 + \frac{k+2}{2}\mathtt{u})\Gamma + \mathtt{realmin}\right)$.

So, when implementing (3.17) in floating-point arithmetic, we can compute an overestimate $\widetilde{R_2}$ that takes into account the roundoff errors of $\widetilde{M_2}$ and $\widetilde{\Gamma}$ with $\widetilde{R_2} \leftarrow \mathrm{fl}_\Delta\left((\widetilde{e} + \widetilde{f} + \widetilde{e}\widetilde{f})\widetilde{\Gamma} + +(\widetilde{e} + \widetilde{f} + \widetilde{e}\widetilde{f} + 1)(\frac{k+2}{2}\mathtt{u}\widetilde{\Gamma} + \mathtt{realmin})\right)$. □

The proof of the roundoff error bound (Section 3.4.2), the global error analysis (Section 4.3), and the implementation of the radius computation (Section 7.3.3) have been performed with the incorrect version (Algorithm 9). However, the small increase in the coefficient of $\widetilde{e} + \widetilde{f} + \widetilde{e}\widetilde{f}$ does not change much the roundoff error and the global error. Similarly, the computation of this coefficient can be performed in such a way that the quantity $\frac{k+2}{2}\mathtt{u}\widetilde{\Gamma} + \mathtt{realmin}$ is reused. So, the overhead of the correction should not change significantly the experimental timings given in Chapter 8.

3.4.2 Roundoff error bound

We assume that no overflow occurs. The midpoint matrix computed by `MMMul2` (Algorithm 9) is identical to $\widetilde{M_3}$, so the roundoff error is the same. The roundoff error on the radius matrix can be bounded as follows.

Proposition 3.7. *Let $\boldsymbol{A} \in \mathbb{IF}^{m\times k}$ and $\boldsymbol{B} \in \mathbb{IF}^{k\times n}$ with $(2k+24)\mathtt{u} \leq 1$.*
The radius $\widetilde{R_2}$ computed by Algorithm 9 verifies

$$\begin{aligned}\widetilde{R_2} - R_2 \leq &\gamma_{k+12}(e+f+ef)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + \\ &(k+2)\mathtt{u}|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + \\ &2\mathtt{realmin}.\end{aligned} \tag{3.18}$$

Proof. The relative accuracies computed at line 1 and 2 verify $\widetilde{e} \leq (1+2\mathtt{u})e$ and $\widetilde{f} \leq (1+2\mathtt{u})f$. Noting $\widetilde{g} = \mathrm{fl}_\Delta(e+f+ef+\frac{k+2}{2}\mathtt{u})$, the previous relations imply $\widetilde{g} \leq (1+2\mathtt{u})^4(e+f+ef+\frac{k+2}{2}\mathtt{u})$. Next, we apply (3.2) to line 4, which gives

$$\widetilde{\Gamma} \leq (1+\gamma_k)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|.$$

Taking into account the last addition with `realmin` and using (3.4), we get

$$\widetilde{R_2} \leq (1+\gamma_{k+12})(e+f+ef)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + (1+\gamma_{k+12})\frac{k+2}{2}\mathtt{u}|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + (1+2\mathtt{u})\mathtt{realmin}.$$

Under the hypothesis $(k+12)\mathtt{u} \leq 1/2$, we have $1+2\mathtt{u} \leq 1+\gamma_{k+12} \leq 2$ and the above inequality implies (3.18). □

3.4.3 Comparison with the Ogita-Oishi's algorithm

In 2005, Ogita and Oishi proposed another algorithm with a cost also dominated by the cost of two matrix products [OO05, Algorithm 6]. Compared to the algorithm proposed above, their method is slightly more computationally expensive and requires more memory accesses, since it needs four additional matrix-vector products. Moreover, the radius overestimation is difficult to bound in general.

3.5 Conclusion

In this chapter, we link the algorithms for interval matrix multiplication to the approximate inner products presented in the previous chapter. This allows us to use the arithmetic error analysis for the inner product as a componentwise error analysis for the matrix products.

The floating-point implementations of interval matrix multiplication entail larger radius results than in real arithmetic. Our method to bound the radius overestimation, due to roundoff error, can be summed up as follows.

1. Decompose the formula for the radius in elementary operations such as scalar multiplications, matrix additions, and matrix products.
2. Bound the temporary results of elementary operations taking their roundoff errors into account. A small list of formulas is sufficient: they are recalled on a single page (Section 3.1).
3. Substitute the temporaries by their bound so as to express the bound on the result as a polynomial function of the inputs.

This approach is subject to the following remarks and limitations.

The first step raises the question of the order that is chosen for the decomposition. Indeed, a complex mathematical formula can be computed in several ways that are mathematically equivalent but that yield different roundoff errors. However, due to the multiplicative nature of the bounds and to the simplicity of the formulas in the algorithms, the bounds given in this chapter allow for any decomposition order.

The second step extensively uses the bounds (3.2) and (3.3) on the roundoff error for a matrix product. These are valid under the hypothesis of computation with the classical three loop algorithm. Note that there is no such restriction for the Rump's bound used in the algorithms.

The formulas that we used to bound the temporary results assume non-negative quantities, which is the case for radii. Nevertheless, substitutions in the third step can be troublesome in the presence of cancellation, as in the computation of $\widetilde{R_5}$ defined by (3.11). This makes the process more delicate and error-prone. Moreover, cancellations worsen the error bound: the difference in the roundoff error bounds for $\widetilde{R_5}$ in (3.14) and for $\widetilde{R_3}$ in (3.8) is up to $4(k+2)\mathtt{u}|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|$. This difference may be important with large values of k or large values of $|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|$.

Chapter 4

Global error analysis

In this chapter, we study the global error of `MMMul2`, `MMMul3`, and `MMMul5`, the three interval matrix product algorithms presented in the previous chapter. The relative Hausdorff error is theoretically bounded from above by the sum of the bound for the arithmetic error that has been established in Chapter 2 and the bound for the roundoff error, established in Chapter 3. We analyze here the behavior of such bounds with respect to several parameters, namely: the matrix dimensions, the input relative accuracies, and the working precision. We also compare the value of the theoretical bound with the actual global error, which we measure on products of random interval matrices.

The first section discusses the issues of measuring the global error. It presents our theoretical and software solutions to these problems. Section 4.2 deals with the analysis of global error for `MMMul3` and it introduces the experimental protocols that are also used for the global error analyses of the other algorithms. Global errors of `MMMul2` and `MMMul5` are analyzed in Section 4.3 and Section 4.4, respectively. The last section concludes by comparing the global errors of the three algorithms and determines the domains where a given algorithm provides the best trade-off between the accuracy of its results and its execution time.

4.1 Measuring the global error experimentally

The difficulty of measuring the actual global error for interval matrix computation lies in that, in general, the exact result is not exactly representable as a matrix of floating-point intervals with the same precision as for the inputs. Nonetheless, we circumvent both problems by using two technical means.

First, as real numbers can be tightly approximated by floating-point numbers with the rounding to nearest operation, real intervals can be best approximated in infimum-supremum representation by optimal outward rounding, defined as in [Neu90, § 1.3 Rounded interval arithmetic].

Definition 4.1. The *optimal outward rounding* $\tilde{\boldsymbol{x}} = [\tilde{a}, \tilde{b}]$ for a real interval $\boldsymbol{x} = [\underline{\boldsymbol{x}}, \overline{\boldsymbol{x}}]$ is the floating-point interval defined by

- $\tilde{a} = \mathrm{fl}_{\nabla}(\underline{\boldsymbol{x}})$,
- $\tilde{b} = \mathrm{fl}_{\Delta}(\overline{\boldsymbol{x}})$.

The midpoint-radius representation is less common than the infimum-supremum representation, so there exists no corresponding notion for the best floating-point approximate. We propose the following one.

Definition 4.2. The *nearest midpoint* interval $\tilde{\boldsymbol{x}}$ of a real interval $\boldsymbol{x}$ is the floating-point interval defined by

- $\operatorname{mid} \tilde{\boldsymbol{x}} = \mathrm{fl}_{\Box}(\operatorname{mid} \boldsymbol{x})$,
- $\operatorname{rad} \tilde{\boldsymbol{x}} = \mathrm{fl}_{\Delta}(\operatorname{rad} \boldsymbol{x} + e)$,

where $e = |\operatorname{mid} \boldsymbol{x} - \operatorname{mid} \tilde{\boldsymbol{x}}|$ is the roundoff error for the midpoint. We note $\tilde{\boldsymbol{x}} = \mathcal{N}(\boldsymbol{x})$.

This notion extends naturally to interval vectors and interval matrices by applying the operator $\mathcal{N}$ to each component. As expected, a real interval is included in its nearest midpoint interval, save exceptional cases. Note that only the situations where $\mathrm{fl}_{\Box}(\operatorname{mid} \boldsymbol{x})$ overflows are troublesome. We will not try to define the meaning of the interval $\langle +\infty, +\infty \rangle$ that results in those cases, and we assume that no overflow occurs in our computations.

Proposition 4.1. *For all $\boldsymbol{x} \in \mathbb{IR}$, $\boldsymbol{x} \subseteq \mathcal{N}(\boldsymbol{x})$ provided that no overflow occurs.*

Proof. We have $|\operatorname{mid} \boldsymbol{x} - \operatorname{mid} \tilde{\boldsymbol{x}}| \leq \operatorname{rad} \tilde{\boldsymbol{x}} - \operatorname{rad} \boldsymbol{x}$ because of the rounding toward $+\infty$ mode used for the computation of $\operatorname{rad} \tilde{\boldsymbol{x}}$. This inequality proves the proposition by (1.2). $\Box$

The nearest midpoint $\mathcal{N}(\boldsymbol{x})$ is one of the tightest floating-point midpoint-radius intervals that include $\boldsymbol{x}$. Such enclosure is not unique, as the exact midpoint may lies exactly half-way of two consecutive floating-point numbers. In that case, two different floating-point intervals contain $\boldsymbol{x}$ and have the minimal possible radius. But the tie rule for the rounding to nearest mode in $\mathrm{fl}_{\Box}(\operatorname{mid} \boldsymbol{x})$ completely determines the nearest midpoint interval $\mathcal{N}(\boldsymbol{x})$.

Second, we choose floating-point matrices in double precision (t = 53) as input matrices in the experiments. The inputs are therefore representable, but, in general, the exact result cannot be computed with double precision arithmetic, because of roundoff errors. And so is the nearest midpoint of the exact result, because of the roundoff error e in Definition 4.2, which should be computed exactly. However, given two floating-point interval vectors in double precision, the nearest midpoint interval of their inner product is computable with *arbitrary precision* arithmetic.

More precisely, the arbitrary precision library MPFR [FHL+07] provides a `mpfr_sum` function that computes the correctly rounded value of a sum. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two $k-$interval vectors. With `mpfr_sum`, a procedure based on equations (2.7) can compute the nearest midpoint rounding $\mathcal{N}(\boldsymbol{z})$ of the exact interval inner product $\boldsymbol{z} = \boldsymbol{x}^T \boldsymbol{y}$. We proceed as follows.

1. The partial products α_i, β_i, γ_i, δ_i, and μ_i are computed exactly in 106-bit precision.
2. The sought midpoint $\operatorname{mid} \tilde{\boldsymbol{x}}$ is obtained with a call to `mpfr_sum`. Its value is the correctly rounded value $\operatorname{mid} \tilde{\boldsymbol{x}} = \mathrm{fl}_{\Box}(\sum \alpha_i + \mu_i)$ of the sum $\sum_{i=1}^{n} [\alpha_i + \mu_i]$.
3. The roundoff error is $e = \sigma(-\operatorname{mid} \tilde{\boldsymbol{x}} + \sum \alpha_i + \mu_i)$, with $\sigma = +1$ if $\operatorname{mid} \tilde{\boldsymbol{x}}$ underestimates the exact midpoint and $\sigma = -1$ otherwise. So, the roundoff error is included in the radius with a second call to `mpfr_sum`:

$$\tilde{r} = \mathrm{fl}_{\Delta} \left(-\sigma \operatorname{mid} \tilde{\boldsymbol{x}} + \sum \beta_i + \gamma_i + \delta_i - |\mu_i| + \sigma \alpha_i + \sigma \mu_i \right).$$

The above procedure can also compute componentwise the nearest midpoint matrix of an interval matrix product. Let us now explain how we will use it in the analysis of the relative forward error. Let $\boldsymbol{A} = \langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle \in \mathbb{IF}^{m \times k}$ and $\boldsymbol{B} = \langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}} \rangle \in \mathbb{IF}^{k \times n}$ be two floating-point interval matrices in midpoint-radius representation. Let consider, for instance, the approximate product $\widetilde{\boldsymbol{C}}_3 \supseteq \boldsymbol{AB}$, which is computed with `MMMul3`. In the global error analysis, we estimate

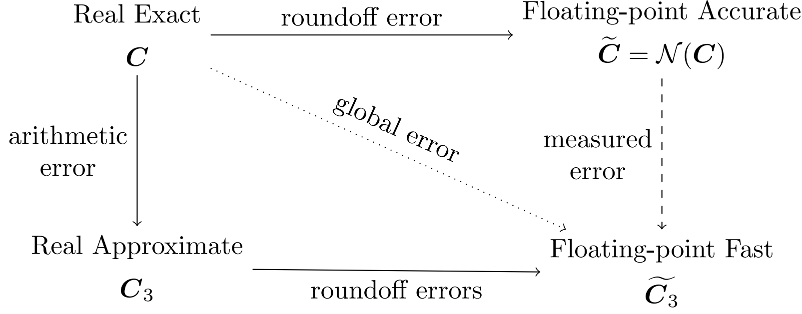

Figure 4.1: Global Error Measurement.

the relative Hausdorff error $\mathrm{d}(\boldsymbol{C}, \widetilde{\boldsymbol{C}_3})/\mathrm{d}(\boldsymbol{C}, 0)$ with the quantity $\mathrm{d}(\mathcal{N}(\boldsymbol{C}), \widetilde{\boldsymbol{C}_3})/\mathrm{d}(\mathcal{N}(\boldsymbol{C}), 0)$. That is, we use the approximate matrix $\mathcal{N}(\boldsymbol{C})$, which is computable, in place of the exact product $\boldsymbol{C}$. The situation is summarized on Figure 4.1.

By construction, $\boldsymbol{x}$ and $\mathcal{N}(\boldsymbol{x})$ are close together and the difference can be quantified as in the next proposition.

Proposition 4.2. *Let $\boldsymbol{x}$ be a real interval. The nearest midpoint rounding $\tilde{\boldsymbol{x}} = \mathcal{N}(\boldsymbol{x})$ of $\boldsymbol{x}$ verifies the following inequality for any interval $\boldsymbol{y}$,*

$$\frac{\mathrm{d}(\tilde{\boldsymbol{x}}, \boldsymbol{y})}{\mathrm{d}(\tilde{\boldsymbol{x}}, 0)} - 3\mathtt{u} \leq \frac{\mathrm{d}(\boldsymbol{x}, \boldsymbol{y})}{\mathrm{d}(\boldsymbol{x}, 0)} \leq (1 + 3\mathtt{u})\frac{\mathrm{d}(\tilde{\boldsymbol{x}}, \boldsymbol{y})}{\mathrm{d}(\tilde{\boldsymbol{x}}, 0)} + 3\mathtt{u}. \tag{4.1}$$

Proof. Using the definition of $\tilde{\boldsymbol{x}}$ and the standard model of floating-point arithmetic (see Section 3.1), we have $|\mathrm{mid}\,\tilde{\boldsymbol{x}} - \mathrm{mid}\,\boldsymbol{x}| \leq \mathtt{u}|\mathrm{mid}\,\boldsymbol{x}|$ and $\mathrm{rad}\,\tilde{\boldsymbol{x}} - \mathrm{rad}\,\boldsymbol{x} \leq 2\mathtt{u}\ \mathrm{rad}\,\boldsymbol{x} + (1 + 2\mathtt{u})\mathtt{u}|\mathrm{mid}\,\boldsymbol{x}|$. The previous inequalities and the definition (1.3) show that the Hausdorff distance between the exact solution $\boldsymbol{x}$ and its floating-point approximation $\tilde{\boldsymbol{x}}$ is minute: $\mathrm{d}(\tilde{\boldsymbol{x}}, \boldsymbol{x}) \leq 3\mathtt{u}\ \mathrm{d}(\boldsymbol{x}, 0)$. The previous bound and a triangular inequality imply $\mathrm{d}(\tilde{\boldsymbol{x}}, \boldsymbol{y}) \leq \mathrm{d}(\boldsymbol{x}, \boldsymbol{y}) + 3\mathtt{u}\ \mathrm{d}(\boldsymbol{x}, 0)$ for any interval $\boldsymbol{y}$. Since $\mathrm{d}(\boldsymbol{x}, 0) \leq \mathrm{d}(\tilde{\boldsymbol{x}}, 0)$, we have the left-hand side inequality in (4.1).

Likewise, we have $|\mathrm{mid}\,\tilde{\boldsymbol{x}}| \leq (1+\mathtt{u})|\mathrm{mid}\,\boldsymbol{x}|$ and $\mathrm{rad}\,\tilde{\boldsymbol{x}} \leq (1+2\mathtt{u})\mathrm{rad}\,\boldsymbol{x} + (1+2\mathtt{u})\mathtt{u}|\mathrm{mid}\,\boldsymbol{x}|$. Since $1 + 2\mathtt{u} + 2\mathtt{u}^2 \leq 1 + 3\mathtt{u}$, the three last inequalities imply $\mathrm{d}(\tilde{\boldsymbol{x}}, 0) \leq (1 + 3\mathtt{u})\mathrm{d}(\boldsymbol{x}, 0)$. Moreover, the bound on $\mathrm{d}(\tilde{\boldsymbol{x}}, \boldsymbol{x})$ and a triangular inequality show that $\mathrm{d}(\boldsymbol{x}, \boldsymbol{y}) \leq \mathrm{d}(\tilde{\boldsymbol{x}}, \boldsymbol{y}) + 3\mathtt{u}\ \mathrm{d}(\boldsymbol{x}, 0)$ for any interval $\boldsymbol{y}$. This inequality and the bound on $\mathrm{d}(\tilde{\boldsymbol{x}}, 0)$ prove the upper bound. ☐

The previous proposition implies that the measured relative error $\mathrm{d}(\mathcal{N}(\boldsymbol{C}), \widetilde{\boldsymbol{C}_3})/\mathrm{d}(\mathcal{N}(\boldsymbol{C}), 0)$ is a good estimate of the relative Hausdorff error $\mathrm{d}(\boldsymbol{C}, \widetilde{\boldsymbol{C}_3})/\mathrm{d}(\boldsymbol{C}, 0)$, unless it is of the order of magnitude of the unit roundoff $\mathtt{u}$. All numerical results that are displayed in figures and tables below are computed with double precision ($\mathtt{u} = 2^{-53}$).

4.2 Global error for `MMMul3`

4.2.1 Method of global error analysis

In the present section and in the next two ones, we conduct the global error analysis by following the same three steps.

- First, a componentwise bound on the radius error is established. This allows us to use the bounds on the arithmetic error that were established in Chapter 2 for the approximate

dot products. We add them to the bounds on the roundoff error from Chapter 3 and we express the sum as a function of the midpoints and relative accuracies of the inputs.

Then, we form a bound on the relative error for the radius for the special case of inputs with the same fixed relative accuracy. As noted in Section 2.4, this is a worst case for the arithmetic error. We then describe how the bound varies when we change the matrix dimensions, the input relative accuracies or the working precision.

- Second, according (1.4), the relative Hausdorff error, which we ultimately want to analyze, is bounded by twice the value of the bound on the relative error for the radius. On a set of random matrices with fixed relative accuracy, we examine the discrepancy between this bound and the measured relative Hausdorff error $\mathrm{d}(\mathcal{N}(\boldsymbol{C}), \widetilde{\boldsymbol{C}}_3)/\mathrm{d}(\mathcal{N}(\boldsymbol{C}), 0)$.

- And finally, we examine the discrepancy between the previous bound and the measured relative Hausdorff error on a set of random matrices whose relatives accuracies vary and are bounded by a given value.

Let us focus now on the global error analysis for `MMMul3` (Algorithm 6).

4.2.2 Upper bounds for the radius global error

The next proposition establishes a bound on the global error $\widetilde{R_3} - R$.

Proposition 4.3. *Let $\boldsymbol{A} \in \mathbb{IF}^{m\times k}$ and $\boldsymbol{B} \in \mathbb{IF}^{k\times n}$ two interval floating-point matrices with $(4k+12)\mathtt{u} \le 1$. Let e and f such that*

$$e = \max\{\mathrm{racc}(\boldsymbol{A}_{ij}) : 1 \le i \le m, 1 \le j \le k\},$$
$$f = \max\{\mathrm{racc}(\boldsymbol{B}_{ij}) : 1 \le i \le k, 1 \le j \le n\}.$$

The global error on the radius computed by Algorithm 6 is bounded as follows

$$\widetilde{R_3} - R \le \left(\min\{e, f, ef\} + \gamma_{2k+6}(e+f+ef) + (2k+4)\mathtt{u}\right)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + 2\,\mathtt{realmin}. \tag{4.2}$$

Proof. Proposition 2.5 gives a bound on the arithmetic error: $R_3 - R \le \min\{e, f, ef\}|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|$. By definition of the relative accuracies, $R_{\boldsymbol{A}} \le e|M_{\boldsymbol{A}}|$ and $R_{\boldsymbol{B}} \le f|M_{\boldsymbol{B}}|$. So, (3.8) can be rewritten as $\widetilde{R_3} - R_3 \le \left(\gamma_{2k+6}(e+f+ef) + (2k+4)\mathtt{u}\right)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + 2\,\mathtt{realmin}$. Summing the bounds on the arithmetic error and the roundoff error implies (4.2). □

The following proposition gives an upper bound on the relative radius error in the special case $e = f$.

Proposition 4.4. *Let $\boldsymbol{A} \in \mathbb{IF}^{m\times k}$ and $\boldsymbol{B} \in \mathbb{IF}^{k\times n}$ two interval floating-point matrices in midpoint-radius representation with $(4k+12)\mathtt{u} \le 1$. Let e and λ two positive real numbers such that*

$$e = \max_{\substack{1\le i\le m\\ 1\le l\le k}}\{\mathrm{racc}(\boldsymbol{A}_{il})\} = \max_{\substack{1\le l\le k\\ 1\le j\le n}}\{\mathrm{racc}(\boldsymbol{B}_{lj})\},$$
$$\lambda = \max\{C > 0 : Ce|\mathrm{mid}\,\boldsymbol{A}| \le \mathrm{rad}\,\boldsymbol{A} \text{ and } Ce|\mathrm{mid}\,\boldsymbol{B}| \le \mathrm{rad}\,\boldsymbol{B}\}.$$

The relative radius error of the approximate product $\widetilde{C_3}$ computed by Algorithm 6 is bounded from above as follows:

$$\frac{\widetilde{R_3} - R}{R} \leq \frac{1}{\lambda}\left\{b + c + d + \epsilon(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|)\right\} \tag{4.3}$$

where

$$b = \frac{(2k+4)\mathtt{u}}{e\max\{2, 1+\lambda e\}}, \qquad d = \gamma_{2k+6}\frac{2+e}{\max\{2, 1+\lambda e\}},$$

$$c = \frac{\min\{1, e\}}{\max\{2, 1+\lambda e\}}, \qquad \epsilon(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|) = \frac{2\mathtt{realmin}}{e\max\{2, 1+\lambda e\}|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|}.$$

Proof. We know from (2.19) that $R \geq (\lambda(\max\{e, f\} + \max\{\min\{e, f\}, \lambda ef\})\,|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|$. In the special case $e = f$, we have $R \geq \lambda e\max\{2, 1+\lambda e\}|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|$. Moreover, $\min\{e, f, ef\} = e\min\{1, e\}$. Then, the numerator of the relative radius error can be bounded using (4.2) and a rewriting gives (4.3). ☐

It is instructive to study the variation of the coefficients with respect to the relative accuracy e and to notice their origin.

- The coefficient b comes from the roundoff error of the computation of the midpoint $M_{\boldsymbol{A}}M_{\boldsymbol{B}}$. It is monotonically decreasing, and it dominates the other coefficients when e is small.
- The coefficient c is related merely to the arithmetic error. Its behavior has been studied in Section 2.3: it is negligible for both small and large relative accuracies.
- The coefficient d corresponds to the relative roundoff error for the radius computation. As $1 \leq \frac{2+e}{\max\{2,1+\lambda e\}} \leq 3$, it does not vary much and is of the order of magnitude of $(2k+6)\mathtt{u}$. It is therefore negligible unless the input relative accuracy is extremely large.
- The last coefficient $\epsilon(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|)$ originates from the value that is added to the radius so as to guard against underflow in midpoint computation. It is the only one that depends on the value of the product $|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|$. It is very small because of the factor `realmin`, which represents the smallest normalized floating-point number, and does not intervene unless $|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|$ is close to 0.

Figure 4.2 illustrates the relative behavior of a, b and c as e varies.

Let $e_0 < 1$ and $e_1 > 1$ be the relative accuracy values defined by $c(e_0) = b(e_0)$ and $c(e_1) = d(e_1)$. We have $e_0 = \sqrt{(2k+4)\mathtt{u}}$ and, in the case $\lambda = 1$, $e_1 = \frac{1}{(2k+6)\mathtt{u}} - 3$. To sum up the previous remarks, the roundoff error on $\widetilde{M_3}$ dominates when the relative accuracy is less than $e_0 \approx \sqrt{2k\mathtt{u}}$. From e_0 to $e_1 \approx \frac{1}{2k\mathtt{u}}$, the bound is dominated by the arithmetic error. Next, when $e_1 \leq e$, the radius roundoff error bound prevails. With respect to the common dimension of the input matrices, b is linear in k, c does not depends on k, and d is almost linear in k for reasonable matrix dimensions since $\gamma_n = n\mathtt{u}/(1 - n\mathtt{u}) \approx n\mathtt{u}$ when $nu \ll 1$. So the intersection point e_0 increases and e_1 decreases as k grows.

4.2.3 Comparison with the measured relative Hausdorff error: fixed relative accuracies

We now compare the behavior of the relative radius error studied above with the actual relative Hausdorff error for two synthetic random sets. The experimental protocol will be the same for all numerical experiments reported here: for a given relative accuracy,

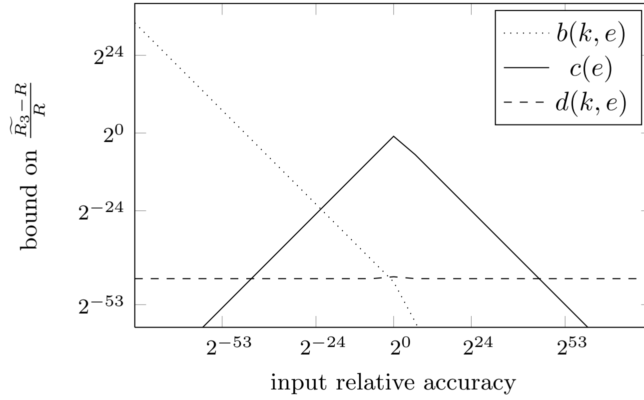

Figure 4.2: Decomposition of the Bound on the Relative Radius Error for `MMMul3` ($k = 128, \lambda = 1$).

- 100 pairs of random matrices are generated as described below;
- as we are doing a componentwise analysis, each component of the 100 approximate products is classified with respect to its relative Hausdorff error;
- we report the proportion of components in the result that have a given relative Hausdorff error.

The first random dataset, thereafter denoted *dataset I*, is defined as follows. For a given relative accuracy e, the midpoint m of each component of $\boldsymbol{A}$ or $\boldsymbol{B}$ is chosen following the standard normal distribution $\mathcal{N}(\overline{m} = 0, \sigma = 1)$; the midpoint being chosen, the component radius is set to $e|m|$. The generated intervals are distributed as on Figure 4.3.

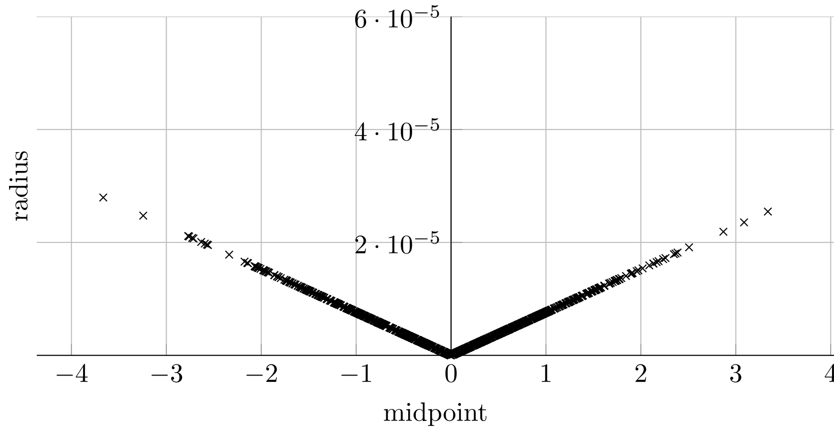

Figure 4.3: Random Dataset I – normal midpoints, fixed relative accuracy ($e = 2^{-17}$, $\lambda = 1$).

The results for the first dataset are displayed in Figure 4.4. The x-axis represents the common relative accuracy of the random factors $\boldsymbol{A}$ and $\boldsymbol{B}$, the y-axis indicates the value of the relative Hausdorff error $\mathrm{d}(\widetilde{\boldsymbol{C}}, \widetilde{\boldsymbol{C}_3})/\mathrm{d}(\widetilde{\boldsymbol{C}}, 0)$, where $\widetilde{\boldsymbol{C}}$ is the nearest-midpoint rounding of the exact product, as explained above. Both axes are in logarithmic scale. The distribution of relative Hausdorff

error for each component of the product $\widetilde{\boldsymbol{C}_3}$ is printed in gray scale: for a given relative accuracy and relative Hausdorff error, the darker the plot, the higher the proportion of components with this relative Hausdorff error in the computed product.

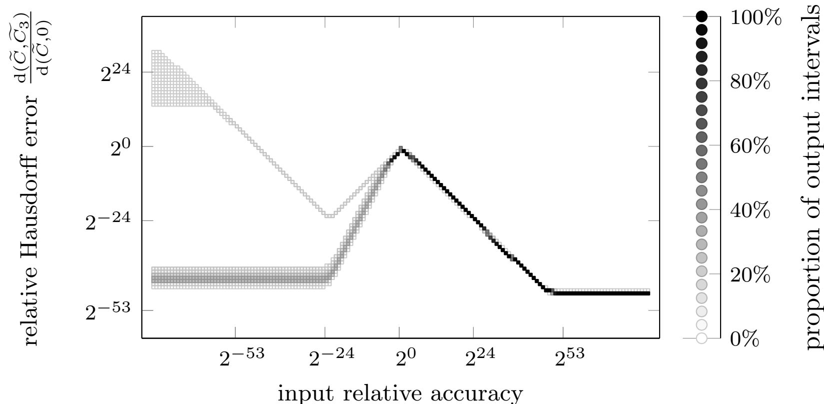

Figure 4.4: Measured Relative Hausdorff Error for MMMul3 with Random Dataset I (matrices of size 128×128).

An upper bound on the relative *radius error* is established in Proposition 4.4. We measure here the relative *Hausdorff error*. Both errors are linked by inequality (1.4): $d(C,\widetilde{C_3})/d(C,0) \leq 2(\widetilde{R_3}-R)/R$. It is apparent on Figure 4.4 that the maximum for the measured relative Hausdorff error behaves like the theoretical bound when the relative accuracies of input vary. The values reported in Table 4.1 confirm that the relative Hausdorff error (second row) is actually bounded from above by twice the bound on the relative radius error (last row, where the value of the bound (4.3) is computed for $\lambda = 1$).

Input relative accuracy	2^{-60}	2^{-53}	2^{-24}	2^{0}	2^{24}	2^{53}
max. relative Hausdorff error						
Dataset I	2^{13}	2^{6}	2^{-23}	2^{-1}	2^{-24}	2^{-47}
Dataset II	2^{14}	2^{7}	2^{-23}	2^{-2}	2^{-23}	2^{-47}
$2\times$ Bound (4.3) (with $\lambda = 1$)	2^{16}	2^{9}	2^{-20}	2^{0}	2^{-22}	2^{-43}

Table 4.1: Comparison between the Maximum Measured Relative Hausdorff Error and the Bound for MMMul3 ($k = 128$).

Concerning the working precision, we see from Table 4.1 that the radius of the computed product may be up to 65 times larger than the exact radius when the inputs in double precision ($\mathtt{u} = 2^{-53}$) are accurate within one unit in the last place ($e = 2^{-53}$). Likewise, the factor of the relative error for the computed product radius is less than 2^{-23} when the inputs in single precision ($\mathtt{u} = 2^{-24}$) are accurate within one unit in the last place ($e = 2^{-24}$) while the computation is performed in double precision arithmetic. The latter factor would equal 2^{-14} with the same precision of inputs, but with calculation in single precision arithmetic.

Moreover, two different patterns are visible on Figure 4.4. On the one hand, for large relative accuracies $e > 1$, the relative Hausdorff error of the computed product closely follows the bound of the radius error. On the other hand, for small input relative accuracy $e < 1$, the relative Hausdorff error distribution is divided in two branches, in the left-hand side of Figure 4.4. For

some instances, the product error is as large as the bound (upper branch), while for a great majority, it is accurate within a few units in the last place (lower branch). Let us show that the situation of a particular component of the result depends on the condition number of its computation. Remember first that we are analyzing the global error componentwise. So, we consider here the simplified case of an inner product between the 1-by-k matrix $\boldsymbol{A}$ with the k-by-1 matrix $\boldsymbol{B}$, without loss of generality. By definition, the relative radius error for $\widetilde{\boldsymbol{C}_3}$ is $\mathrm{rre} = \frac{\widetilde{R_3}-R}{R}$ and the relative Hausdorff error is $\mathrm{rHe} = \frac{\mathrm{d}(\widetilde{\boldsymbol{C}_3},\boldsymbol{C})}{\mathrm{d}(\boldsymbol{C},0)}$. Their ratio is therefore equal to

$$\frac{\mathrm{rre}}{\mathrm{rHe}} = \frac{\widetilde{R_3} - R}{\mathrm{d}(\widetilde{\boldsymbol{C}_3}, \boldsymbol{C})} \times \frac{\mathrm{d}(\boldsymbol{C}, 0)}{R}. \tag{4.4}$$

The first factor is easy to bound: since $\widetilde{\boldsymbol{C}_3} \supseteq \boldsymbol{C}$ we have $\frac{1}{2} \leq \frac{\widetilde{R_3}-R}{\mathrm{d}(\widetilde{\boldsymbol{C}_3},\boldsymbol{C})} \leq 1$ by (1.2). Let us now define the quantities α_i, β_i, γ_i, δ_i, and μ_i as in Proposition 2.3, page 28. Note that in our case, we have $e < 1$ and $\beta_i = \delta_i = e|\alpha_i|$, and $|\mu_i| = \delta_i = e^2|\alpha_i|$. Then, the second factor in (4.4) depends on the condition number for the inner product, defined as $\kappa = \frac{\sum|\alpha_i|}{|\sum\alpha_i|}$. Indeed, we have $R = 2e\sum|\alpha_i|$ and $\mathrm{d}(\boldsymbol{C}, 0) = (1 + e^2)|\sum\alpha_i| + 2e\sum|\alpha_i|$. Thus,

$$\frac{\mathrm{d}(\boldsymbol{C}, 0)}{R} = 1 + \frac{1 + e^2}{2e}\frac{1}{\kappa}. \tag{4.5}$$

If the condition number is large, say when $\frac{1+e^2}{2e} \ll \kappa$, then the ratio in (4.5) is close to one, and the ratio (4.4) is between $\frac{1}{2}$ and 1. In that case the upper bound on relative radius error is an accurate upper bound for the relative Hausdorff error. If the condition number is moderately small, say $\kappa \ll \frac{1}{2e}$, then the second summand in the right-hand side sum of (4.5) dominates. In that case the relative radius error is several order of magnitude larger than the relative Hausdorff error, around $\frac{1}{2e}$ times larger. In summary, the bound on the relative radius error is accurate for huge condition numbers, and we are in the upper branch of Figure 4.4. Conversely, the bound on the relative radius error greatly overestimates the bound on the relative Hausdorff error if the condition number is moderate, which is likely to be the case with normally distributed midpoints.

The error distribution observed for input relative accuracy $e = 2^{-53}$ is reported in Table 4.2. These observations confirm that the lower branch on Figure 4.4 is much more representative of the relative Hausdorff error behavior than the upper branch.

relative Hausdorff error	$[2^{-46}, 2^{-40})$	$[2^{6}, 2^{7})$
proportion of components	93%	7%

Table 4.2: Repartition of the Measured Relative Hausdorff Error of `MMMul3` with Dataset I, $e = 2^{-53}$ and $k = 128$

4.2.4 Comparison with the measured relative Hausdorff error: bounded relative accuracies

The second random dataset, denoted *dataset II*, tries out the relevance of the bound (4.3) when some interval components of the input factors are thin ($\lambda = 0$). For a given relative accuracy e, the midpoints are normally distributed, as for the previous dataset, but the radius corresponding to the midpoint m is chosen uniformly at random in $[0, e|m|]$. Figure 4.5 illustrates the distribution of the generated intervals.

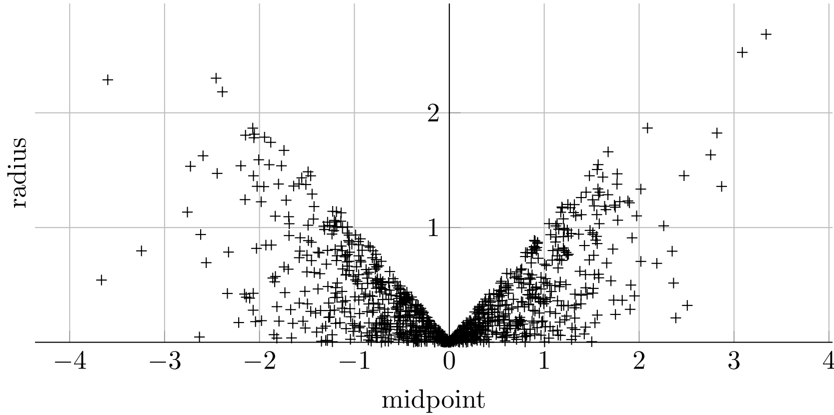

Figure 4.5: Random Dataset II – normal midpoints, bounded relative precision ($e = 1$ and $\lambda = 0$).

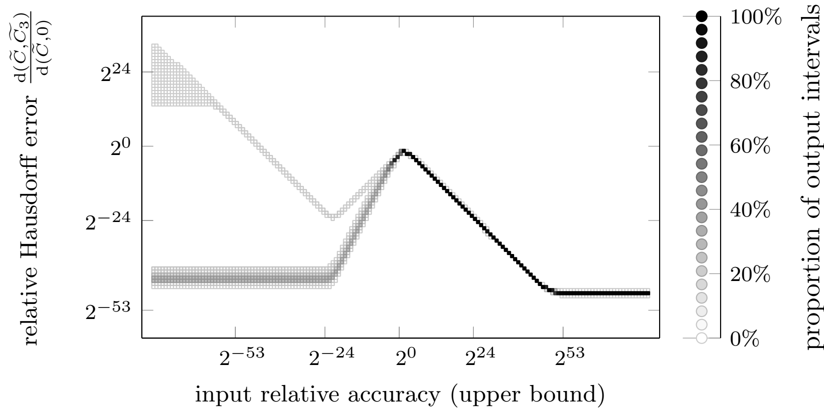

Figure 4.6: Relative Hausdorff Error for `MMMul3` with Random Dataset II (matrices of size 128×128).

The relative Hausdorff error for the approximate product $\widetilde{C_3}$ is presented on Figure 4.6 in a display analog to Figure 4.4.

The result values are quite similar to the fixed relative accuracy case, and the third row in Table 4.1 confirms this observation. This behavior can be explained by the fact that quantities computed from small radii are small and therefore they are absorbed when added to their counterparts derived from larger radii. So the overall behavior is dictated by intervals with large radii. The midpoints being chosen according to a standard normal distribution, they tend to have comparable orders of magnitude, and consequently, interval with large radius are likely to be intervals with large relative accuracy.

Note that the bound given in (4.3) is not finite when $\lambda = 0$. However, the measured relative Hausdorff error seems to be as good as in the case $\lambda = 1$, as can be seen when comparing Figures 4.4 and 4.6 and the second and third rows in Table 4.1. Further investigation would be needed to determine if this is an artifact due to the random distribution. The fact that we do not observe very large relative relative Hausdorff error may be due to the fact that the chosen

random distribution rarely generates catastrophic cancellations in the midpoint computation. Nonetheless, two observations indicate that the λ^{-1} factor in the bound (4.3) grossly overestimates the actual maximum relative Hausdorff error. First, we noted in Section 2.3 that the lower bound $(\lambda \max\{e, f\} + \max\{\min\{e, f\}, \lambda e f\}) \, |\mathrm{mid}\, \boldsymbol{x}|^T |\mathrm{mid}\, \boldsymbol{y}|$ on $\mathrm{rad}\, \boldsymbol{z}$ in (2.19) is not tight, and this is from where the λ^{-1} factor in the bound (4.3) originates. In addition, the relative Hausdorff errors that we measured in our numerical experiments do not differ much between the two cases $\lambda = 1$ and $\lambda = 0$. Therefore, it is not unrealistic to hope for a bound on the relative error for `MMMul3` that depends on e, k, and $\mathtt{u}$, but no more on λ.

In addition, when $e = 2^{-53}$, the repartition of the measured relative Hausdorff error is almost the same as for dataset I (compare Tables 4.2 and 4.3). As a result, the conclusion is identical: the dominant behavior is represented by the lower branch when $e < 1$.

relative Hausdorff error	$[2^{-46}, 2^{-40})$	$[2^{6}, 2^{8})$
proportion of components	93%	7%

Table 4.3: Repartition of the Measured Relative Hausdorff Error of `MMMul3` with Dataset II, $e = 2^{-53}$ and $k = 128$

4.3 Global error for `MMMul2`

We perform here for `MMMul2` (Algorithm 9[1]) the analysis of the Hausdorff error with the protocol presented in the previous section. The global error $\widetilde{R_2} - R$ may be bounded as follows.

Proposition 4.5. *Let $\boldsymbol{A} \in \mathbb{IF}^{m\times k}$ and $\boldsymbol{B} \in \mathbb{IF}^{k\times n}$ two interval floating-point matrices with $(2k+24)\mathtt{u} \leq 1$. Let e, f, and λ three positive real numbers such that*

$$\begin{aligned}
e &= \max\{\mathrm{racc}(\boldsymbol{A}_{ij}) : 1 \leq i \leq m, 1 \leq j \leq k\},\\
f &= \max\{\mathrm{racc}(\boldsymbol{B}_{ij}) : 1 \leq i \leq k, 1 \leq j \leq n\},\\
\lambda &= \max\{C > 0 : Ce|\mathrm{mid}\, \boldsymbol{A}| \leq \mathrm{rad}\, \boldsymbol{A} \text{ and } Ce|\mathrm{mid}\, \boldsymbol{B}| \leq \mathrm{rad}\, \boldsymbol{B}\}.
\end{aligned}$$

The global error on the radius computed by Algorithm 9 is bounded as follows

$$\begin{aligned}
\widetilde{R_2} - R \leq &(k+2)\mathtt{u}|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + \\
&(\min\{e, f, ef\} + (1-\lambda)(e + f + (1+\lambda)ef)) \, |M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + \\
&\gamma_{k+12}(e + f + ef)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + \\
&2\,\mathtt{realmin}, \qquad (4.6)
\end{aligned}$$

Proof. The bound in (4.6) is a combination of (2.34) and (3.18). □

From the previous bound, we can derive a bound on the relative radius error. We will focus on the case $e = f$.

Proposition 4.6. *Let $\boldsymbol{A} \in \mathbb{IF}^{m\times k}$ and $\boldsymbol{B} \in \mathbb{IF}^{k\times n}$ two interval floating-point matrices with $(2k+24)\mathtt{u} < 1$. Let e and λ two positive real numbers such that*

$$\begin{aligned}
e &= \max_{\substack{1\leq i\leq m\\ 1\leq l\leq k}} \{\mathrm{racc}(\boldsymbol{A}_{il})\} = \max_{\substack{1\leq l\leq k\\ 1\leq j\leq n}} \{\mathrm{racc}(\boldsymbol{B}_{lj})\},\\
\lambda &= \max\{C > 0 : Ce|\mathrm{mid}\, \boldsymbol{A}| \leq \mathrm{rad}\, \boldsymbol{A} \text{ and } Ce|\mathrm{mid}\, \boldsymbol{B}| \leq \mathrm{rad}\, \boldsymbol{B}\}.
\end{aligned}$$

[1]See the erratum in the footnote page 48.

The relative radius error of the approximate product $\widetilde{C_2}$ computed by Algorithm 9 is bounded from above as follows:

$$\frac{\widetilde{R_2} - R}{R} \leq \frac{1}{\lambda}\left\{b + c + d + \epsilon(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|)\right\} \tag{4.7}$$

where

$$b = \frac{(k+2)\mathtt{u}}{e\max\{2, 1+\lambda e\}}, \qquad d = \gamma_{k+12}\frac{2+e}{\max\{2, 1+\lambda e\}},$$

$$c = \frac{\min\{1, e\}}{\max\{2, 1+\lambda e\}} + (1-\lambda)\frac{2+(1+\lambda)e}{\max\{2, 1+\lambda e\}}, \qquad \epsilon(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|) = \frac{2\,\mathtt{realmin}}{e\max\{2, 1+\lambda e\}|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|}.$$

Proof. (4.7) is a consequence of (2.19) and (4.6) when $e = f$. □

The same observations as in the `MMMul3` case can be made on the coefficients b, c, and d. The contributions of b, c, and d are shown on Figure 4.7. The value of λ does not change much the values of b and d, so we display b and d for $\lambda = 1$ only.

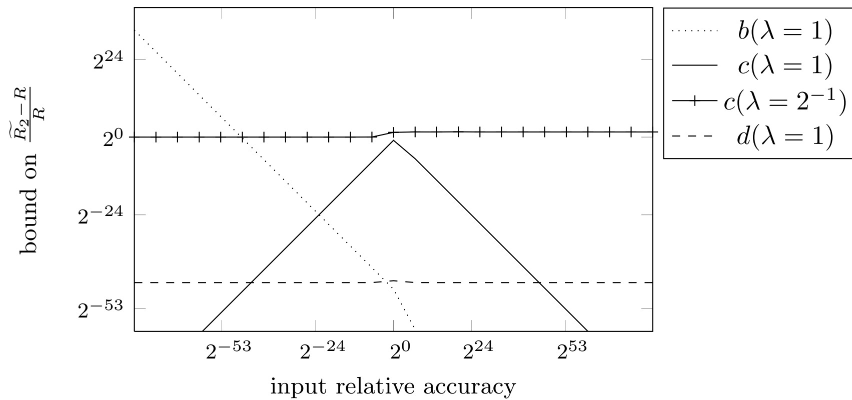

Figure 4.7: Decomposition of the Bound on the Relative Radius Error for `MMMul2` ($k = 128$).

As noted in Section 2.5, the value of λ determines two completely different behaviors for the arithmetic error. When $\lambda = 1$, the observations on Figure 4.7 are similar to those made above on Figure 4.2: the roundoff error on the midpoint computation, represented by the parameter b, dominates until the arithmetic error (parameter c) crosses, increasing as long as $e \leq 1$, then decreases till the point where the radius roundoff error is dominant. When $\lambda < 1$, the roundoff error on the midpoint computation b dominates for tight inputs until the arithmetic error c becomes preponderant. Figure 4.8 illustrates the variation of the bound $(b + c + d) \times 2/\lambda$ in both cases. Note that we neglect the coefficient $\epsilon(|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|)/\lambda$ for the same reasons as in Section 4.2.

With the random dataset I, described above, the maximum relative Hausdorff error closely follows the theoretical bound, except for very large relative accuracies (see Figure 4.9 and Table 4.4).

The same behavior in two branches appears for inputs with small relative accuracies, the branch of a particular component depending on whether its exact midpoint is computed with catastrophic cancellations or not. The conclusions are similar to the ones for the previous algorithm: the maximum measured relative Hausdorff error matches the bound given by Proposition 4.6 but this bound is usually pessimistic when the input relative accuracies are small (this

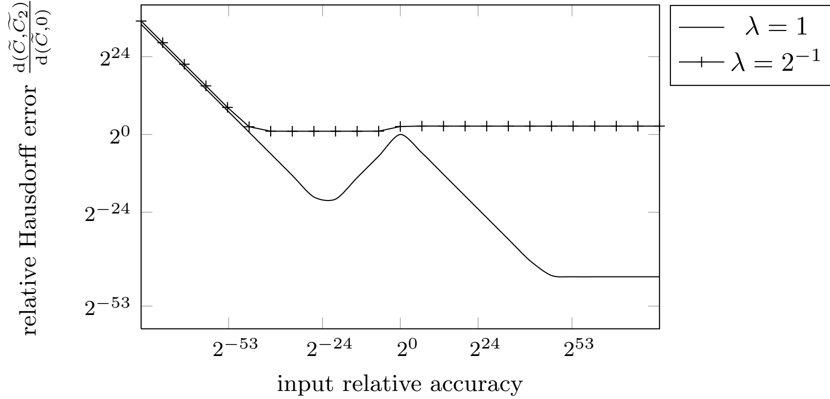

Figure 4.8: Bound on the Relative Hausdorff Error for `MMMul2` ($k = 128$).

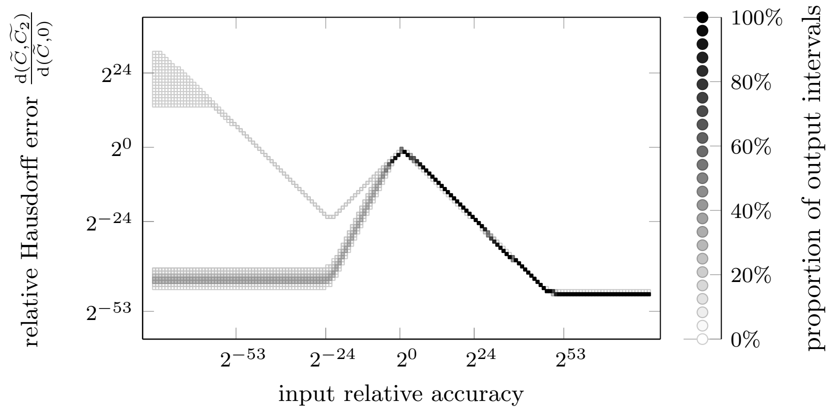

Figure 4.9: Relative Hausdorff Error for `MMMul2` with Random Dataset I (matrices of size 128×128).

Input relative accuracy	2^{-60}	2^{-53}	2^{-24}	1	2^{24}	2^{53}
max. relative Hausdorff error						
Dataset I	2^{13}	2^{6}	2^{-23}	2^{-1}	2^{-24}	2^{-47}
$2\times$ Bound (4.7) (with $\lambda = 1$)	2^{13}	2^{6}	2^{-23}	2^{0}	2^{-23}	2^{-43}

Table 4.4: Bound and Actual Values of Relative Hausdorff Error for `MMMul2` with Dataset I ($\lambda = 1$, $k = 128$).

situation is represented by the lower branch in the left part of Figure 4.9). When the input relative accuracies are large ($e > 1$, right-hand side of Figure 4.9), the bound is a good estimate of the actual relative Hausdorff error.

The relative Hausdorff error with dataset II is presented in Figure 4.10.

Again, in that case, (4.7) does not yield a finite bound since $\lambda = 0$. Like `MMMul3`, `MMMul2` produces results whose the relative Hausdorff error seems to be finite but, unlike the former algorithm, the bound (4.7) with $\lambda = 1$ and the maximum measured value do not match. Instead, the maximum relative Hausdorff error seems to follow the variation of the bound (4.7) with

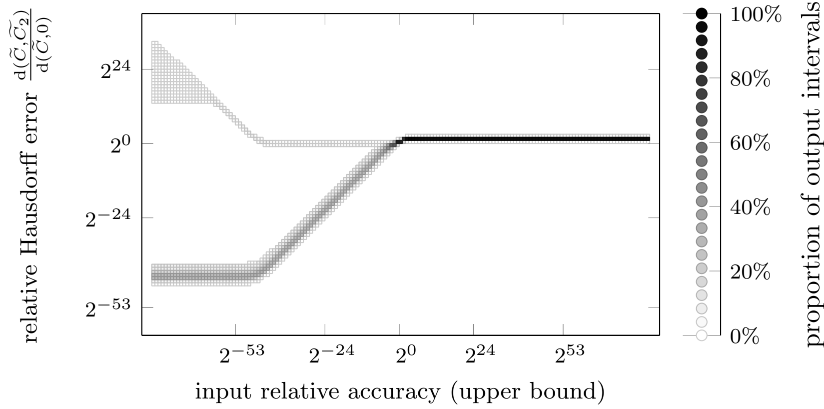

Figure 4.10: Relative Hausdorff Error for MMMul2 with Random Dataset II (matrices of size 128×128).

$\lambda = 2^{-1}$, and the numerical results in Table 4.5 corroborate this hypothesis. As noted previously, we would need a deeper analysis to prove this numerical evidence.

Input relative accuracy	2^{-60}	2^{-53}	2^{-24}	1	2^{24}	2^{53}
max. relative Hausdorff error						
Dataset II ($\lambda = 0$)	2^{14}	2^{7}	2^{0}	2^{1}	2^{2}	2^{2}
$2\times$ Bound (4.7) (with $\lambda = 2^{-1}$)	2^{16}	2^{9}	2^{2}	2^{3}	2^{3}	2^{3}

Table 4.5: Bound and Actual Values of Relative Hausdorff Error for MMMul2 with Dataset II ($\lambda = 0, k = 128$).

Moreover, two branches also appear in the case of inputs with small relative accuracy. As can be observed on Figure 4.10, the majority of computed components have a smaller relative Hausdorff error than predicted by the tentative bound (4.7) with $\lambda = 2^{-3}$. In conclusion, the situation contrasts with the case of fixed relative accuracy (dataset I), the bound is never less than 4 and we should expect a worse result when the relative accuracy of input matrices varies.

4.4 Global error for MMMul5

We conduct in this section the analysis for the relative Hausdorff error corresponding to MMMul5. First, the global error bound for the radius is given by the next proposition.

Proposition 4.7. *Let* $\boldsymbol{A} = \langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle \in \mathbb{IF}^{m\times k}$ *and* $\boldsymbol{B} = \langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}} \rangle \in \mathbb{IF}^{k\times n}$ *two interval floating-point matrices with* $4(k+1)\mathtt{u} \le 1$. *Let* $e, f,$ *and* λ *such that*

$$\begin{aligned}
e &= \max\{\mathrm{racc}(\boldsymbol{A}_{ij}) : 1 \le i \le m, 1 \le j \le k\},\\
f &= \max\{\mathrm{racc}(\boldsymbol{B}_{ij}) : 1 \le i \le k, 1 \le j \le n\},\\
\lambda &= \max\{C > 0 : Ce|\mathrm{mid}\,\boldsymbol{A}| \le \mathrm{rad}\,\boldsymbol{A} \text{ and } Ce|\mathrm{mid}\,\boldsymbol{B}| \le \mathrm{rad}\,\boldsymbol{B}\}.
\end{aligned}$$

The global error on the radius computed by Algorithm 8 is bounded as follows

$$\begin{aligned}\widetilde{R_5} - R \leq &\gamma_{2k+8}(1+e+f+ef)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + \\ &(4k+2)\mathtt{u}(1+\min\{1,e\}\min\{1,f\})|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + \\ &(\max\{\min\{e,f,ef\},1\}-1)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + \\ &4\,\mathtt{realmin}.\end{aligned} \tag{4.8}$$

Proof. Note that $|P_{\boldsymbol{A}}| \leq \min\{1,e\}|M_{\boldsymbol{A}}|$ and $|P_{\boldsymbol{B}}| \leq \min\{1,f\}|M_{\boldsymbol{B}}|$, by definition. Then, the arithmetic error

$$R_5 - R \leq (\max\{\min\{e,f,ef\},1\}-1)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|$$

comes from (2.20). The roundoff error

$$\begin{aligned}\widetilde{R_5} - R_5 \leq &\gamma_{2k+8}(1+e+f+ef)|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + \\ &(4k+2)\mathtt{u}(1+\min\{1,e\}\min\{1,f\})|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}| + \\ &4\,\mathtt{realmin}\end{aligned}$$

is a consequence of (3.14). □

The following proposition states the upper bound on the relative radius error in the special case $e = f$.

Proposition 4.8. *Let $\boldsymbol{A} \in \mathbb{IF}^{m\times k}$ and $\boldsymbol{B} \in \mathbb{IF}^{k\times n}$ two interval floating-point matrices with $4(k+1)\mathtt{u} < 1$. Let e and λ two positive real numbers such that*

$$\begin{aligned} e &= \max_{\substack{1\leq i\leq m\\ 1\leq l\leq k}}\{\operatorname{racc}(\boldsymbol{A}_{il})\} = \max_{\substack{1\leq l\leq k\\ 1\leq j\leq n}}\{\operatorname{racc}(\boldsymbol{B}_{lj})\}, \\ \lambda &= \max\{C>0 : Ce|\operatorname{mid}\boldsymbol{A}| \leq \operatorname{rad}\boldsymbol{A} \text{ and } Ce|\operatorname{mid}\boldsymbol{B}| \leq \operatorname{rad}\boldsymbol{B}\}.\end{aligned}$$

The relative radius error of the approximate product $\widetilde{C_5}$ computed by Algorithm 8 is bounded from above as follows:

$$\frac{\widetilde{R_5} - R}{R} \leq \frac{1}{\lambda}\left\{b(k,e) + c(e) + d(k,e) + \epsilon(e, |M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|)\right\} \tag{4.9}$$

where

$$b = (4k+2)\mathtt{u}\frac{1+\min\{1,e^2\}}{e\max\{2,1+\lambda e\}}, \qquad d = \gamma_{2k+8}\frac{(1+e)^2}{e\max\{2,1+\lambda e\}},$$

$$c = \max\left\{0, \frac{e-1}{e\max\{2,1+\lambda e\}}\right\}, \qquad \epsilon(e,|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|) = \frac{4\mathtt{realmin}}{e\max\{2,1+\lambda e\}|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|}.$$

Proof. (4.9) is a consequence of (2.19) and (4.8) when $e = f$. □

As in Proposition 4.4, the coefficient b arises from the roundoff error in the computation of the product midpoint. The coefficient c comes from the arithmetic error only, it behaves as described in Section 2.3. Finally, the coefficient d is used to bound the remaining roundoff errors in the radius computation.

Let us see how the bound in (4.9) varies as the input relative accuracy e grows. Both coefficients b and d decrease monotonically with e until they reach small multiples of $\mathtt{u}$ at $e = 1$. From then on, their behaviors differ: d remains almost constant, while $b \sim 1/e^2$ becomes

negligible. The sum $(b+d)/\lambda$ represents the whole bound when $e<1$, since the arithmetic error bound is zero ($c=0$) for input intervals that do not contain zero. Next, the coefficient c/λ dominates after a sudden increase around $e=1$. Then, c and d cross at $e=e_1\approx\frac{1}{(2k+8)\mathtt{u}}$. When $e_1\leq e$, the bound stays close to $(2k+8)\mathtt{u}/\lambda$, which is approximately equal to the value of the bound d on the radius roundoff error (see Figure 4.11).

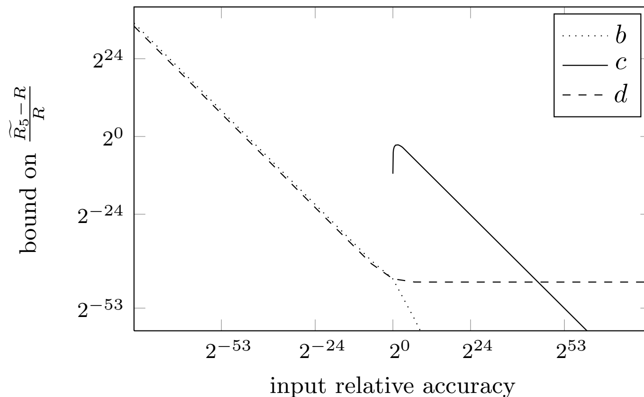

Figure 4.11: Decomposition of the Bound on the Relative Radius Error for `MMMul5` ($k=128$, $\lambda=1$).

Experimental measures of the relative Hausdorff error with dataset I (fixed input relative accuracy, $\lambda=1$) are shown in Figure 4.12.

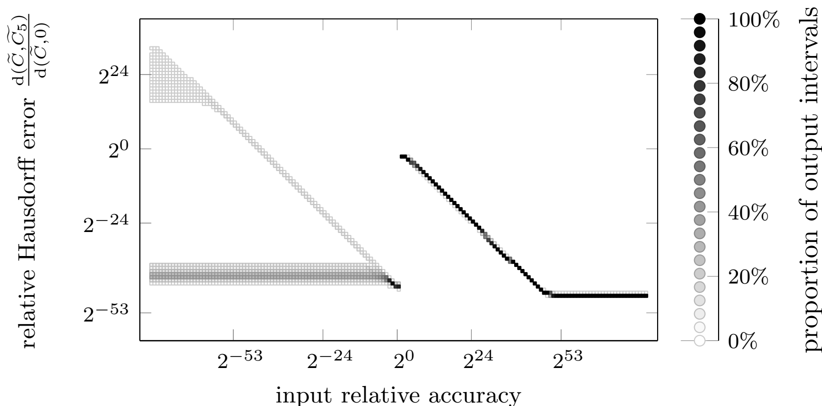

Figure 4.12: Relative Hausdorff Error for `MMul5` with Random Dataset I (matrices of size 128×128).

The conclusion is similar to that for `MMMul3` and dataset I: the bound (4.9) is a good estimate of the actual relative Hausdorff error when $e>1$, while the main part of the computed coefficients presents a tiny relative Hausdorff error, several order of magnitude smaller than the bound, when $e<1$. For small relative accuracy, however, a minority of product components still have a measured relative Hausdorff error that is close to the theoretical bound. These observations are confirmed by values displayed in Table 4.6.

Figure 4.13 presents the measured relative Hausdorff error with dataset II and the maximum

Input relative accuracy	2^{-60}	2^{-53}	2^{-24}	1	2	2^{24}	2^{53}
max. relative Hausdorff error							
Dataset I	2^{15}	2^{8}	2^{-21}	2^{-44}	2^{-3}	2^{-24}	2^{-47}
$2\times$ Bound (4.9) (with $\lambda = 1$)	2^{17}	2^{10}	2^{-18}	2^{-42}	2^{-3}	2^{-24}	2^{-43}

Table 4.6: Bound and Actual Values of Relative Hausdorff Error for `MMMul5` with Dataset I ($k = 128$, $\lambda = 1$).

relative Hausdorff errors measured for this dataset are reported in Table 4.7. As for `MMMul2`, we observe that the actual relative Hausdorff error behaves as the $\lambda = 2^{-1}$ case.

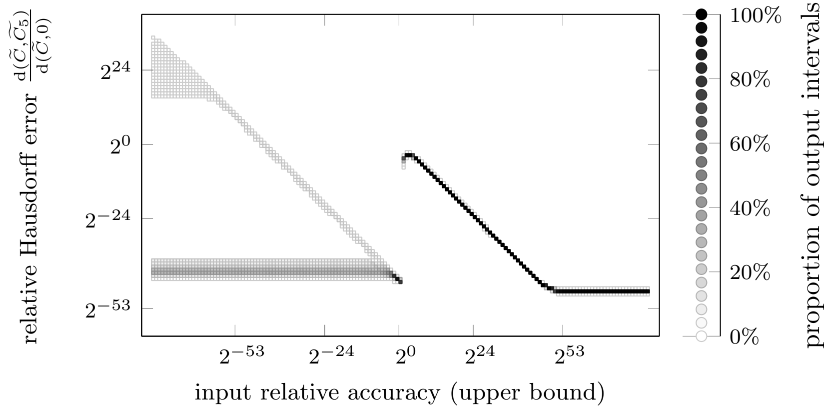

Figure 4.13: Relative Hausdorff Error for `MMul5` with Random Dataset II (matrices of size 128×128).

Input relative accuracy	2^{-60}	2^{-53}	2^{-24}	1	2	2^{24}	2^{53}
max. relative Hausdorff error							
Dataset II	2^{17}	2^{10}	2^{-19}	2^{-41}	2^{-1}	2^{-22}	2^{-43}
$2\times$ Bound (4.9) (with $\lambda = 2^{-1}$)	2^{18}	2^{11}	2^{-18}	2^{-40}	2^{0}	2^{-20}	2^{-41}

Table 4.7: Bound and Actual Values of Relative Hausdorff Error for `MMMul5` with Dataset II ($k = 128$).

4.5 Conclusion

We conclude this chapter and the part on the error analysis by comparing the three algorithms `MMMul2`, `MMMul3`, and `MMMul5`.

In previous articles [Rum12, Ngu11], error analyses for `MMMul3`, and `MMMul5` were limited to what we call here the arithmetic error. The results of such analyses are summarized in Table 4.8.

One may have the feeling, when considering these data, that some accuracy in the result can be traded for a smaller computing cost with `MMMul3`, or, reciprocally, that a greater accuracy may be reached at some computational cost with `MMMul5`. This is not always the case, as it can be

Algorithm	Computed Radius	Cost
`MMMul3`	at most 1.5× exact radius	about 3 `gemm`
`MMMul5`	at most 1.18× exact radius	about 5 `gemm`

Table 4.8: Bound on Errors and Cost Comparison for Algorithms `MMMul3` and `MMMul5`. The roundoff errors are neglected.

concluded from a global error analysis like the one performed in this part. In order to determine the exact domains where the bound on the relative Hausdorff error with a given algorithm is less than the one corresponding to the other algorithm, we compare below the results that have been established in this chapter.

First, Figure 4.14 compares, for $\lambda = 1$, the bounds on the relative radius errors for `MMMul3`, proved with Proposition 4.4, and for `MMMul5`, proved with Proposition 4.8. We can see three

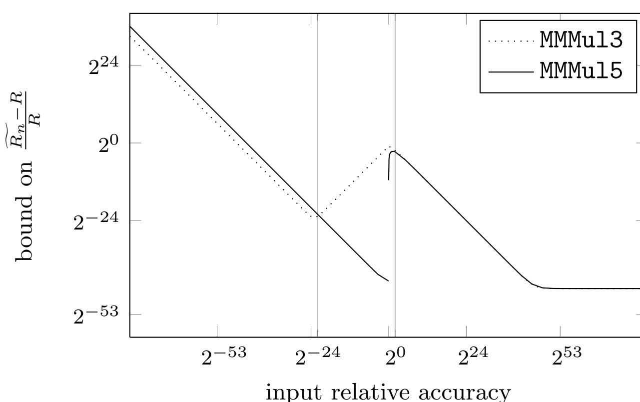

Figure 4.14: Comparison of the Upper Bounds on the Relative Radius Error for `MMMul3` and `MMMul5` (matrices of size 128×128, $\lambda = 1$).

regions on Figure 4.14. Let $e_\times$ be the point where the two graphs cross. When $e \leq e_\times$, the bound on the relative radius error of the product is less when computed by `MMMul3` than when computed by `MMMul5`. This phenomenon can be explained by the fact that roundoff errors dominate and `MMMul3` requires less computation than `MMMul5`. Solving symbolically the equality of the bounds in (4.3) and (4.9), we can approximate the crossover point $e_\times$ by $\sqrt{(4k+2)\mathtt{u}}$. In the second region, where $e_\times \leq e \leq 4$, `MMMul5` produces a better result than the ones produced by `MMMul3`. Actually, in the third region, where $4 \leq e$, the results yielded by `MMMul5` are also the best, but the negligible gain in radius overestimation is not worth the computational overhead. The starting value for this region is somewhat arbitrary, we choose here the first power of two (namely, 4) after the local maximum for the `MMMul5` bound.

Let us now compare the two algorithm with the smallest computational cost: `MMMul2` and `MMMul3`. The relevant parameter that distinguishes them is not the value of the relative accuracy of the inputs, as in the previous comparison of `MMMul3` and `MMMul5`. What have a strong effect on the relative radius error is the homogeneity of the relative accuracies. This homogeneity is measured by the λ parameter.

- When $\lambda = 1$, the relative accuracy is fixed and the bounds on $\frac{\widetilde{R_3}-R}{R}$ in (4.3) and $\frac{\widetilde{R_2}-R}{R}$ in (4.7) are very close. With the random dataset I, `MMMul2` yields results as accurate as the

ones of MMMul3. This is clearly apparent when comparing Figures 4.4 and 4.9, on the one hand, and experimental values reported in Tables 4.1 and 4.4, on the other hand.

- When $\lambda \neq 1$, the numerical results show that the relative errors for MMMul2 grow, while it does not have a strong impact on the results of MMMul3. Moreover, the global error bound for MMMul2 is always greater than the one corresponding to MMMul3 (see Figure 4.15).

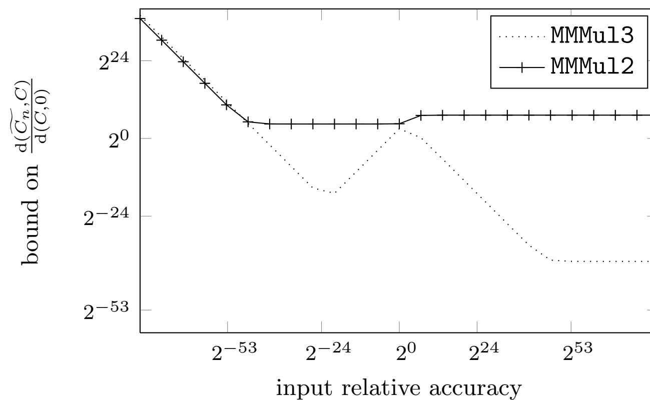

Figure 4.15: Comparison of the Upper Bounds on the Relative Hausdorff Error for MMMul3 and MMMul2 (matrices of size 128×128, $\lambda = 2^{-3}$).

As a conclusion, based on the bound of the relative Hausdorff error, we can construct the following decision tree (Figure 4.16) to select the best efficiency-accuracy tradeoff given the relative accuracies for the components of the input matrices.

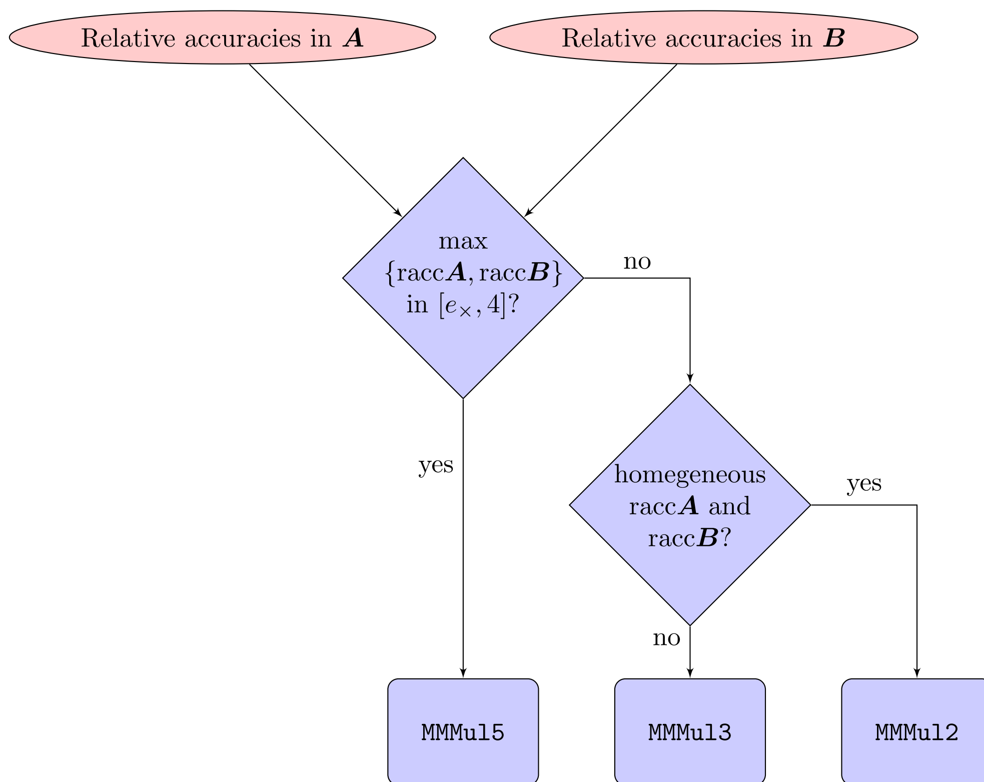

Figure 4.16: How to choose the most precise and less expensive algorithm for interval matrix multiplication.

Part II

Parallel Implementation

Parallel interval linear algebra on multi-core processors

Algorithms for the interval matrix multiplication, like those presented in Chapter 3, can be implemented in two ways. One can either call numerical linear algebra routines to compute the floating-point matrix products, as promoted by their authors, or one can use custom routines, merging two products for instance. Even though the former approach requires much less programming effort, we will follow the latter, mainly because of correctness reasons.

On the one hand, the execution time of a program is highly dependent on the actual processor that executes it. On the other hand, high-level programming languages use abstract models for the memory and processing units in order to be portable across different architectures. Consequently, an efficient implementation of a given algorithm has to take into account the relevant characteristics of the processor and to be written in such a way that the executing program makes use of these characteristics.

For the implementation of the interval matrix multiplication, we target current commodity processors, specifically the family of x86 multi-core processors. Such processors are widespread and cheap. They are used in general-purpose as well as high-performance computers. So, the restriction to these processors is not a severe constraint. Moreover, x86 processors come with a large choice of programming and computing tools. This lessens significantly the programming effort for the implementation. In particular, two extensions to the C language, the intrinsics functions and the OpenMP parallel constructs, will help us to express data and task parallelism in the code, which will yield an efficient parallel implementation of the interval matrix multiplication.

Since the early 2000's, x86 processors are composed of several identical cores that duplicate the processing units and share the same memory bus. The individual computing power of the cores is therefore multiplied by their number, while the individual memory throughput is divided when memory is accessed concurrently. The change is relatively new and the design of numerical algorithms for processor architectures with multiple cores is an active field of research. We show here that it is possible to implement efficiently the interval matrix multiplication on multi-core processors and we exhibit the similarities and differences between multiplication algorithms for floating-point matrices, on the one hand, and for interval matrices, on the other hand.

The present part on parallel implementation is organized as follows. While algorithms for the interval matrix multiplication that use the midpoint-radius representation promise efficiency, ease of programming and portability, they rely on several implicit assumptions that complexify their actual parallel implementation. Chapter 5 makes these assumptions more explicit and show how they affect the implementation. The difficulty to actually fulfill those prerequisites motivates the need for a different approach for implementing such algorithms. The method we followed is defined and discussed in Chapter 6. The actual implementation of a simple and new interval matrix multiplication (Algorithm 9, page 48) is detailed using two successive points of

view. Chapter 7 presents how we exploit the parallelism on one core at the instruction level, and Chapter 8 deals with the multi-core multi-threaded part of the implementation. The last chapter contains the conclusions of this study with possible improvements and applications.

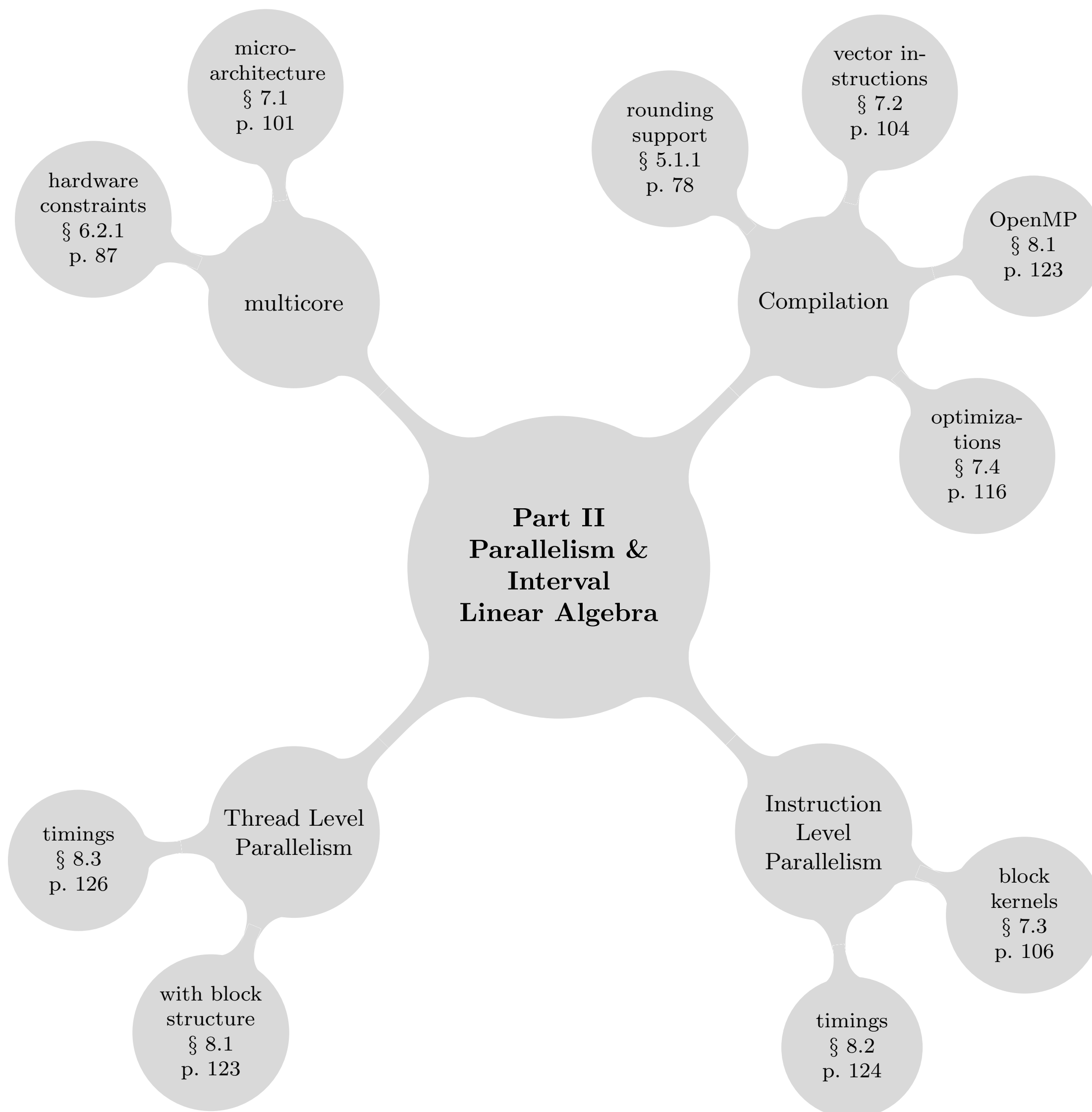

Figure 4.17: Synopsis of Part II.

Chapter 5

Implementation issues with regard to interval linear algebra

The inclusion property is fundamental for the correctness of algebraic operations with interval quantities. Ensuring this property all along the computation is therefore an uttermost requirement if the interval result is to be presented as a guaranteed enclosure of the exact result. While arithmetic operations on floating-point intervals are easily implemented in floating-point arithmetic, in such a way that the inclusion property is provably verified, many issues may well be overlooked when dealing with more complex operations. In this chapter, we point out two major sources of failure concerning interval matrix algorithms.

As seen above, the interval matrix multiplication can be done by means of floating-point matrix products. In what follows, we detail the implementation issues of `MMMul3` (Algorithm 4, page 42), as a case study. Part of the following discussion has been published in [RT14].

5.1 Rounding modes

Several techniques may be used to take the roundoff error into account. A simple one is to systematically enlarge the computed result by multiplying it with an appropriate coefficient, whose value is usually one plus a small multiple of the unit roundoff `u`, and to add to the result a small quantity as a guard for a possible underflow (see [RZBM], for example). This straightforward approach has been used for a long time (for instance, in Profil/BIAS [Knü94] for some platforms), until most processors support the directed rounding modes. Actually, it is possible that floating-point computations output exact value when the result is representable as a floating-point number. In such circumstances, systematic radius augmentation tends to induce much larger intervals and requires anyway more floating-point operations than the corresponding computation with directed rounding.

Indeed, the other way to automatically accumulate roundoff errors in the radius of the result is to compute it with rounding toward $+\infty$. The output is then an overestimate of the exact value when the computation involves only a sequence of additions and multiplications. While this seems to be also the case for matrix multiplications, this method may lead to incorrect results in two ways: when the rounding mode is not taken into account and when the computation with rounding toward $+\infty$ does not produce an overestimated result. We examine both cases in turn below.

5.1.1 Language and compiler support

The directed rounding modes are mandatory on an IEEE-754 compliant platform [IEE08]. Fortunately, most recent processors respect this floating-point standard, whose first version was published in 1985. As a consequence, we can encounter two kinds of instruction sets in such processors. The first kind provides a low level instruction to change the rounding mode. It is then represented as a state variable, which is global in a given process context, but specific to each process. In the second kind, the instruction corresponding to each floating-point operation exists in several versions, one for each rounding mode. The latter is true for graphic processor units (GPU) and some general purpose processors (for instance, the Intel Xeon Phi processor [Int]). In the following, we will discuss only the x86 processors, which are widespread and belong to the first kind.

Rounding modes being accessible by the programmer at the assembly level, designers of programming language were encouraged to provide support at a higher level. For instance, the C99 standard revision [ISO99] of the C language provides two functions `fegetround` and `fesetround` to access and set the current rounding mode at run-time. The point, for our concern, is that these functions are not well supported even by up-to-date compilers, as illustrated by the following example (Listing 5.1, excerpt of the GCC bug report #34678 [gcc08], still not fixed).

```
#include <fenv.h>

void xdiv (double x, double y, double* lo, double* hi)
{
  #pragma STDC FENV_ACCESS ON

  fesetround(FE_DOWNWARD);
  *lo = x/y;
  fesetround(FE_UPWARD);
  *hi = x/y;
}
```

Listing 5.1: Interval Division.

The previous code snippet is meant to compute the interval enclosure [`lo`, `hi`] of a floating-point quotient `x/y`. In fact, the GCC compiler produces a erroneous assembly code even though correct options are set. Considering that the division is a costly operation, the optimization phases simply copy the first computation result in the right endpoint variable, saving the cost of the second division, and ignoring the in-between change of the rounding mode. The interval result is therefore always reduced to a single point and may not contain the exact value. Admittedly, this could be avoided by disabling all optimizations associated to floating-point arithmetic, but this bug is an indication that the rounding mode in not well represented in the compiler intermediate language. Thus, when using the `fesetround` function and a compiler based on GCC, bugs related to rounding mode in a more subtle manner than in the previous example are susceptible to appear fortuitously in interval computations.

5.1.2 Library support

The main advantage of `MMMul3` is that it explicitly uses products of floating-point matrices. Therefore, it can be implemented using optimized BLAS libraries, reducing the implementation effort while getting good performance and portability. Yet, this algorithm changes the rounding mode during the computation and depends on the fact that the numerical library in use respects the given rounding mode. Many libraries do not so and assume that the default rounding mode

(rounding to nearest) is set. A first reason is that this behavior is explicitly permitted by the C99 standard [ISO99, 7.6 Floating-point environment `<fenv.h>`]:

> Certain programming conventions support the intended model of use for the floating-point environment:175)
>
> – [. . .]
>
> – a function call is assumed to require default floating-point control modes, unless its documentation promises otherwise;
>
> – [. . .]

with the associated note:

> 175) With these conventions, a programmer can safely assume default floating-point control modes (or be unaware of them). The responsibilities associated with accessing the floating-point environment fall on the programmer or program that does so explicitly.

So, it is clear from the previous excerpt of the C99 standard that the directed rounding modes may validly be ignored by libraries. Moreover, the definition of Basic Linear Algebra Subprograms only distinguishes *proper rounding*, meaning correct rounding to nearest, and *IEEE rounding*, meaning correct rounding to nearest with ties to even [BCD+01, 1.6 Numerical Accuracy and Environmental Enquiry]. The notion of directed rounding mode does not appear anywhere in the BLAS standard.

Similarly, parallel programming facilities often pay no attention to the rounding modes. For instance, while it is not explicitly mentioned, we can deduce from the following excerpt of the OpenMP specification [Ope13, 1.6 Normative References] that the generated threads do not inherit the rounding mode set by their parent (the “Fortran 2003 Section 14” it refers to addresses rounding modes):

> This OpenMP API specification refers to ISO/IEC 1539-1:2004 as Fortran 2003.
>
> The following features are not supported:
>
> - IEEE Arithmetic issues covered in Fortran 2003 Section 14
> - [. . .]

The OpenCL standard is more explicit (from [Khr11, 7.1 Rounding Modes]):

> Round to nearest even is currently the only rounding mode required by the OpenCL specification for single precision and double precision operations and is therefore the default rounding mode. In addition, only static selection of rounding mode is supported. Dynamically reconfiguring the rounding modes as specified by the IEEE 754 spec is unsupported.

Hence, parallel linear algebra libraries implemented with OpenMP or OpenCL are unlikely to support rounding modes other than the rounding to nearest.

In some cases, it is even certain that computation with directed rounding modes will produce incorrect results. This is the case with extended precision BLAS, described in [BCD+01, 4 Extended and Mixed Precision BLAS]. The reference implementation of these functions [LDB+02] uses a double-double arithmetic to simulate a higher working precision in the computation. With

respect to our matter, it suffices to know that double-double arithmetic requires rounding to nearest to be able to determine the error of the most significant part of the pair (see, for instance, [MBdD+10, 14 Extending the precision]).

Besides, when computing $\widetilde{R_3}$ at line 2 in Algorithm 4 (page 42), we do want to sum overestimates of the matrix products $R_{\boldsymbol{A}}(|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}})$ and $|M_{\boldsymbol{A}}|(R_{\boldsymbol{B}} + k\mathtt{u}|M_{\boldsymbol{B}}|)$. In that case, the notation $\mathrm{fl}_{\Delta}(XY)$ is somewhat misleading: the fact that all intermediate computations are done in rounding toward $+\infty$ does not necessarily guarantee that the computed result overestimates the exact one. In fact, matrix multiplications can be done in several manners. We illustrate here two possible algorithms, and show how they differ with respect to a global rounding mode.

Let A, B, and C be three square matrices of dimension 2^n with $A \geq 0$ and $B \geq 0$. The first algorithm that we examine is the widespread recursive block multiplication. Given the following decomposition of A, B, and C into sub-matrices of dimension 2^{n-1}:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22}, \end{pmatrix},$$

the product C can be computed as follows

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned}$$

Each product $A_{ik}B_{kj}$ of sub-matrices in turn can be recursively computed by the previous formulas. Counting the matrix operations, we see that the classical recursive scheme needs 8 sub-matrix products at each step of the recursion. From the point of view of the rounding mode used for the computation, the current scheme is compatible with the rounding toward $+\infty$ mode, yielding an overestimate of the product (*mutatis mutandis* an underestimate with rounding toward $-\infty$). In fact, if $\widetilde{C_{11}} = \mathrm{fl}_{\Delta}(\widetilde{A_{11}}\widetilde{B_{11}} + \widetilde{A_{12}}\widetilde{B_{21}})$ is the floating-point computation of C_{11} with $\widetilde{A_{ij}} \geq A_{ij}$ and $\widetilde{B_{ij}} \geq B_{ij}$, then the result verifies $\widetilde{C_{11}} \geq C_{11}$.

Another way to perform the matrix multiplication is the Strassen's algorithm [Str69]. Given the same sub-matrix decomposition as above, we compute the following quantities:

$$\begin{aligned} T_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\ T_2 &= (A_{21} + A_{22})B_{11}, \\ T_3 &= A_{11}(B_{12} - B_{22}), \\ T_4 &= A_{22}(-B_{11} + B_{22}), \\ T_5 &= (A_{11} + A_{12})B_{22}, \\ T_6 &= (-A_{11} + A_{21})(B_{11} + B_{12}), \\ T_7 &= (A_{12} - A_{22})(B_{21} + B_{22}). \end{aligned}$$

Then, we compute C's sub-matrices with the following sums and differences

$$\begin{aligned} C_{11} &= T_1 + T_4 - T_5 + T_6, \\ C_{12} &= T_2 + T_4, \\ C_{21} &= T_3 + T_5, \\ C_{22} &= T_1 + T_3 - T_2 + T_6. \end{aligned}$$

In exact arithmetic, this algorithm requires one sub-matrix product less than the classical recursive scheme. However, the floating-point implementations of Strassen's algorithm are not compatible with directed rounding. They are able to deliver only approximate result, not guaranteed overestimate, nor underestimate. Let us see why it does not compute an overestimate with rounding toward $+\infty$. In order to have $\widetilde{C_{11}} \geq C_{11}$ with $\widetilde{C_{11}} = \mathrm{fl}_\Delta(\widetilde{T_1} + \widetilde{T_4} - \widetilde{T_5} + \widetilde{T_6})$, we need three overestimates $\widetilde{T_1} \geq T_1$, $\widetilde{T_4} \geq T_4$, and $\widetilde{T_6} \geq T_6$, as well as one underestimate $\widetilde{T_5} \leq T_5$. For $\widetilde{C_{21}} \geq C_{21}$ with $\widetilde{C_{21}} = \mathrm{fl}_\Delta(\widetilde{T_3} + \widetilde{T_5})$, we need overestimates $\widetilde{T_3} \geq T_3$ and $\widetilde{T_5} \geq T_5$. Actually, T_5 has to be computed twice: once for an overestimate and once for an underestimate. The same double computation is also required for T_2. Therefore, instead of 7 sub-matrix multiplications, the computation of an overestimate of a matrix product with Strassen's algorithm demands 9 floating-point sub-matrix products, more than the classic recursive block algorithm!

Other matrix multiplication algorithms that have a lesser count of arithmetic operations than the recursive block product algorithm are based on such reordering of operations with cancellations. No implementation of a fast matrix product algorithm would take directed rounding modes as a means to indicate the expected direction of the overall error, because this would ruin the advantage of doing less sub-matrix multiplications than the simple recursive algorithm.

In conclusion, a result computed with the `MMMul3` algorithm verifies the inclusion principle only if we can ensure that the computation of the matrix product complies with the correct rounding mode and that the result is an actual overestimate of the exact product when the rounding mode toward $+\infty$ is set. And this very last point is possibly not verified, depending on the internal, and possibly selected at run-time, algorithms used by the library. As far as we know, the issue of the respect of the rounding mode by numerical and multi-threaded libraries when computing with intervals was first raised by Lauter and Ménissier-Morain in [LMM12].

5.1.3 Example of rounding mode violation

The behavior exposed in the previous theoretical deduction is actually observable as shown by the following example. Let A be the following n-by-n matrix

$$A = \begin{pmatrix} 1 & 0 & \cdots & 0 & \mathtt{u} \\ 0 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ \vdots & & \ddots & 1 & \vdots \\ 0 & \cdots & \cdots & 0 & \mathtt{u} \end{pmatrix},$$

where $\mathtt{u} = 2^{-53}$ is the roundoff unit. The exact product AA^T is

$$AA^T = \begin{pmatrix} 1+\mathtt{u}^2 & \mathtt{u}^2 & \cdots & \mathtt{u}^2 \\ \mathtt{u}^2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1+\mathtt{u}^2 & \mathtt{u}^2 \\ \mathtt{u}^2 & \cdots & \mathtt{u}^2 & \mathtt{u}^2 \end{pmatrix}.$$

The correctly rounded answers are therefore:

$$\mathrm{fl}_\Box(AA^T) = \begin{pmatrix} 1 & \mathtt{u}^2 & \cdots & \mathtt{u}^2 \\ \mathtt{u}^2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1 & \mathtt{u}^2 \\ \mathtt{u}^2 & \cdots & \mathtt{u}^2 & \mathtt{u}^2 \end{pmatrix}, \qquad \mathrm{fl}_\Delta(AA^T) = \begin{pmatrix} 1+2\mathtt{u} & \mathtt{u}^2 & \cdots & \mathtt{u}^2 \\ \mathtt{u}^2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1+2\mathtt{u} & \mathtt{u}^2 \\ \mathtt{u}^2 & \cdots & \mathtt{u}^2 & \mathtt{u}^2 \end{pmatrix}.$$

The above example computation can be implemented as in Listing 5.2. We checked the results

```
double *A;
double *C;

A = (double*)malloc (n * n * sizeof(double));
C = (double*)malloc (n * n * sizeof(double));

memset (A, 0, n * n * sizeof(double));
for (i = 0; i < n; i++) A[i * n + i]      = 1.0;
for (i = 0; i < n; i++) A[i * n + n - 1] = 0x1p-53;

#pragma STDC FENV_ACCESS ON
fesetround (FE_UPWARD);
cblas_dgemm (CblasRowMajor, CblasnoTrans, CblasTrans,
             n, n, n, 1.0, A, n, A, n, 0.0, C, n);
```

Listing 5.2: Rounding Mode Violation.

of the previous program with several BLAS libraries on a Core2 machine for `A` and `C` of dimension $n = 2^k$, with $k = 1, \ldots, 15$. When linked against the Intel BLAS library (MKL version 11.0.2) or the ATLAS library (version 3.8.4), it computes the correctly rounded matrix $\mathrm{fl}_\Delta(AA^T)$ for both sequential and multithreaded executions. When linked against the OpenBLAS library (version 0.1.0, based on GotoBLAS2 version 1.13), it computes the correct value in a sequential execution. Correct results are also returned when $n < 16$. However, if $n \geq 16$ and the thread number is not limited to 1, then the OpenBLAS library computes wrong values for the last three quarters of the diagonal components. For instance when $n = 1024$, components C_{ii} are equal to 1 instead of $1 + 2\mathtt{u}$ for $i = 256, \ldots, 1022$[1]. This suggests that $n = 16$ is the threshold beyond which the product computation is distributed among several threads. It also suggests that the rounding mode is not inherited from the parent thread.

In conclusion, the previous discussion shows that we cannot expect that a BLAS library will compute an overestimate of a matrix product by simply changing the mode to rounding toward $+\infty$.

5.2 Execution order

As recalled in Section 3.2.1, Rump saved one matrix multiplication in `MMMul3` compared to his original algorithm published in 1999. In the latter (see Algorithm 2, page 41), the roundoff error in the product of midpoint matrices is taken into account as follows. The product of midpoint matrices is computed twice, once with rounding toward $-\infty$ and it is then added to the infimum endpoint, and once with toward $+\infty$, then added to the supremum endpoint. We have shown in Section 5.1 that this method may not produce the expected results. Nevertheless, Rump proves the following bound on the midpoint calculation error (Theorem 3.4 page 42, reproduced below for ease of reading). It is important to note that this bound only requires computations with the default rounding to nearest mode. Its flaw now resides in its assumptions.

Theorem (Theorem 2.1 in [Rum12])**.** *Let $A \in \mathbb{F}^{m\times k}$ and $B \in \mathbb{F}^{k\times n}$ with $2(k+2)\mathtt{u} \leq 1$ be given, and let $C = \mathrm{fl}_\square(AB)$ and $\Gamma = \mathrm{fl}_\square(|A||B|)$. Here C may be computed in any order, and we assume*

[1]Here, we use the C convention, array index starts from 0.

that Γ *is computed in the same order. Then*

$$|\mathrm{fl}_\Box(AB) - AB| \leq \mathrm{fl}_\Box\left(\frac{k+2}{2}\mathrm{ulp}(\Gamma) + \texttt{realmin}\right). \tag{5.1}$$

We can use the bound provided by Rump's Theorem only if all its hypotheses are satisfied. In particular, MMMul3 (Algorithm 4, page 42) requires that the product $\mathrm{fl}_\Box\left(M_{\boldsymbol{A}}M_{\boldsymbol{B}}\right)$ (line 1) and the product[2] $\mathrm{fl}_\Delta\left(|M_{\boldsymbol{A}}|(R_{\boldsymbol{B}} + (k+2)\mathrm{ulp}|M_{\boldsymbol{B}}|)\right)$ (line 2) are computed in the same order. This last condition may be difficult to ensure, especially if the computation is parallel.

In that respect, let us note that some vendors decided to address the problem of the reproducibility of numerical results between different processors or from run to run. Indeed, the version 11.0 of MKL now provides special modes of execution [Tod12] where the user can control, at some loss in performance, the scheduling of internal tasks and the type of computing kernels in use. In these modes, identical numerical results are guaranteed on different processors when they share the same architecture and run the same operating system. Moreover, reproducibility from run to run is ensured under the condition that, in all executions, the matrices have identical memory alignment and the number of threads remains constant. Some users requested this functionality because of their legal obligations or verification constraints.

We can use this kind of control to solve the problem of the computation order of AB and $|A||B|$. As the processor and the operating system remain the same during the computation of the two products computations, it suffices, first, to compute AB, second, to transform in place matrix components into their absolute values ensuring the identity of memory alignments, then third, to recompute the product on the new input with the same number of threads. The drawback of this solution is its specificity to the Intel library as long as others do not adopt a compatible means for controlling the numerical reproducibility.

5.3 Conclusion

The fact that directed rounding modes may be a problem when using BLAS libraries is well known. For instance, when dealing with interval matrix multiplication in [OR02], Oishi and Rump explicitly make the assumption that it is possible to "switch the rounding mode" and they remark that this hypothesis is not sufficient when the matrix multiplication is realized with a fast algorithm, like Strassen's. However, the problem of respect of the rounding mode by libraries, for which, as we have seen here, there is no guarantee, is completely overlooked. As the BLAS libraries try to exploit all the available computing power, they are likely to be executed in parallel on current multi-core processors. And the same parallel execution that provides short execution times also questions the prerequisites of a guaranteed overestimation. One should avoid such situations in certified computing.

In the next chapters, we present how to implement the algorithms for interval matrix products without using any BLAS library and still avoiding the issues listed here.

[2] This value obviously overestimates the bound $\mathrm{fl}_\Delta\left(|M_{\boldsymbol{A}}|R_{\boldsymbol{B}}\right) + \mathrm{fl}_\Box\left((k+2)\mathrm{ulp}|M_{\boldsymbol{B}}|\right)$ that is derived from the theorem.

Chapter 6

Implementation methodology

Because they rely on floating-point matrix-matrix multiplications, algorithms similar to `MMMul3` (Algorithm 4, page 42) were thought as easy to implement efficiently. However, some care is needed and the previous chapter discussed some implementation issues that are easily overlooked. It follows from these observations that BLAS libraries cannot be directly used to implement interval matrix multiplication algorithms. In the following, we focus on the implementation of one particular algorithm: the `MMMul2` interval matrix product (Algorithm 9, page 48), which is simpler than `MMMul3`. All techniques for the implementation of `MMMul2` that we describe below can be applied for other algorithms based on the midpoint-radius representation of interval matrices.

From now on, we present a global methodology for the implementation of dense interval matrix product algorithms (this chapter), going into deeper detail with a direct implementation of a particular interval matrix multiplication algorithm (the next Chapters 7 and 8).

6.1 Priority list of implementation goals

This section addresses the question: what is an efficient parallel implementation of an algorithm for interval matrix multiplication when the targeted platform is an x86 multi-core processor? We give a list of desirable properties of the implementation, motivate their choice, propose criteria for deciding on their accomplishment, when possible, and discuss their relative importance. These properties will guide the choice of implementation techniques described in the next section.

We aim at the four objectives presented below in decreasing order of priority.

- *Correctness.* The output interval shall include the exact mathematical result.
- *Sequential performance.* The timings should be comparable with the execution time of a floating-point matrix product (`dgemm`).
- *Scalability.* The speed-up of a multi-threaded execution should be as good as the corresponding speed-up of `dgemm` in up-to-date BLAS libraries.
- *Portability.* The code should be as independent of the platform, compiler, and libraries as possible.

Correctness is needed because we want to promote the interval computing as a means to guarantee the output. The inclusion property allows one to do sound reasoning with the computed results and this counterbalances the unavoidable overhead in execution time and memory space.

The next goal is to obtain an *efficient sequential implementation*. The sequential efficiency is a measurable goal and we aim at an implementation reaching at least 75% of the maximal possible performance for the algorithm on the given platform. This has to be compared with the 90% peak performance level often reached by current implementations of `dgemm`. It is a lower threshold, but, as will be seen below, the comparison with the peak performance may not be relevant for algorithms other than the floating-point matrix multiplication.

The third aim is the *scalability* of the multi-threaded part of the computation. More precisely, the execution time shall decrease for a given matrix product as it is performed with more processing units (strong scalability). In our case, the processing units are the multiple cores of the processor. Scalability is also a measurable goal: we aim at a speed-up as good as the one measured with the MKL implementation of `dgemm`. This goal does not clash with the previous one because matrix product components can be computed independently. So, the data dependence is limited to the computation of a given component, while the computations of different components can be performed by different threads and in any order. This allows us to map data and computation to the underlying architecture by distinguishing the in-core and mono-threaded part of the computation from the multi-core multi-threaded part. Thanks to this separation into levels, we can employ simple abstract models of the underlying hardware and well-known implementation techniques of numerical linear algebra.

The last objective, *portability*, is the least quantifiable. It is also in conflict with sequential efficiency and scalability, which are very dependent on the platform. However, a portable implementation may be useful in a great extent of contexts and this quality is an indication that the techniques used to achieve the previous goals are not too specific. We have no definite metric to measure the achievement of this goal. Instead, we shall list parameters that affect the performance and the scalability with portability in mind.

First, the code should be independent of the platform, which we define here as the combination of the hardware and operating system in use. The processors we are aiming at are already defined: we restrict ourselves to the x86 multi-core processors. But, even limited to this family, the set of such processors presents a diversity of micro-architectures that has an influence on the performance.

Our implementation should not depend on the operating system. Numerical functions make little use of system calls, so this may not be a strong constraint. Nonetheless, the application binary interface may differ between different operating systems and even versions, i.e. 32-bit versus 64-bit, of the same system. This has deep implications on the code if it is directly written in assembly language. To avoid this issue, we shall use a higher level language and let the compiler deal with such details.

That very last choice has to be balanced by the fact that we want an implementation with performance characteristics that are as much independent of the compiler in use as possible. This means that we cannot rely on specific optimization passes of a peculiar compiler version and that we must employ only widespread language features, and common optimization techniques. In spite of this restriction, the compiler continues to be an important factor of the overall performance and variations in efficiency with respect to the compilation phase will be reported.

The last parameter that we have to withdraw from our performance analysis relates the libraries that are used by the program. As seen in Chapter 5, BLAS libraries are troublesome for interval linear algebra. We solve this issue by not using any BLAS function in our code. Instead, we write an interval matrix multiplication function from scratch. The only library we need is a system for the threads management. The chosen one shall be taken into account in the scalability analysis of the program.

Lastly, note that the ease of coding is no more one of our goals, in contrast to the explicit intention of other authors of midpoint-radius matrix algorithms. We rather put the correctness

in the first place. This choice has an obvious cost in terms of programming effort, but this situation is usual in high-performance computing.

6.2 Parallel linear algebra on multi-core processors

This section deals with the question of the expression of the parallelism in a program that performs linear algebra computation on a multicore platform. We first expose the details of the architecture of a x86 multicore multiprocessor that have the most important effects on parallel numerical computations. Next, we examine some possible ways to express parallelism at the software level. Finally, we explain how our choice of data structure is adapted to the efficient implementation of algorithms for interval matrix multiplications.

6.2.1 Hardware constraints

Parallelism at the hardware level means that several processing units can be used simultaneously. Knowing how these processing units are organized is a necessary condition to their efficient utilization. We present here a hierarchical view of the computing platform and we stress the constraints of their use.

At the highest level, the platform is composed of one or several computing units, the processors, linked to one or several memory units. Figure 6.1 shows the general structure of our testing platform as an example. Each processor (P0, P1, P2, and P3, resp.) is linked to a memory module (M0, M1, M2, and M3, resp.). Each processor can access any memory module but the access time is the lowest for the module that is attached to it. Otherwise, the access time increases with the number of processors traversed for accessing the memory module where data are stored. This is typical of NUMA[1] architectures.

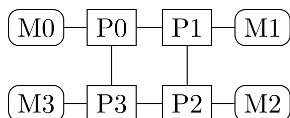

Figure 6.1: High Level View of the Testing Machine.

Our testing platform is composed of identical Intel Xeon E5-4620 processors. Table 6.1 below gives some technical specifications. The general processor architecture is broadly described below

number of cores	8
architecture model	Sandy Bridge
clock speed (max.)	2.20GHz
SIMD instruction set	SSE4.2, AVX
L1 data caches	32KB per core
L2 caches	256KB per core
L3 cache	16MB shared

Table 6.1: Intel Xeon E5-4620 Processor.

and the characteristic of the Sandy Bridge model are more precisely discussed in Section 7.1. The clock speed of the processor determines the rate at which instructions are executed. It may vary

[1]NUMA stands for Non-Uniform Memory Access.

under some circumstances and this point is addressed in Section 6.3 as it affects the measures of execution times. Modern x86 processors can execute a single instruction on several data at each clock cycle. Theses instructions are named *SIMD*[2] instructions or *vector* instructions. The Sandy Bridge model has two different SIMD instruction sets: SSE instructions can process vectors of two floating-point components in double precision and AVX instructions process vectors twice as large as SSE vectors. Finally, the external memory is backed by internal memory caches in order to alleviate the cost of data access. The system of caches plays an important role in the efficiency of parallel program and is detailed below. Let us note here from the data in Table 6.1 that the processor contains three different levels of caches with increasing sizes.

From the point of view of the parallelism supported by the hardware, the processor is not the basic processing unit. In fact, a single processor is divided in multiple cores that can execute in parallel different programs or different execution threads of the same program, as do multiple processors. The main difference with the higher multi-processor level is that the cores in a processor share the access bus to the memory, while different processors on a same platform may have independent accesses to the memory, provided they address different memory modules (see Figure 6.1).

In turn, the cores are composed of several functional units that can process different instructions. Each functional unit is dedicated to a certain kind of instruction, so the parallelism at this level is more restricted than at the multi-core level. Constraints of this level are detailed in depth in the next chapter.

In addition, each core is able to handle two execution threads at a time and tries to assign to a given thread the resources that are not used by the other thread. This feature is called *Symmetric Multi-Threading* (SMT), or *hyperthreading* in Intel processors. It can be beneficial when the threads that are executed concurrently require different resources or when one of the thread is frequently idle, waiting for data. We will not use this capability because of the great regularity of the computation performed by each thread.

As an example of this hierarchy of processing units, our main platform for experimental measures is depicted in Figure 6.2. The machine consists of four identical multi-core processors (Socket P#0 to Socket P#3). Each node is an eight-core processor (namely, an Intel Xeon E5-4620), the cores being labeled Core P#0 to Core P#7. Inner details of the cores are not shown. Because the four processors share the same memory space, by default the operating system does not distinguish the cores and treats them as equivalent processing units. Hence, the cores are also globally numbered from PU#0 to PU#31.

So, from the processors to the functional units, we have a stack of processing components with decreasing power and increasing constraints. Superimposed to this hierarchy of processing units, the memory is also arranged in a hierarchical structure. Actually, the access to the main memory is slow. To improve the latency of memory operations, a system of intermediate memory areas takes place into the processor itself. It consists in several memory caches with a decreasing access time but also a decreasing size, due to an increasing cost. Each core has a private area (caches of levels 1 and 2 in Sandy Bridge processors) and all cores of a given processor share the last level of cache (level 3 in our case). For the testing machine, Figure 6.2 shows the memory hierarchy with three levels of cache denoted L1, L2, L3. The first two levels L1 and L2 are private to the core, while the last level L3 is shared among the cores of a same socket.

In the following, we only describe the particular details of cache functioning that will determine our data structure for an interval matrix and our implementation of matrix products. In particular, we omit here the set structure and associativity of cache, the mechanism of coherency between caches in different cores, the page structure of the memory, and issues related to the limitation of the translation look-aside buffer. For more information, one can refer to the thor-

[2]SIMD stands for Single Instruction Multiple Data.

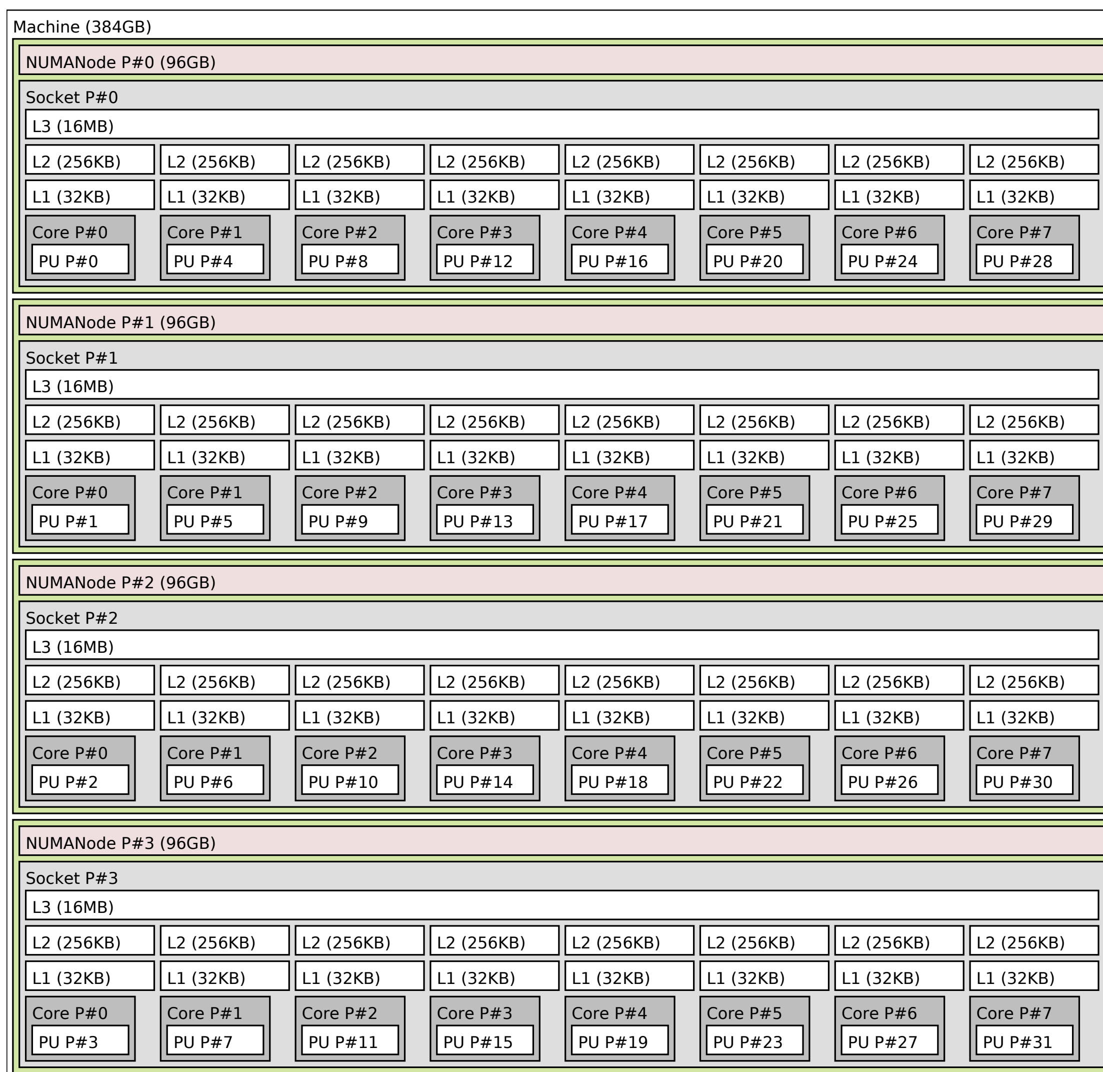

Figure 6.2: Multi-Core Multi-Processor Machine Synopsis. Output of the `lstopo` command of the Portable Hardware Locality (hwloc) software package.

ough review of the memory cache mechanism in the famous book of Hennessy and Patterson [HP07, § 5: Memory Hierarchy Design *and* Appendix C: Review of Memory Hierarchy].

Memory caches contain copies of pieces of information that reside in the main memory. Individual pieces of information are identified and referred to by their address. When the processor looks for a piece of information, it is first searched into the first levels of cache. If the sought information is found there, what is called a *cache hit*, data is brought back to the core registers in a handful of clock cycles. Otherwise (*cache miss*), the last level of cache is scanned for the requested data. In the case data are present in this cache level, they are copied into the level 1 cache and into registers, and it takes a hundred of cycles for the memory operation to complete. Otherwise, the processor has to issue a request for the data to external memory and it takes thousands of cycles to fetch a data from the memory into the level 1 cache and the processor

registers. This is an important cost compared to the execution time of instructions other than memory, which is typically under ten cycles.

Under normal circumstances, a write operation transfers data from the registers into the level 1 cache. Since the size of caches increases with their level, caches of a higher level contain more information than caches of lower levels. So, at some point it is necessary to “evict” data from a cache to make room for another data. Data evicted from a given cache are written into the cache of the immediately succeeding level and this cascading process eventually ends up into a write into the external memory. A important characteristic of the cache management is that it tries to maintain in the faster levels data that are likely to be reused in the near future, what is called *temporal locality*. A common policy of cache eviction is to keep the most recently used data in the cache when a choice is to be made.

Additionally, in the definition of our data structure, we will take into account the following detail of the cache mechanism. Data are gathered into small groups when transferred to and from memory, or between caches of different levels. Each group consists of a segment of several memory words that have contiguous addresses. They are named *cache lines* or *cache blocks*. For recent x86 processors, the cache line size is 64 bytes long. This means that a cache line contains 16 floating-point values in single precision or 8 floating-point values in double precision. The important point to note here is that this mechanism implies that the average cost to access data depends on the way they are accessed. In particular, successive accesses to contiguous data is more efficient than accesses in a random order. As a matter of fact, if the first requested data is not present in the first cache level, the cost of a cache miss is amortized by the cache hits of the successive accesses to contiguous data, because the next data are in the cache line that has been copied in level 1 cache. On the contrary, random access to data is likely to entail a cache miss for each access. It is therefore very important to maximize data reuse of a cache line and this is the basic principle of the blocking technique described in Section 6.2.3. Moreover, the hardware is able to detect sequential memory access and to fill in advance level 1 cache with the next data as soon as the memory bus is free. This feature is called *hardware prefetching* and reduces greatly the latency of memory operations in favorable cases.

6.2.2 Tools for parallel programming

Linear algebra algorithms present many opportunities for exploiting parallelism: many computations are independent and the same operations are performed on a great amount of different data. Let us see now how we can express and exploit the parallelism in the computation of interval matrix products.

First, we can express the parallelism at the instruction level by producing an executable code that contains vector instructions. This can be done either by the compiler or by the programmer.

Recent compilers have automatic vectorization capabilities and, in some cases, they can transform a sequential code that is written in a high-level language, like C or Fortran, into a vectorized executable code. Most of the time, this requires the programmer to write the program with particular constructs that the compiler will recognize. It is also possible to give some information to the compiler by annotating the code. These annotations have long been specific to the compiler, but new versions of parallel extensions to C or Fortran, tend to adopt and to standardize them: see, for instance, the pragma `simd` in the last version of the OpenMP standard[3] [Ope13, § 2.8 SIMD Constructs].

The other solution for a program that makes use of vector instructions is to call them explicitly in the source code. This can be done by writing directly in the assembly language, but the source code is not portable between systems since the application binary interface (ABI) and

[3]The OpenMP standard is otherwise more oriented to *thread* level parallelism.

the number of available registers, among other factors, varies greatly, even if the underlying hardware remains the same. The vector instructions for the x86 platforms are however directly available from higher language level as so-called *intrinsic functions* or *intrinsics*. These functions are directly transformed by the compiler into the corresponding vector instruction. The set of intrinsic functions is not normalized as an official extension of the C or Fortran language, but it is a *de facto* standard. The support of intrinsic functions is actually widespread among compilers for x86 targets.

The solution of writing code with intrinsics has two main advantages compared to an assembly version. On the one hand, it gives the programmer a tight control on the vectorized part of the executable code. On the other hand, the burden of the adaptation to the ABI of the target system and the difficult and error-prone process of the register allocation is left to the compiler. We use this technique for expressing parallelism at the instruction level and we show in Chapter 7 that the source code is readable while the execution time is often better than with the automatic vectorization of a plain C code. The main drawbacks of this solution are that the produced source code is limited to x86 compilers and to processors that support the corresponding instruction set, and that the data have alignment constraints.

Second, the computation of interval matrix products can be split into several execution tasks that compute different parts of the result matrix. This is possible because the computation of each component of the result is independent of the computation of the other components.

The first possibility for the assignment of computational tasks to different threads of execution is to use a low level library dedicated to thread management. The POSIX standard [IT13] specifies an application programming interface for threads that is widespread on UNIX systems. The main disadvantage is that the creation, synchronization, and destruction of the threads and the task assignment have to be explicitly written in the source code. Another solution is to use a library for parallel tasks that provides high level constructs for a parallel execution. Intel TBB [Int14b], XKaapi [GFMR13], and Quark [YKD11] are such libraries, for different application fields. These three libraries propose to express task parallelism instead of thread parallelism. Thus, the program has to be decomposed as a collection of tasks that process the data and the run-time library handles the distribution of the tasks to the available threads, taking into account the various dependencies between tasks and trying to balance the workload of the processors. This solution is suitable for high-level operations in linear algebra, like matrix factorizations or solution of system of linear equations, where the computation can be expressed as a sequence of different tasks. In our case, computations in matrix multiplication are very regular, each thread executing the same operations on different data. So, it is easier and more efficient to balance the workload statically.

The solution we chose for the implementation of interval matrix products is to express the task parallelism with the OpenMP extension [Ope13] of the C language. The extension consists principally of a set of annotations (pragma's) in the sequential code that are interpreted by the compiler as thread directives. The same code can be executed sequentially, that is with a unique execution thread on a unique core, or in parallel on several cores. The number of execution threads can be fixed in the code or determined at run-time with the value of special environment variables. Likewise, it is also possible to specify at run-time the cores that can be used to execute the code and the way the tasks are assigned to them. The choice of OpenMP for the implementation is an intermediate solution between the low-level and the high-level libraries for thread parallelism. It gives a tight control on the parallel execution, while thread management and synchronization can be left to the compiler.

6.2.3 Data structures in dense linear algebra

We now describe the data structure chosen for the storage of the matrix components and we motivate this choice.

The hardware constraints are clear: it is important to avoid cache misses, so the needed data for a computation should fit in level 1 cache and they should be reused as much as possible before they are evicted from the cache. It is also important to access memory consecutively, taking advantage of hardware prefetching and to process contiguous data using vector instructions.

Several data structures are used in numerical linear algebra for dense unstructured matrices. The traditional and most common structure, used for example by the BLAS libraries, consists of a simple two dimensional array. In that case, the matrix components are stored in contiguous cells of a Fortran or C array variable [BCD+01, § 2.2 Matrix Storage Scheme]. The order of components storage may be the *Row Major* order, like the native array type in the C language, or *Column Major* order, like the native array type in the Fortran language. Consecutive columns of a single row, in the first case, and consecutive rows of a single column, in the second case, are stored in contiguous cells in memory. Thus, when data of two consecutive rows are accessed with the row major order, the data come from two different cache lines. And scanning a complete column in a matrix in row major order creates a cache miss for each access. Moreover, when the matrix dimension grows, the level 1 cache becomes too small to contain a single row. These considerations explain the inefficiency of naive algorithms with the traditional 2D array structure.

Two techniques take advantage of the higher access speed of memory caches: recursive algorithms and block algorithms. The classical recursive algorithm for matrix multiplication and the Strassen's algorithm have been presented in Section 5.1.2. The input matrices are recursively subdivided up to the point where all input data needed for the product of the sub-matrices fit into the level 1 of memory cache. At this point, the throughput of the memory is no more the limiting factor on the speed of the computation. For the second kind of algorithms, matrices are divided from the start and in a single step into small sub-matrices so that three of them can fit into level 1 cache. Such sub-matrices are named *blocks*, and the corresponding algorithms *block algorithms*. Block algorithms can be classified into two categories depending on the type of data structure used for representing matrices.

- Algorithms of the first category handle matrices as two dimensional arrays, so they are compatible with the BLAS format. So as to fit in the cache and to be stored in a contiguous memory area, the sub-matrices have to be copied on the fly into temporary blocks. There is an extensive literature about the problem of limiting these copies and reusing the blocks best. Let us just cite two prominent articles that consider sequential algorithms for matrix multiplication. From the point of view of the choice of the best block size for a given machine, the cache size depending on the particular processor in use, Whaley proposes a technique for determining automatically the best block size in [WD98]. From the algorithmic point of view, Goto and van de Geijn describe how to deal with the constraints of the memory structure and management and compare several matrix decompositions into blocks of various shapes in [Gv08].

  The parallelization of block algorithms for matrix multiplication stresses even more the cost of data copies in cache. In distributed memory, data copies may involve network communications and are therefore very costly. In shared memory, the memory bus is shared between the processing units of the different hierarchical levels, and it may be a bottleneck if the data transfer from memory to cache is too heavy. One can refer to [MZG+07] for a discussion about the problem of the best decomposition for a multi-threaded program in shared memory context.

- In the second category, the data structure reflects directly the block decomposition. In that case, we give up the compatibility with BLAS and the matrix components are stored either as a sequence of blocks or as a recursive structure of blocks. In the first case, advocated for a long time by Gustavson, the matrices are two dimensional arrays in row-major or column-major order, but instead of being bare matrix components, the elements of such arrays are matrix blocks that are small enough to fit in cache (see [Gus06, Gus08, Gus12]). In the second case, matrices are represented by a recursive structure of blocks. Recursive blocks are themselves composed of sub-blocks, until the level of basic blocks, which are simply two dimensional arrays that fit in cache. Such a recursive structure is used in the FLAME project for multi-threaded linear algebra operations in a shared memory context (see the SuperMatrix description in [QOQOG⁺09]).

Figure 6.3 below summarizes the situation.

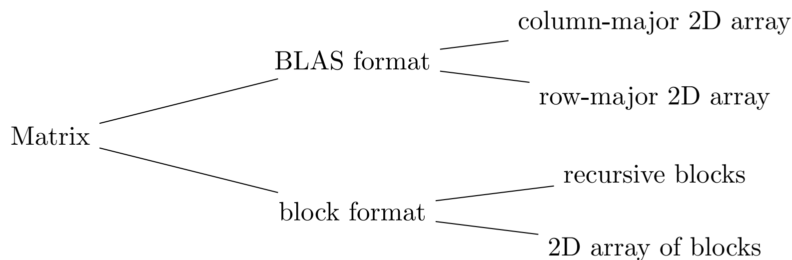

Figure 6.3: Matrix Storage Formats.

Let us now focus on the case of interval matrices. The BLAS Technical Forum Standard proposes a storage format for interval matrices [BCD⁺01, Annex C.4 Interval BLAS]. In the proposed format, a matrix $\boldsymbol{A}$ is stored as a two dimensional arrays of pairs $(\tilde{\imath}, \tilde{s})$ of floating-point numbers, the first element $\tilde{\imath}$ of the pair representing the infimum value $\underline{a_{ij}}$ and the second element $\tilde{s}$ the supremum value $\overline{a_{ij}}$ of the interval component $\boldsymbol{A}_{ij}$. Unfortunately, this format is not well-suited for vector operations because the arithmetic operations are different or the rounding modes are different for the endpoints when one performs an interval arithmetic operation. Lambov [Lam06] proposes to represent intervals with the pair $(-\tilde{\imath}, \tilde{s})$, where the first element of the pair represents the *opposite* of the left endpoint. This way, all arithmetic operations can be executed with rounding towards $+\infty$ only. However, the interval multiplication for this representation involves conditional branches, which lower performance. Moreover, we are dealing here with algorithms that process interval matrices in midpoint-radius representation and the BLAS Technical Forum Standard does not specify any format for this representation.

6.2.4 Our data structure for interval matrix in midpoint-radius representation

Since the midpoint-radius representation of interval matrices is not commonly used (yet), we can choose any data structure that fulfills our needs without breaking any compatibility with existing software. For the general disposal of an interval matrix, we will follow the ideas of Gustavson and we represent the matrix as a sequence of contiguous blocks. The blocks themselves are composed of two basic sub-blocks, one for the midrad sub-matrix components and one for the corresponding radius sub-matrix. If we note s the size of a basic block, $\boldsymbol{M}$ the matrix of midpoints, and $\boldsymbol{R}$ the

matrix of radii, we may see a block as the pair:

$$\left(\begin{pmatrix} m_{11} & \cdots & m_{1s} \\ \vdots & & \vdots \\ m_{s1} & \cdots & m_{ss} \end{pmatrix}, \begin{pmatrix} r_{11} & \cdots & r_{1s} \\ \vdots & & \vdots \\ r_{s1} & \cdots & r_{ss} \end{pmatrix} \right),$$

which is stored contiguously in memory as follows (we choose a row-major order for the storage of basic blocks)

$$m_{11} \cdots m_{1s} m_{21} \cdots m_{2s} \cdots m_{s1} \cdots m_{ss} r_{11} \cdots r_{1s} r_{21} \cdots r_{2s} \cdots r_{s1} \cdots r_{ss}. \tag{6.1}$$

On the one hand, with our choice of general data structure, the data are stored in two dimensional arrays of blocks, and, as noted by Gustavson, to any scalar algorithm corresponds a block algorithm. Where the former performs arithmetic operations on scalar values, the latter performs corresponding operations on matrix blocks. This is the key point of our algorithm implementation as it clearly distinguishes a local part that processes local data, which we will name the *computation kernel*, and a global part that delegates actual computation to kernels. In the case of interval matrix multiplication, the local computation kernel need not to be unique, and different kernels can be launched simultaneously in multiple execution threads as long as they process different blocks. This distinction at the algorithmic level maps successfully to the hierarchical hardware levels of the multiple cores and the enclosing processor. Table 6.2 illustrates this idea.

Category	Local level	Global level
Hardware	core	multi-core processor
Data structure	block	matrix of blocks
Algorithm	block kernel	block algorithm
Implementation	sequential & vectorized	multi-threaded

Table 6.2: Correspondence between Hardware Structure, Data Structure, and Algorithmic Structure.

On the other hand, with this choice of the block structure, we can have in the cache at the same time all data needed for interval matrix multiplications. Indeed, the midpoint-radius algorithms presented in Part I involve computations on the midpoint and radii components of both input matrices. Let us focus on the case of data in double precision, and let us assume that the level 1 of cache can contains S_{L1D} floating-point numbers in double precision. If we want that b basic blocks of the two input matrices and the output matrix fit together into the level 1 of cache, then the block size s must verify

$$s^2 \leq \frac{S_{L1D}}{b}.$$

We examine the case of our testing platform as an example. The capacity of its level 1 caches is 32 KBytes (see Table 6.1), the size of a value in double precision is 8 Bytes, so a level 1 cache can contain simultaneously no more than 4096 values in double precision. If the algorithm needs to process at the same time all the midpoints and radii of the matrix blocks, then $b = 6$ blocks shall fit in cache (2 basic blocks for the midpoint and radii of both input matrices plus 2 basic blocks for the output matrix). The maximum block size in that case is $s = \sqrt{4096/6} \approx 26$. However, in the `MMMul2` algorithm implemented in the next chapters, we only need $b = 4$ basic blocks in the cache at the same time: the two input midpoint blocks, the output midpoint block for their

product and a block for the product of the absolute value of the midpoints. In that case, the maximum block size is $s = \sqrt{4096/4} = 32$. Apart from this upper bound, other constraints apply to the size of the block. They are listed below.

We can now give the details of the actual C implementation of the data structure. Listing 6.1 shows the definition of the corresponding types. The description focuses on matrices in double precision. Similar types are defined for single precision components.

```
/* Blocks are contiguous arrays of square midpoints array followed
   by square radii array in row major order.
   Blocks addresses are assumed to be aligned on cache line start.
*/
typedef float *  smr_block_ptr; /** @< single precision midrad block */
typedef double * dmr_block_ptr; /** @< double precision midrad block */

/** Double precision midrad block matrix. */
typedef struct {
  unsigned int nrow;
  unsigned int ncol;
  dmr_block_ptr blocks;
} __dmr_struct_t;
typedef __dmr_struct_t *dmr_ptr;
```

Listing 6.1: Data Structures for Interval Matrices.

The `dmr_block_ptr` type[4] represents a pointer to an array of double precision values. This values correspond to the elements of the basic blocks for midpoints and radii described above. The size s of the basic blocks is not stored in the structure but it is a global parameter whose value is represented by the symbol `IBLAS_BLOCK_DIM_DOUBLE` (while the symbol `IBLAS_BLOCK_SIZE_DOUBLE` represents the number $2s^2$ of values in the pair of basic blocks). This allows one to experiment several block sizes by changing a single value in the code. The memory area pointed by the `dmr_block_ptr` is then interpreted by the program as the basic blocks stored in the order (6.1).

The `_dmr_struct_t` type represents the data structure for the interval matrix. It contains two fields for the dimension of the matrix and the last field `blocks` points to the memory area where the blocks are stored contiguously in row-major order. The main drawback of the block structure compared to the BLAS format is that the latter allows handling sub-matrices without any copy. In fact, it suffices to provide a pointer in the enclosing matrix, the dimensions of the sub-matrices along with the *leading dimension*, or *stride*, of the enclosing matrix, that is the dimension of rows in row-major order or the dimension of columns in column-major order. This is no more possible with our block format as sub-matrices may not overlap complete blocks. As a consequence, computations on a sub-matrix implies the copy of its data from the original matrix to a new memory area.

Another drawback is that some memory is wasted as the dimension of the matrix may not be a multiple of the dimension of basic blocks. In that case, the value of block elements that do not correspond to matrix components is set to zero (*padding*).

Finally, in order to maximize the data reuse, blocks have to fill complete cache lines. That is, the block size must be a multiple of the cache line size. The processors we are targeting have cache lines that can contain 8 values in double precision, so the block size should be a multiple of eight. The start address of basic blocks also has to be aligned on the start address of cache lines. Storing the sequence of blocks in a contiguous memory area has the fine property that, if

[4] Naming convention uses the following elements: `d` stands for double precision, `mr` for midpoint-radius representation, `_ptr` for pointer, and `_t` for user-defined type.

the first block is correctly aligned with respect to cache lines and if the block size is a multiple of cache line size, then the following blocks are also correctly aligned.

6.3 Experimental protocols

We describe here the choice of the performance metric and the experimental protocols for the measures. In the next two chapters, we propose several implementations of the interval matrix multiplication and we measure the actual execution times of the corresponding executable codes on the same platform. The main purpose of these measures is to determine whether a given implementation meets the criteria for sequential performance and scalability that are discussed in Section 6.1. A secondary objective is to compare several optimization techniques and to select the most efficient. These two objectives will guide our choice of performance metrics.

Two related metrics are commonly used for performance analysis of numerical computations. The first is the ratio of the number of floating-point operations to the execution time measured in seconds. The second is the ratio of the first metric to its maximum possible value on the testing platform, that is to the *peak performance* of the platform. When using these metrics, one does not distinguish different types of floating-point operations and it is assumed that two different operations can be performed as soon as two floating-point units are free. This simplistic model of the micro-architecture is sufficient for classical algorithms of floating-point matrix multiplications on current x86 processors. Actually, the number of additions and of multiplications in such algorithms are almost equal, while such processors can process one addition and one multiplication per cycle. But the situation is different for interval matrix products, where additions outnumber multiplications. For the estimation of a lower bound on the execution time of our algorithms, we need a finer model of the micro-architecture (see Section 7.1 below). This model focusing on the processor pipeline of instructions, the natural unit for the execution time is the processor clock cycle. So, we will estimate, measure, and compare execution times in clock cycles rather than in seconds.

The actual measurement of the execution time of a given executable code can be realized in several ways, but the technique is globally the same: the clock time before running the code under test is noted, the code is executed, the clock time just after completion is noted, and the difference between the dates of start and end of execution is reported. The various possibilities differ in the accuracy of the measures. The simplest manner is to time the whole executable code with an external program, like the `time` command under Linux. This solution does not need to modify the executable code but its accuracy is low, about a millisecond. For a finer resolution, we choose to instrument the code by adding instructions that read more accurate clock time. As a matter of fact, x86 processors support hardware counters that measure various internal events, and among them, a counter for the processor clock with a resolution of about one cycle. These hardware counters are accessible to applications by several means, the most direct access for the count of clock cycles being the `RDTSC`[5] instruction. The counter read by this instruction is set to zero when the processor starts up or is reset, then it is incremented at a constant rate. To quote the relevant part of the Intel documentation [Int14a, § 17.13 Time-stamp counter]:

> Time-stamp counter – Measures clock cycles in which the physical processor is not in deep sleep.

This technique of time measurement comes with many difficulties (see [ZJH09] for an in-depth discussion):

[5]The name `RDTSC` stands for ReaD Time-Stamp Counter.

1. The measured time in cycles may vary if the clock frequency of the processor varies. The reference time for the time-stamp counter is fixed and refers to the frequency of the system bus and not to the processor frequency. So, the measured values may differ for two runs of the same program if, for instance, the processor operates at 2.20 GHz for the first run and at 2.60 GHz for the second.

2. The counter is related to a given core, and counters of different cores are not synchronized. So, if the execution thread is migrated to another core by the thread manager in the course of an execution, the difference between the dates noted at the start and the end of executions no more represents the execution time.

3. On the opposite, some processor models are able to execute two threads simultaneously. Two hardware threads sharing the same execution units, the simultaneous execution of two threads may increase the concurrency for the execution unit accesses. In particular, the pressure on execution units is likely to be high when the threads perform at the same time an identical sequence of operations for different iterations of the same loop. Thus, depending on the scheduling of the threads, the availability of the execution units that are requested for the computation may vary and so does the measured execution time.

4. The operating system may interrupt the code, then switches to another program, and later resumes the code under test. In such a case, the reported measure includes the execution time of the foreign program in addition to the expected one.

5. The number of possible cache misses may vary from run to run with the placement of the data in memory. This means that the measures for the same executable on the same platform may differ from one run to another if the data allocation is not identical in both cases.

6. The process of measurement itself takes some time to complete and this adds some overhead to the measured time.

In order to overcome these difficulties, we will follow the following experimental protocol:

1. We fix the frequency of the testing platform. The processor on the testing platform is able to adjust its clock frequency in response to the workload. This allows several levels of tradeoff between performance and power consumption. First, we turn off the TurboBoost capability, which increases temporarily the clock frequency of a given core when other cores of the same processor are idle. And second, we ask the Linux operating system to use a policy for frequency scaling that favors performance: the clock frequency is set to 2.20 GHz, the highest possible value when all cores are running, and CPUs never enter a sleep state. This eliminates the variability of the clock frequency and ensures a constant ratio between the measured clock cycles and the processor cycles.

2. We assign each thread to a single core. Actually, the migration of OpenMP threads can be disabled by setting the environment variable `OMP_PROC_BIND` to true.

3. We disable the symmetric multi-threading capability (hyperthreading) of the processor from the start. Without hyperthreading, a given core does not process more than one thread at a given clock cycle, but it still can handle several threads in sequence and after a context switch.

4. We experiment on a dedicated platform. Measurements are taken on a platform that runs solely the interval matrix multiplication program. Note that the system activity may

randomly perturb the measure as many “background” programs, which perform system tasks such as network or disk buffering management, are running at the same time as the tested program. However, because of the low priority of these programs, the variation is of minor importance. We compute an average value of several time measures so as to alleviate these perturbations when needed.

5. The code is written so that the number of cache misses is minimized. Given the cost in time of the management of a cache miss, limiting them to their bare minimum is a critical objective for a fast program. This is one of the motivation of the block algorithms presented above.

6. The execution times we are interested in are of order of magnitude of the hundreds of clock cycles at least. Fortunately, the direct use of the `rdtsc` instruction causes an overhead of only 7 cycles[6], which is of the order of magnitude of the experimental variation of the measure.

Lastly, the code that we want to time is structured in two levels. So, we have two categories of measurement protocols.

Small computation kernels that operate on matrix blocks constitute the first group. In the case of block kernels, we measure sequential execution times only. Emptying the cache between successive measures (cold cache) or not (warm cache) does not change much the figures, as it is expected if the cache line reuse is high. The reported measures in the next two chapters are with warm caches. The execution times for this group are small, between hundreds of clock cycles and hundred of thousands of clock cycles. They are therefore unlikely to be interrupted by the operating system (point 4 above). Thus, the timings in Section 7.4 below are typical values selected to have about 5% variation in a small group of experimental measures. These measures do not represent a minimum for the execution time of the block kernels, but the outliers are filtered out.

The second category is related to the measure of sequential and multi-threaded runs of the whole matrix multiplication. In that case, the running time is long enough for operating system to interrupt the computation. So, the measures of execution times contain overhead due to the operating system. Moreover, in the case of a multi-threaded run, the measures also include some extra clock cycles spent by the OpenMP run-time library for thread management. These additions to the computation time are intended, as we want a measure of a typical execution time that could be observed by a user. Alas, by including the costs of all operations that are performed during the suspension of the execution thread, we might also add randomly the execution times of processes not directly related with the computation, like, for instance, network management or logging of system events. For the purpose of yielding representative measures, we use a more elaborate protocol than for block kernel measures. We acquire measures of a single run until we get a subset of measures that do not differ by more than 1%. The size of this subset depends on the dimensions of the input matrices, so as to have a reasonable duration for the complete experiment. For dimensions less than 2,048, the subset contains 11 measures, from 2,048 to 4,096, it contains 6 elements, and only 4 elements between 4,096 and 8,192. The least value of the set is then reported as an average minimum for the execution time.

For multi-threaded runs, we measure, as indicated above, and report the execution time of the master thread, that is the thread that executes the sequential part in addition to its own part of the parallel computation. However, this sequential part is quite reduced in the implementation of `MMMul2` that is detailed in Chapters 7 and 8.

---

[6]The actual value for this overhead was evaluated by measuring the time of an empty loop.

6.4 Conclusion

In this chapter, we set the main objectives for an efficient parallel implementation of the algorithms for interval matrix multiplications. We determined the main obstacles from the hardware as well as from the software sides that we will face and exposed the solutions we have chosen. We also discussed of the difficulties of a reproducible and representative measure of execution times.

It will be shown in the next two chapters how the principles described here are implemented and lead to the realization of an efficient parallel code for the interval matrix multiplication.

Chapter 7

Hardware model and blocking for single core computations

In this chapter, we focus on the parallelism at the instruction level. We detail the implementation of the computation of a block of the matrix product using the `MMMul2` algorithm. Several classical optimizations techniques are applied and we compare experimentally the benefits of manual and automatic optimization in the case of interval matrix products.

The first section introduces a simple model of the micro-architecture in order to determine a lower bound on the execution time. The second section explains how vector instructions can be used in a high level language, like the C language. The code for the block computation of an interval matrix product is discussed in the third section. In the fourth section, we exhibit the limitations of compilers when processing code for interval computations by comparing several automatic optimizations against a manually optimized version. Finally, the last section concludes on the pertinence of our implementation choices, on current compiler limitations, and on the limits of our model for predicting the performance.

7.1 Hardware model for single core performance prediction

Most processors of the x86 family have pipelined, out-of-order superscalar cores with vector computation capabilities. These qualifications mean:

- *pipelined*: a core can process at least one new instruction per clock cycle, regardless of the latencies of previous instructions,
- *out-of-order*: a core can reorder the sequence of instructions, provided that the result is the same as for the original program order, the purpose being to use an instruction order that avoid execution stalls,
- *superscalar*, a core can execute several instructions simultaneously,
- *vector computation*: the same operation can be performed simultaneously on several numerical data.

It is not surprising that the hardware in charge of the pipeline management, which we will name the *micro-architecture* of the core, is very sophisticated. In this section, we focus on a small set

of elements in the core micro-architecture. Our goal is to determine the limiting factor in the computation of interval matrix products by using a simplified model of the core. In addition, we want to derive from the model some lower bounds for the execution times of the computation kernels that are detailed in Section 7.3.

The core micro-architecture of out-of-order x86 processors can be divided in two parts (see [Int13a] for a more complete description).

The first part is mainly in charge of the instruction decoding, branch prediction, and register management. In this part, instructions are processed according to the program order. Hardware elements that interact at this level of instruction processing, like the instruction cache, the decoder, the branch predictor, or the register files are often limiting factors for the program execution. However, as we are dealing here with very regular codes consisting principally in loops that use a small number of instructions and a small number of registers, we will neglect these limitations and we will not go into further details.

The second part consists of the out-of-order execution engine. It has the responsibility to perform the actual computation by doing operations in parallel when possible. So, the concurrent use of a limited number of execution units is likely to be the bottleneck of a numerical computation. In the remainder of this section, we describe the micro-architecture of a Sandy Bridge core giving the minimum details that are useful for a simple performance prediction model.

Figure 7.1 depicts the out-of-order execution engine of a Sandy Bridge processor model, focusing on vector execution units (from [Int13a, § 2.2 Intel Micro-architecture Code Name Sandy Bridge]).

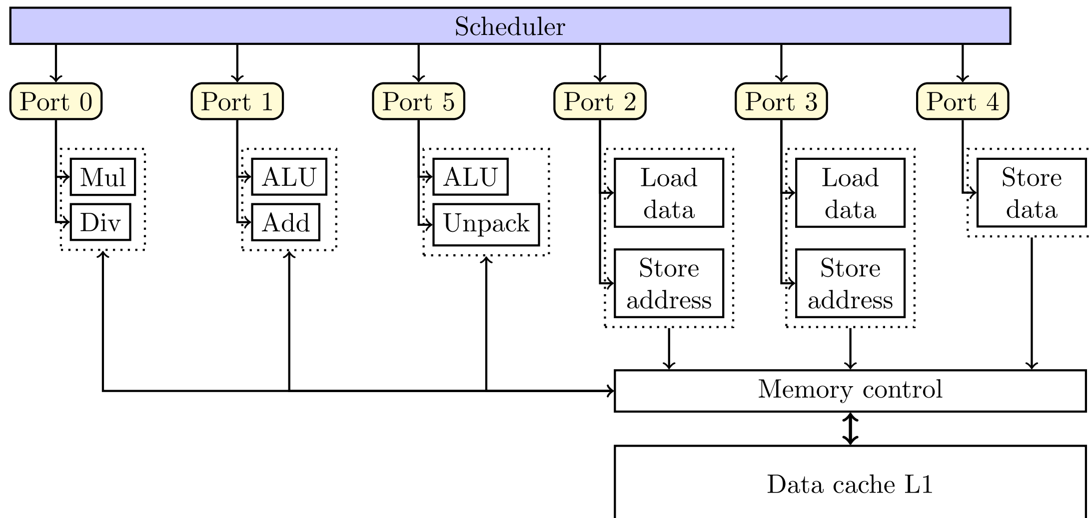

Figure 7.1: Out-of-Order Execution Engine Model (Intel Sandy Bridge Micro-architecture).

After being decoded by the first pipeline part, instructions are reordered by the scheduler and dispatched to an execution unit that can process it. The dispatching is constrained by the disposal of the execution units. Execution units are grouped and a given group is accessible through a corresponding execution port. Execution ports are accessed concurrently and are therefore a possible bottleneck for the program execution. More precisely, a given execution port can accept only one instruction at given clock cycle, but different execution port can accept different instructions at the same clock cycle.

Execution ports are of two kinds.

- Ports number 0, 1, and 5 are entry ports to computational units. We list here only execution units that operate on vectors.
  - Port 0 is linked to the multiplier unit (Mul) and the divider unit (Div).
  - Port 1 is linked to the first arithmetic and logic unit (ALU) and to the adder (Add).
  - Port 5 is linked to the second arithmetic and logic unit and to an execution unit that can shuffle vector components (Unpack).
- Ports number 2, 3, and 4 are entry ports for memory operations.
  - Ports 2 and 3 are identical. Both are linked to a read memory unit (Load data) and an execution unit that performs memory address computations (Store address).
  - Port 4 is linked to the unique write memory unit (Store Data).

In the following, we group the similar ports 1 and 5 into a group of ports, which will be named *Port 15*, and the ports 2 and 3, which have identical functionality, into the group named *Port 23*.

Finally, at the bottom of Figure 7.1, we can see that functional units in charge of memory operations are linked to the memory controller. The purpose of this element is to supply, in time, execution units with the data they need and to store their results. To this end, the controller communicates with the first level of cache that contains data (bottom right), with level 2 of cache (not shown), and with register files (not shown). In the following, we will assume that the data needed for the computation of a matrix block of the result fit in the first level of the cache. The validity of this assumption depends on the size of the block: the cache line reuse has to be maximal and the number of computation sufficiently high to amortize the compulsory cache misses needed for bringing data from the memory to the level 1 of the data cache.

We propose the following method to predict the execution time of numerical algorithms whose data fit in the level 1 of the data cache. First, for a given algorithm implementation, we compute the number of requests for a given execution unit or port by analyzing the individual demand of each instruction in the source code. The most demanded execution unit or port is likely to be the bottleneck for the computation. So, we compute a lower bound on the execution time from the total count of requests for this bottleneck among execution units and execution ports. The execution units and ports that are required for a specific instruction are specified in the following section.

With the method described above, the prediction of the execution time is highly dependent of the architecture of the execution units and ports. Since the time of Pentium Pro, processors of the x86 family have an out-of-order engine with a design similar to that described above. Unfortunately, and to the best of our knowledge, the detailed information about the way execution units are grouped is only available for recent processors (starting from the Sandy Bridge architecture). In the Intel Architectures Optimization Reference Manual [Int13a] for example, the description of the Core 2 micro-architecture, that is the model previous to Sandy Bridge, is not sufficient to determine which unit can process comparison between floating-point vectors. In such a case, the lack of detail prevents us to use the model described here for determining a lower bound on the execution time. We could still establish an inferior bound by simply using the throughput of the vector instructions on the given platform. This quantity represents the maximal number of a given instruction that a core can process at each cycle. It is related to the number of execution units that can perform the operation and it assumes that all needed hardware is free. This last method ignores the fact that the execution units corresponding to the instruction are accessible through a limited number of execution ports. So, a given instruction may be delayed because a different instruction is demanding the same execution port at the same

time. As a consequence, the lower bound that is computed from the instruction throughput is always underestimated with respect to the one derived from our architecture model.

For a better accuracy in the performance prediction, we will focus on the Sandy Bridge architecture in the following sections.

7.2 Vector instruction set: Streaming SIMD Extensions

We give here a short summary about floating-point intrinsics for Streaming SIMD Extensions, see [Int13b, § Intrinsics] for a complete description. We describe only the small subset of functions we use in the implementation of `MMMul2`. The purpose of this description is twofold. First, it helps the reader unfamiliar with this extension to the C language. Second, it is a systematic review of how the listed functions use the execution units described in Section 7.1.

The intrinsic extension defines several new types corresponding to vectors of data. These types differ in two respects. On the one hand, they are related to the native type of the vector elements, which can be an integer or floating point type. On the other hand, they differ in the size of the vectors, which can be either 64-, 128-, or 256-bit long. In the following, our code only uses the `__m128d` type. This type corresponds to two floating-point values in double precision packed in a double quadword, that is a 128-bit memory location or register. Figure 7.2 illustrates the composition of such a vector: the first floating-point value X_1 is stored in the 64 most significant bits of the vector and the second floating-point value X_0 in the 64 least significant bits. We refer below to X_1 as the highest part of the `__m128d` variable and to X_0 as the lowest part. The variables that are of an intrinsic type, like the `__m128d` type, do not support the arithmetic operators `+`, `-`, `*` and `/` of the C language. They can only be used as left-hand side values of an assignment and as parameters of an intrinsic function.

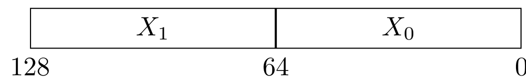

Figure 7.2: `__m128d` Variable.

The intrinsic extension defines a set of intrinsic functions, also named *intrinsics*. Most of them correspond directly to vector instructions (either from the MMX, SSE, or AVX instruction set) and such functions are translated into a single assembly instruction by the compiler. The other intrinsic functions are helper functions, they are translated into only a few assembly instructions. This gives a tight control on the vector instructions that appear in the assembly code generated by the compiler. As a consequence, this eases the performance prediction of a given code written with intrinsics. We list below the requirements in terms of execution units for every intrinsic function that may be used in our implementation.

The intrinsic functions may be grouped as follows[1]:

- memory, assignment and move operations:
  - `_mm_load_pd`. This SSE2 instruction copies a pair of two adjacent floating-point values from the memory to a `__m128d` variable. The memory address must be aligned on a 16-bit boundary.

    The execution of this instruction requires an available load unit and it has a latency of 2 cycles on Core2 and 3 cycles on Sandy Bridge, if no cache miss occurs.

[1] Here, the prefix `_mm_` is reserved to the intrinsic functions, the suffix `_pd` means *packed double* and refers to the `__m128d` type described above

- `_mm_store_pd`. This SSE2 instruction copies the value of a `_m128d` variable to memory. The memory address must be aligned on a 16-bit boundary.

  The execution of this instruction requires an available store unit and store address unit and it has a latency of 3 cycles if no cache miss occurs.
- `_mm_setzero_pd`. This SSE2 instruction sets the `_m128d` variable to zero.

  It requires no execution unit for its execution and has a latency of 1 cycle.
- `_mm_loaddup_pd`. This SSE3 instruction copies a floating-point value from memory in both elements of a `_m128d` vector. The memory address may be unaligned on a 16-bit boundary.

  The execution of this instruction requires an available load unit and it has a latency of 3 cycles on Sandy Bridge if no cache miss occurs.
- `_mm_unpackhi_pd`. This SSE2 instruction copies the highest parts of two `_m128d` values to a third `_m128d` value. Figure 7.3 illustrates this operation.

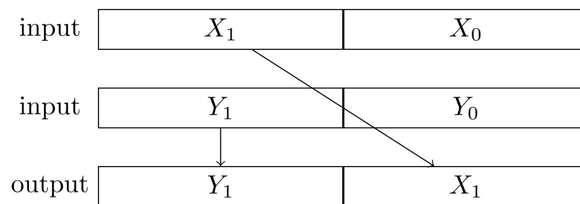

Figure 7.3: `_mm_unpackhi_pd` Intrinsics.

  It requires an available SSE shuffle unit for its execution and has a latency of 1 cycle.
- `_mm_unpacklo_pd`. This SSE2 instruction copies the lowest parts of two `_m128d` values to a third `_m128d` value.

  It requires an available SSE shuffle unit for its execution and has a latency of 1 cycle.
- `_mm_set1_pd`. This helper function sets both elements of the `_m128d` variable to a given floating-point value.

  The documentation [Int13b] indicates that it is a composite of SSE2 instructions, without further detail. Disassembling a simple code using this intrinsic shows that it amounts to a load followed by an unpack instruction. So, it requires one available load unit and one available shuffle unit.

• arithmetic operations: Figure 7.4 illustrates how SSE intrinsics perform an arithmetic or logical operation *op* on two _m128d variables. Note that arithmetic operations take the global rounding mode into account.

- `_mm_add_pd`. This SSE2 instruction adds together the highest parts, resp. the lowest parts, of two `_m128d` values and stores the sum as the highest part, resp. the lowest part, of a `_m128d` variable.

  It requires an available SSE adder unit for its execution and has a latency of 3 cycles.
- `_mm_mul_pd`. This SSE2 instruction multiplies together the highest parts, resp. the lowest parts, of two `_m128d` values and stores the product as the highest part, resp. the lowest part, of a `_m128d` variable.

  It requires an available SSE multiplier unit for its execution and has a latency of 5 cycles.

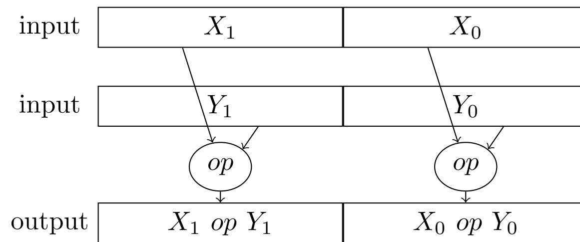

Figure 7.4: Arithmetic and Logic Intrinsics.

- `_mm_div_pd`. This SSE2 instruction divides the highest parts, resp. the lowest parts, of two `_m128d` values and stores the quotient as the highest part, resp. the lowest part, of a `_m128d` variable.

  It requires the SSE division unit for its execution and has a latency up to 20 cycles. In contrast to other SSE execution units, the divider is not pipelined, so a new division can be computed only after the previous one is completed.

- comparison and logical operation:
  - `_mm_cmpgt_pd`. This SSE2 instruction compares together the highest parts, resp. the lowest parts, of two `_m128d` values. If a part of the first parameter is greater than the corresponding part of the second parameter then an all-one bit-mask is stored in the corresponding part of the `_m128d` result, else an all-zero bit-mask is stored.

    It requires an available SSE arithmetic and logic unit (ALU) for its execution and has a latency of 3 cycles.
  - `_mm_or_pd`. This SSE2 instruction performs a bitwise *OR* between the highest parts, resp. the lowest parts, of two `_m128d` values and stores the result as the highest part, resp. the lowest part, of a `_m128d` variable.

    It requires an available SSE ALU for its execution and has a latency of 1 cycle.
  - `_mm_and_pd`. This SSE2 instruction performs a bitwise *AND* between the highest parts, resp. the lowest parts, of two `_m128d` values and stores the result as the highest part, resp. the lowest part, of a `_m128d` variable.

    It requires an available SSE ALU for its execution and has a latency of 1 cycle.
  - `_mm_andnot_pd`. This SSE2 instruction performs a bitwise *AND* between the bitwise negation of the highest part of its first `_m128d` parameter and the highest part of its second parameter. The result is stored in the highest part of a `_m128d` variable. A similar operation is performed between lowest parts.

    It requires an available SSE ALU unit for its execution and has a latency of 1 cycle.

Finally, the intrinsic extension also defines the macro function `_MM_SET_ROUNDING_MODE()` for changing the rounding mode. Its parameter may be one of the symbols `_MM_ROUND_NEAREST` and `_MM_ROUND_UP`, among other possibilities, to set the rounding mode for subsequent SSE operations.

7.3 Block computations

Let us assume that we are given two pointers `a` and `b` to two different s-by-s blocks of interval matrices. We want to compute their product in a third block pointed by `c` using a variant of

Algorithm 9.

The variant we use improves the tightness of the result for a negligible overhead. The differences are as follows. Instead of a single upper bound e on the relative accuracies of the whole block pointed by `a`, we compute a row vector $(e_1 \dots e_s)$ composed of the maximum accuracies of each column of the block. Figure 7.5 illustrates this idea. Similarly, for the relative accuracies

$$\begin{pmatrix} \langle M_{\boldsymbol{A}11}, R_{\boldsymbol{A}11} \rangle & \cdots & \langle M_{\boldsymbol{A}1s}, R_{\boldsymbol{A}1s} \rangle \\ \vdots & & \vdots \\ \langle M_{\boldsymbol{A}s1}, R_{\boldsymbol{A}s1} \rangle & \cdots & \langle M_{\boldsymbol{A}ss}, R_{\boldsymbol{A}ss} \rangle \end{pmatrix} \xrightarrow{\text{max racc}} \begin{pmatrix} e_1 & \cdots & e_s \end{pmatrix}$$

Figure 7.5: Row-Vector of Upper Bounds on Relative Accuracies.

of the block pointed by `b`, we replace the upper bound f by a column vector of upper bounds on each row of the block (see Figure 7.6). Lastly, in the computation of $\widetilde{R_{2ij}}$ at line 5 of Algorithm 9,

$$\begin{pmatrix} \langle M_{\boldsymbol{B}11}, R_{\boldsymbol{B}11} \rangle & \cdots & \langle M_{\boldsymbol{B}1s}, R_{\boldsymbol{B}1s} \rangle \\ \vdots & & \vdots \\ \langle M_{\boldsymbol{B}s1}, R_{\boldsymbol{B}s1} \rangle & \cdots & \langle M_{\boldsymbol{B}ss}, R_{\boldsymbol{B}ss} \rangle \end{pmatrix} \xrightarrow{\text{max racc}} \begin{pmatrix} f_1 \\ \vdots \\ f_s \end{pmatrix}$$

Figure 7.6: Column-Vector of Upper Bounds on Relative Accuracies.

the multiplicative factor[2] of $\widetilde{\Gamma}_{ij}$ is replaced by the sum $e_j + f_i + f_i e_j + \frac{s+2}{2}$. All other operations are the same as in Algorithm 9 and we will use the same function name `MMMul2` for the original algorithm and its variant.

Two different rounding modes are used in `MMMul2`:

- Rounding to nearest for the computation of $\widetilde{M_2}$ and $\widetilde{\Gamma}$,
- Rounding toward $+\infty$ for the computation of $\tilde{e}$, $\tilde{f}$, and $\widetilde{R_2}$.

Moreover, Theorem 3.4 gives a computable bound on the roundoff error for $\widetilde{M_2}$ under the hypothesis that $\widetilde{M_2}$ and $\widetilde{\Gamma}$ are computed in the same order (see Section 5.2). These constraints lead us to divide `MMMul2` in four subroutines[3]:

- `mm_dmidmul2_bb` computes $\widetilde{M_2}$ and $\widetilde{\Gamma}$ at the same time, in the same order, and with rounding to nearest,
- `mm_raccrow_bb` computes an overestimate of the maximum relative accuracy for each row of the block pointed by `a`,

[2] See the erratum in the footnote page 48.

[3] The naming of subroutines is as follows: the prefix `mm_` stands for the midpoint-radius format of both matrices and the suffix `_bb` indicates block computations. The middle parts `dmidmul2` and `dradmul2` point to the midpoint and radius computations, respectively, of the `MMMul2` algorithm, and `raccrow` and `racccol` point to the determination of maximum relative accuracy vectors.

- mm_racccol_bb computes an overestimate of the maximum relative accuracy for each column of the block pointed by b,
- mm_dradmul2_bb computes an overestimate of the radius matrix.

Listing 7.1 shows how these subroutines are called to perform the product of interval block matrices. Some temporary memory is needed: for the product of absolute values and for the two vectors e and f. In the first case, the memory location for the floating-point matrix $\widetilde{\Gamma}$ is pointed by d. In the other cases, pointers e and f point to an array where the corresponding vector is stored. The temporary memory is allocated so as to be aligned on a cache line boundary.

```
int
mm_dmul2_bb (const dmr_block_ptr a, const dmr_block_ptr b,
             dmr_block_ptr c)
{
  double def[_DEF_SIZE] __attribute__ ((aligned(64)));

  /* pointer aliases to ease reading */
  double * d = &def[0];
  double * e = &def[_BLOCK_HALFSIZE];
  double * f = &def[_BLOCK_HALFSIZE + IBLAS_BLOCK_DIM_DOUBLE];

  mm_dmidmul2_bb (a, b, c, d);
  mm_raccrow_bb  (a, e);
  mm_racccol_bb  (b, f);
  mm_dradmul2_bb (d, e, f, c);

  return 0;
}
```

Listing 7.1: The mm_dmul2_bb Kernel.

The following sections detail in turn the actual implementation of the mm_dmidmul2_bb, mm_raccrow_bb, mm_racccol_bb, and mm_dradmul2_bb subroutines. The description focuses on the optimization techniques that are manually applied to the code: loop fusion, vectorization, elimination of conditional branches, and loop unrolling. For the sake of conciseness, some constants and variables that appear in the code are not explained, but their meaning or value can be inferred from their name.

Moreover, note that the code for mm_raccrow_bb and mm_racccol_bb functions studied below accepts only full blocks. Padded blocks require a slightly modified version, not shown here. Indeed, one has to stop the seek of the maximal relative accuracy before the padding zero elements.

For each subroutine below, we first present the source code, then make some comments about it, and finally we try to compute a lower bound on its execution time by considering the bottleneck of the computation.

7.3.1 Combined products of matrices and of their absolute values

In this section, we describe the mm_dmidmul2_bb function, which computes simultaneously the product of two matrix-blocks $M_{\boldsymbol{A}}$ and $M_{\boldsymbol{B}}$ and the product of their absolute values $|M_{\boldsymbol{A}}|$ and $|M_{\boldsymbol{B}}|$. The corresponding code with SSE intrinsics is presented in Listing 7.2. The inputs are

two pointers a and b on the input blocks and the results are output in another block pointed by c and a temporary half-block pointed by d. Because they are modified during the computation, the output blocks are assumed to be different from the input ones.

```
1  static inline void
2  mm_dmidmul2_bb (const dmr_block_ptr restrict a,
3                  const dmr_block_ptr restrict b,
4                  dmr_block_ptr restrict c, double * restrict d)
5  {
6    unsigned int j, k;
7    unsigned int row_offset;
8
9    const uint64_t sign_mask[2] __attribute__((aligned(16)))
10     = {0x7fffffffffffffff, 0x7fffffffffffffff};
11   __m128d smask = _mm_load_pd ((double *)sign_mask);
12
13   /* alias (ease reading) */
14   const unsigned int dim = IBLAS_BLOCK_DIM_DOUBLE;
15
16   _MM_SET_ROUNDING_MODE(_MM_ROUND_NEAREST);
17
18   for (row_offset = 0; row_offset < _BLOCK_HALFSIZE; row_offset += dim) {
19     for (j = 0; j < dim; j+=2) {
20       __m128d dot    = _mm_load_pd (&c[row_offset + j]);
21       __m128d absdot = _mm_setzero_pd ();
22       for (k = 0; k < dim; k++) {
23         __m128d bk;
24         __m128d p1;
25
26         bk = _mm_load_pd (&b[k * _dim + j]);
27         p1 = _mm_mul_pd (bk,
28                          _mm_loaddup_pd (&a[row_offset + k]));
29         dot    = _mm_add_pd (dot, p1);
30         absdot = _mm_add_pd (absdot,
31                              _mm_and_pd (smask, p1));
32       }
33       _mm_store_pd (&c[row_offset + j], dot);
34       _mm_store_pd (&d[row_offset + j], absdot);
35     }
36   }
37 }
```

Listing 7.2: The mm_dmidmul2_bb Function with Intrinsics.

The rounding mode is set at the beginning of the routine (line 16). This ensures that both products are computed with the rounding to nearest mode, as assumed by Theorem 3.4.

The product computation is performed with the classical 3-loop algorithm, where the inner loop correspond to the calculation of two pairs of dot products between a pair of rows of $M_{\boldsymbol{A}}$, resp. $|M_{\boldsymbol{A}}|$, and a pair of columns of $M_{\boldsymbol{B}}$, resp. $|M_{\boldsymbol{B}}|$.

The combination of the computation of $M_{\boldsymbol{A}}M_{\boldsymbol{B}}$ and $|M_{\boldsymbol{A}}||M_{\boldsymbol{B}}|$ saves two memory reads and one multiplication per iteration. More precisely, since $|M_{\boldsymbol{A}ik}M_{\boldsymbol{B}kj}| = |M_{\boldsymbol{A}ik}||M_{\boldsymbol{B}kj}|$, we can use the partial product p1 $= M_{\boldsymbol{A}ik}M_{\boldsymbol{B}kj}$ (lines 27–28) for accumulation in both dot (line 29) and absdot (lines 30–31).

Moreover, the computation of an absolute value can be computed with a bitwise logical operation with a mask. A floating-point value in double precision is stored in memory as a quadword. We use the fact the most significant bit represents the sign of that floating-point value. A logic *AND* between the floating-point value and a 0x7fffffffffffffff bit-mask clears the sign bit (line 31) and does not change other bits, yielding the absolute value of the input. From the point of view of efficiency, this bitwise operation for computing an absolute value has the great advantage to avoid conditional branches in the inner loop. As the sign of the midpoints are random, such conditional branches will be often incorrectly predicted by the hardware, leading to poor performance.

We can now determine which execution unit is a bottleneck in the `mm_dmidmul2_bb` computation. By counting each kind of SSE operation, we estimate the occupancy of the execution units and execution ports per iteration in the inner loop (iteration on k, lines 22–32) and in the middle loop (iteration on j, lines 19–35). The result count is displayed in Table 7.1.

execution unit	mul	add	logic	load	store	execution port	0	1	5	23	4
inner loop	1	2	1	2			1	2	1	2	
middle loop				1	2					3	2

Table 7.1: Use of Execution Units and Execution Ports per Iteration in `mm_dmidmul2_bb`.

Note that, since there is one ALU on each of the ports 1 and 5, the unique *AND* operation in an iteration can use port 5, which is not used by the *ADD* operations.

The total usage count of execution units and ports in the whole `mm_dmidmul2_bb` computation is the product of the enumerations in Table 7.1 by the number of iterations. For square blocks of dimension s, the outer loop is executed s times, the middle loop $s^2/2$ times and the inner loop $s^3/2$ times. Table 7.2 gives the total count of use requests by type of execution unit in `mm_dmidmul2_bb`.

mul	add	logic	load	store	ports	0	1	5	23	4
$\frac{s^3}{2}$	s^3	$\frac{s^3}{2}$	$s^3 + \frac{s^2}{2}$	s^2		$\frac{s^3}{2}$	s^3	$\frac{s^3}{2}$	$s^3 + \frac{3}{2}s^2$	s^2

Table 7.2: Total Number of Use of Execution Units in `mm_dmidmul2_bb`.

The highest number of requests is for the load unit and the group of ports 23. However, Sandy Bridge cores contain two load units that are accessible through two different ports. Consequently, the core can execute two loads per cycle. On the contrary, there is only one addition unit per core. Since $s \geq 2$, the addition unit is the bottleneck among execution units. And each run of `mm_dmidmul2_bb` needs at least s^3 cycles to complete, if we assume that the processor can start an SSE operation each clock cycle, provided that one corresponding execution unit and the port that is attached to it are free.

7.3.2 Determination of the maximum relative accuracy

The computation of the relative accuracies from the midpoint-radius components of an interval matrix requires several divisions. However, floating-point divisions are slow operations on x86: a single division may require 20 cycles on Sandy Bridge architecture and, moreover, the division unit is not pipelined. Reducing the number of division when seeking the maximal relative accuracy is therefore a profitable operation. The following proposition shows how to use this optimization and give valid results.

Proposition 7.1. *Let m, r, m_1 and r_1 be four non-negative numbers.*
If $\mathrm{fl}_\Delta(mr_1) \le \mathrm{fl}_\Delta(m_1 r)$, then $\mathrm{fl}_\Delta(m/r) \le \mathrm{fl}_\Delta\left((1+2\mathrm{u})\, m_1/r_1\right)$.

Proof. By definition of rounding toward $+\infty$, we have $mr_1 \le \mathrm{fl}_\Delta(mr_1)$. Using the bound of Section 3.1, $\mathrm{fl}_\Delta(m_1 r) \le (1+2\mathrm{u})m_1 r$. So, if $\mathrm{fl}_\Delta(mr_1) \le \mathrm{fl}_\Delta(m_1 r)$, then $mr_1 \le (1+2\mathrm{u})m_1 r$ and the proposition is true because rounding toward $+\infty$ is non-decreasing. □

The following example shows that $\mathrm{fl}_\Delta(mr_1) \le \mathrm{fl}_\Delta(m_1 r)$ does not imply $\mathrm{fl}_\Delta(m/r) \le \mathrm{fl}_\Delta(m_1/r_1)$.

Example 7.1. In decimal arithmetic with 1-digit precision, the roundoff unit for rounding toward $+\infty$ is $2\mathrm{u} = 1$. Let $m = r = 3$, $m_1 = 7$, and $r_1 = 10$. Then, $mr_1 = \mathrm{fl}_\Delta(mr_1) = 30$ and $m_1 r = 21$ rounded to $\mathrm{fl}_\Delta(m_1 r) = 30$. While $m/r = \mathrm{fl}_\Delta(m/r) = 1$ and $m_1/r_1 = \mathrm{fl}_\Delta(m_1/r_1) = 0.7$.

So, Proposition 7.1 allows us to compare products instead of quotients in mm_raccrow_bb. The code, shown in Listing 7.3 below, uses this idea.

```
1  static inline void
2  mm_raccrow_bb (const dmr_block_ptr a, double * restrict e)
3  {
4    unsigned int i, j;
5    const uint64_t sign_mask[2] __attribute__((aligned(16)))
6      = {0x7fffffffffffffff, 0x7fffffffffffffff};
7    __m128d smask = _mm_load_pd ((double *)sign_mask);
8    const __m128d rnd_bound =
9      _mm_set1_pd (1 + 2 * IBLAS_UNIT_ROUNDOFF_DOUBLE);
10
11   /* alias (ease reading) */
12   const unsigned int dim = IBLAS_BLOCK_DIM_DOUBLE;
13
14   _MM_SET_ROUNDING_MODE(_MM_ROUND_UP);
15
16   for (j = 0; j < dim; j += 2) {
17     __m128d mid = _mm_and_pd (smask, _mm_load_pd (&a[j]));
18     __m128d rad = _mm_load_pd (&a[_BLOCK_HALFSIZE + j]);
19     for (i = 1; i < dim; i++) {
20       /* compute absolute value.
21          WARNING: change sNaN into qNAN and vice-versa */
22       const __m128d mida = _mm_and_pd (smask, _mm_load_pd (&a[i*dim+j]));
23       const __m128d rada = _mm_load_pd (&a[_BLOCK_HALFSIZE + i*dim+j]);
24       const __m128d cmp_mask = _mm_cmpgt_pd (_mm_mul_pd (rad, mida),
25                                              _mm_mul_pd (mid, rada));
26       rad = _mm_or_pd (_mm_and_pd (cmp_mask, rad),
27                        _mm_andnot_pd (cmp_mask, rada));
28       mid = _mm_or_pd (_mm_and_pd (cmp_mask, mid),
29                        _mm_andnot_pd (cmp_mask, mida));
30     }
31     _mm_store_pd(&e[j],
32                  _mm_mul_pd (rnd_bound, _mm_div_pd (rad, mid)));
33   }
34 }
```

Listing 7.3: The mm_raccrow_bb Function with Intrinsics.

Let us make some comments about the code of Listing 7.3. First, note that the rounding mode for vector operation is set to rounding toward $+\infty$ at line 14, regardless of the rounding

mode in use before the call. At line 22, we compute the absolute value of the midpoint by a logic *AND* with a bit-mask. This branch-free technique is explained above in Section 7.3.1. In order to avoid another conditional branching, we also use a bitwise logical operation to select the component of the row that has the maximal relative accuracy. At lines 24–25, we compare two products in application of Proposition 7.1. The result `cmp_mask` of the comparison is a pair of masks, which can be either all-zeros or all-ones values. At lines 26–29, we select with the value of `cmp_mask` the pair of midpoints and radii that maximizes the relative accuracies, up to a $1 + 2\mathtt{u}$ factor. The correction to this possible underestimation is performed at line 32, where the relative accuracies are actually computed, then multiplied by a $1 + 2\mathtt{u}$ factor according to Proposition 7.1. Note that $1 + 2\mathtt{u}$ can be computed exactly in floating-point number. So, the rounding mode does not play any role and the corresponding constant can be set at lines 8–9, before the change of rounding mode at line 14.

We now determine the bottleneck among processor units. The usage of execution ports and units is reported in Table 7.3. Note that we have put ports 1 and 5, in a single group, named 15, as they are used indifferently by logical operations.

execution unit	mul	div	logic	load	store	execution port	0	15	23	4
inner loop	2		8	2			2	8	2	
outer loop	1	1	1	2	1		2	1	3	1

Table 7.3: Use of Execution Units and Execution Ports per Iteration in `mm_raccrow_bb`.

We remark in Table 7.3 a high count of logical operations, and consequently a strong pressure on the ALUs. However, the ALU is duplicated on Sandy Bridge. So, if we ignore the long chain of data dependence in the inner loop, two logical operations could be started at each clock cycle. For square blocks of dimension s, the outer loop is executed $s/2$ times and, at each iteration of the outer loop, the inner loop is executed s times. Table 7.4 gives the total count of use requests by type of execution unit and execution port when `mm_raccrow_bb` is called.

mul	div	logic	load	store	ports	0	15	23	4
$s^2 + \frac{s}{2}$	$\frac{s}{2}$	$4s^2 + \frac{s}{2}$	$s^2 + s$	$\frac{s}{2}$		$s^2 + s$	$4s^2 + \frac{s}{2}$	$s^2 + \frac{3}{2}s$	$\frac{s}{2}$

Table 7.4: Total Number of Use of Execution Units and Execution Ports in `mm_raccrow_bb`.

If we assume that, at each clock cycle, two logical operations can be started and that the group of execution ports 15 can accept two logical operations, then the counts in Table 7.4 shows that logical units and the group of execution ports 15 are bottlenecks. With the above figures, `mm_raccrow_bb` requires at least $2s^2 + \frac{1}{4}s$ clock cycles per call.

Let us turn now to the computation of a column vector of maximum relative accuracies. For clarity, the code of `mm_racccol_bb` is split in two listings. Listing 7.4 below shows the general structure.

```
1  static inline void
2  mm_racccol_bb (const dmr_block_ptr b, double * restrict f)
3  {
4    unsigned int i, j;
5    const uint64_t sign_mask[2] __attribute__((aligned(16)))
6      = {0x7fffffffffffffff, 0x7fffffffffffffff};
7    __m128d smask = _mm_load_pd ((double *)sign_mask);
8    const __m128d rnd_bound =
9      _mm_set1_pd (1 + 2 * IBLAS_UNIT_ROUNDOFF_DOUBLE);
```

```
10
11    /* aliases (ease reading) */
12    const unsigned int dim = IBLAS_BLOCK_DIM_DOUBLE;
13    double * midb = &b[0];
14    double * radb = &b[_BLOCK_HALFSIZE];
15
16    _MM_SET_ROUNDING_MODE(_MM_ROUND_UP);
17
18    for (i = 0; i < dim; i += 2) {
19      __m128d cmp_mask;
20      __m128d mid0 = _mm_and_pd (smask, _mm_load_pd (midb));
21      __m128d rad0 = _mm_load_pd (radb);
22      __m128d mid1 = _mm_and_pd (smask, _mm_load_pd (midb + dim));
23      __m128d rad1 = _mm_load_pd (radb + dim);
24
25      /*
26        select pairs of midpoint-radius components with the maximum
27        relative accuracies in their row for two consecutive rows.
28      */
29      ...  see code below ...
30
31      /*
32        reorder
33        mid0=(m[i,j],m[i,j+1]),mid1=(m[i+1,j],m[i+1, j+1]) ->
34        mid0=(m[i,j],m[i+1,j]),mid1=(m[i,j+1],m[i+1, j+1])
35        idem rad0, rad1
36      */
37      __m128d tmp = mid0;
38      mid0 = _mm_unpackhi_pd (tmp, mid1);
39      mid1 = _mm_unpacklo_pd (tmp, mid1);
40      tmp = rad0;
41      rad0 = _mm_unpackhi_pd (tmp, rad1);
42      rad1 = _mm_unpacklo_pd (tmp, rad1);
43
44      /* max relative accuracies of one even row and one odd row */
45      cmp_mask = _mm_cmpgt_pd (_mm_mul_pd (rad0, mid1),
46                               _mm_mul_pd (mid0, rad1));
47      rad0 = _mm_or_pd (_mm_and_pd (cmp_mask, rad0),
48                        _mm_andnot_pd (cmp_mask, rad1));
49      mid0 = _mm_or_pd (_mm_and_pd (cmp_mask, mid0),
50                        _mm_andnot_pd (cmp_mask, mid1));
51      _mm_store_pd(&f[i],
52                   _mm_mul_pd (rnd_bound, _mm_div_pd (rad0, mid0)));
53      midb += 2 * dim;
54      radb += 2 * dim;
55    }
56  }
```

Listing 7.4: The `mm_racccol_bb` Function with Intrinsics.

The functions `mm_raccrow_bb` and `mm_racccol_bb` differ in the relative order of the two loops. Function `mm_racccol_bb` computes a column-vector whose components are the maximum relative accuracies of the corresponding matrix row (see Figure 7.6). The inner loop scans two successive rows in the input matrix. After execution of the inner loop, a post-process operation is needed

in order to compute a pair of adjacent components of the output vector. It is done by the code at lines 37–50, which reorders interval components so that midpoint and radius of the sought interval in the even[4] row are in the highest parts of `__m128d` vectors and the midpoint and radius of the sought interval in the odd row is in the lowest parts. Then, the pair of corresponding relative accuracies is computed and stored at lines 51–52.

Listing 7.5 shows the code of the inner loop. The code is similar to the inner loop in Listing 7.3 except the row-wise reading of the matrix components. This similitude gives us the opportunity to show an example of the loop unrolling technique and to compare the complexity of the resulting code (Listing 7.5) with an non-unrolled equivalent (inner loop in Listing 7.3).

```
for (j = 2; j < dim; j += 2) {
  const __m128d midb_even =
    _mm_and_pd (smask, _mm_load_pd (midb + j));
  const __m128d radb_even =
    _mm_load_pd (radb + j);
  const __m128d midb_odd =
    _mm_and_pd (smask, _mm_load_pd (midb + dim + j));
  const __m128d radb_odd =
    _mm_load_pd (radb + dim + j);
  cmp_mask = _mm_cmpgt_pd (_mm_mul_pd (rad0, midb_even),
                           _mm_mul_pd (mid0, radb_even));
  rad0 = _mm_or_pd (_mm_and_pd (cmp_mask, rad0),
                    _mm_andnot_pd (cmp_mask, radb_even));
  mid0 = _mm_or_pd (_mm_and_pd (cmp_mask, mid0),
                    _mm_andnot_pd (cmp_mask, midb_even));
  cmp_mask = _mm_cmpgt_pd (_mm_mul_pd (rad1, midb_odd),
                           _mm_mul_pd (mid1, radb_odd));
  rad1 = _mm_or_pd (_mm_and_pd (cmp_mask, rad1),
                    _mm_andnot_pd (cmp_mask, radb_odd));
  mid1 = _mm_or_pd (_mm_and_pd (cmp_mask, mid1),
                    _mm_andnot_pd (cmp_mask, midb_odd));
}
```

Listing 7.5: The `mm_racccol_bb` Inner Loop.

Let us comment the code in Listing 7.5. The loop is unrolled, which means that it scans two consecutive rows in the same time. In both rows, it selects the interval components with the highest relative accuracy among even columns and store them in `mid0` and `rad0`. Likewise, interval components with the highest relative accuracy among odd columns are stored in `mid1` and `rad1`. After the loop completes, the actual maximum relative accuracies for both rows are computed by selecting the best component from the odd and even columns candidates (what is done by the code at lines 37–52 of Listing 7.4). Compared to a similar non-unrolled loop (for instance, the inner loop in Listing 7.3), the length of the code doubles. This is the reason why, apart from this example, the listings given in Section 7.3 only present non-unrolled loops.

Nowadays, the loop unrolling is a common optimization phase that is supported by most compilers. Yet, manual unrolling of the inner loop may be beneficial to the execution time, as shown by experimental measures in Section 7.4.2 below.

Table 7.5 shows the execution units and execution ports usage of the code of `mm_racccol_bb` under the same hypotheses as for `mm_raccrow_bb`.

For square blocks of dimension s, the outer loop is executed $s/2$ times and, at each iteration of the outer loop, the inner loop is executed $s/2$ times. The total number of requests for each

[4] Here, array indices start from 0, as in the C language.

execution unit	mul	div	logic	unpack	load	store	execution port	0	15	23	4
inner loop	4		16		4			4	16	4	
outer loop	3	1	9	4	4	1		4	13	5	1

Table 7.5: Use of Execution Units and Execution Ports per Iteration in `mm_racccol_bb`.

execution unit and for execution ports is given in Table 7.6.

mul	div	logic	unpack	load	store	ports	0	15	23	4
$s^2 + \frac{3}{2}s$	$\frac{s}{2}$	$4s^2 + \frac{9}{2}s$	$2s$	$s^2 + 2s$	$\frac{s}{2}$		$s^2 + 2s$	$4s^2 + \frac{13}{2}s$	$s^2 + \frac{5}{2}s$	$\frac{s}{2}$

Table 7.6: Total Number of Use of Execution Units and Execution Ports in `mm_racccol_bb`.

It is apparent in Table 7.6 that the group 15 of execution ports is the bottleneck among execution units and ports. If we assume that the instructions are scheduled in such a manner that the group of ports 15 handles 2 instructions per cycle, then the minimum execution time of `mm_racccol_bb` cannot be lower than $2s^2 + \frac{13}{4}s$ clock cycles.

7.3.3 Computation of the radius

After the computation of the product of the absolute value matrices and the determination of the vectors e and f, the function `mm_draddmul2_bb` (see Listing 7.6) computes an overestimate[5] for the radius of the product.

```
static inline void
mm_dradmul2_bb (const double * abs, const double * e, const double * f,
                dmr_block_ptr c)
{
  unsigned int i, j;
  __m128d tmp;
  const __m128d rnd_bound =
    _mm_set1_pd ((IBLAS_BLOCK_DIM_DOUBLE + 2) / 2 * IBLAS_UNIT_ROUNDOFF_DOUBLE);
  const __m128d underflow_guard =
    _mm_set1_pd (IBLAS_REALMIN_DOUBLE);

  const unsigned int dim = IBLAS_BLOCK_DIM_DOUBLE;
  double * restrict radc = &c[_BLOCK_HALFSIZE];

  _MM_SET_ROUNDING_MODE(_MM_ROUND_UP);

  for (i = 0; i < dim; i++) {
    __m128d fi = _mm_loaddup_pd (&f[i]);
    __m128d fi_plus_rnd_bound = _mm_add_pd (fi, rnd_bound);
    for (j = 0; j < dim; j += 2) {
      __m128d ej = _mm_load_pd (&e[j]);
      tmp = _mm_mul_pd (_mm_load_pd (abs),
                        _mm_add_pd (_mm_mul_pd (ej, fi),
                                    _mm_add_pd (ej, fi_plus_rnd_bound))));
      tmp = _mm_add_pd (_mm_add_pd (underflow_guard, tmp),
                        _mm_load_pd (radc));
```

[5] See the erratum in the footnote page 48.

```
        _mm_store_pd (radc , tmp);
        abs += 2;
        radc += 2;
      }
   }
}
```

Listing 7.6: The mm_draddmul2_bb Function with Intrinsics.

Like for the previous functions, we can count the requests for any execution unit in the inner and outer loops of mm_draddmul2_bb. The results are displayed in Table 7.7.

execution unit	mul	add	load	store	execution ports	0	15	23	4
inner loop	2	4	3	1		2	4	4	1
outer loop		1	1				1	1	

Table 7.7: Use of Execution Units and Execution Ports per Iteration in mm_draddmul2_bb.

For square blocks of dimension s, the outer loop is executed s times and, at each iteration of the outer loop, the inner loop is executed $s/2$ times. The total number of requests for execution units and execution ports is given in Table 7.8.

mul	add	load	store	ports	0	15	23	4
s^2	$2s^2+s$	$\frac{3}{2}s^2+s$	$\frac{s^2}{2}$		s^2	$2s^2+s$	$2s^2+s$	$\frac{s^2}{2}$

Table 7.8: Total Number of Use of Execution Units and Execution Ports in mm_draddmul2_bb.

If we assume that the processor can perform two loads per cycle, then the adder unit is the bottleneck. At least $2s^2+s$ cycles are needed to complete a mm_draddmul2_bb computation.

7.4 Execution time of block kernels

In this section, we analyze experimental execution times for the functions presented above. Our main goal is to study, on the one hand, the effects of the compiler optimizations and, on the other hand, the criteria of choice for the block size parameter. All execution times are measured on the Sandy Bridge platform described in Section 6.3.

7.4.1 Compiler optimizations and correctness

First, we discuss the behaviors of the GCC and Intel compilers from the point of view of the respect of the rounding mode.

As explained in Section 5.1.1, optimization phases of GCC, even with the latest 4.9 version, do not take into account changes of the rounding mode. The bug is due to a defective internal representation, thus codes written with intrinsics are also affected, see [gcc11]. Let us see if it is possible to prevent this situation. GCC optimizations are controlled by a large set of compilation options that enable or disable a particular phase and by six global options:

- "-O0", which turns off the optimizer. Even if individual optimization options are set in addition to this option, the compiler performs no optimization at all.
- "-O1" or its alias "-O", "-O2", "-O3", and "-Ofast", which enable increasing levels of optimizations.

- “-Os”, which asks to optimize the size of the executable code.

With this setup, one may envisage two different strategies to prevent GCC from violating the rounding mode change.

A first strategy could be to selectively disable the optimization phases that may incorrectly transform the code, with respect to rounding mode. This would require an intimate knowledge of the GCC’s optimization phases. In addition, the solution would be specific to a given version of the compiler, because optimizers are subject to continuous improvement, which may change the situation. Besides, this strategy is unfeasible since it is not possible to turn off some optimization. For instance, if we compile the code of Listing 5.1 with the “-O1” option and all possible options that disable individual optimization, the produced assembly code is still subject to the bug described in Section 5.1.1.

In a second possible approach, one may wonder if it is relevant to compile without optimization. Indeed, the code that is detailed in Section 7.3 is already vectorized. Table 7.9 below presents the measure of execution times, in cycles, for optimized but unsafe codes and for the safe unoptimized code (compiled with “-O0”). Both are compiled with the version 4.6.3 of the GCC compiler.

compilation option	-O3	-O2	-O1	-O0
execution time	47639	48227	49794	519151

Table 7.9: Function `mm_dmul2_bb` compiled by GCC with and without optimization – Execution Times (in cycles).

Not optimizing the code leads to a dramatic performance drop, the time is more than ten times larger. The conclusion is clear: as long as the bug described in Section 5.1.1 is not fixed, GCC should be excluded when optimizing codes that modify the rounding mode.

The case of the Intel compiler (ICC) is simpler. The Intel documentation specifies that the compiler can correctly handle code where the rounding mode changes if the “-fp-model strict” option is set. Actually, we verified that ICC compiles code in Listing 5.1 in a correct assembly code. We examine below the cost of the “-fp-model strict” option in terms of the performance of the executable code. Table 7.10 shows the execution times for the code compiled with the “-O3” option, which turns on the highest global level of optimization, with “-xSSE3 -vec”, which enables the use of SSE instructions, and with and without the “-fp-model strict” option. The version of the compiler is 13.1.0.

compilation option	execution time
-O3 -xSSE3 -vec	47428
-O3 -xSSE3 -vec -fp-model strict	49675

Table 7.10: Function `mm_dmul2_bb` optimized by ICC with and without safe option – Execution Times (in cycles).

Here, the slowdown is tolerable: the execution time is increased by about 5%. Thus, restricting the floating-point optimization to transformations compliant with the C standard is an acceptable solution.

7.4.2 Automatic and manual unrolling

Unrolling loop is a common optimization technique that is supported by most compilers. The unrolling operation consists in computing several iterations of a loop in the same time. This

gives the compiler more opportunities to find an effective scheduling of operations. Listing 7.5 shows an example of a 2-fold unrolling in the inner loop of `mm_racccol_bb`.

In this section, we compare the execution times of two versions of the subroutines in the `mm_dmul2_bb` kernel. The first version corresponds to the code given in the previous section compiled with the Intel compiler and the compilation options "-O3 -fp-model strict", where the "-O3" option turns on loop unrolling. Manually unrolled subroutines make up the second version: a 4-fold unrolling for the inner loop of `mm_dmidmul2_bb`, a 2-fold unrolling for the inner loops of `mm_raccrow_bb` and `mm_dradmul2_bb`. In both cases, the code of `mm_racccol_bb` is the same, with the inner loop being unrolled twice as in Listing 7.5. The measured execution times corresponding to both versions are displayed in Table 7.11.

	unrolling	
	automatic	manual
`mm_dmidmul2_bb`	52517	37477
`mm_raccrow_bb`	7749	5457
`mm_racccol_bb`	5913	5909
`mm_dradmul2_bb`	2785	2561
`mm_dmul2_bb`	69673	49719

Table 7.11: Automatic versus Manual Unrolling – Execution Times (in cycles).

As can been seen in the measured timings given in Table 7.11, the automatically unrolled version is 40% slower than the manually unrolled version. So, from the point of view of efficiency, manual unrolling is a valuable transformation of the code. In the following sections, the code for `mm_dmul2_bb` is the manually unrolled version that is described above.

7.4.3 Block size

We now analyze the effect of the block size on the performance. In Section 7.3, we stated lower bounds on the execution times for the subroutines of `mm_dmul2_bb`, using a model of occupancy of execution units. We compare here these lower bounds with actual measures of execution time.

Table 7.12 gathers the lower bounds determined in Section 7.3.

routine	theoretical minimum	bottleneck
`mm_dmidmul2_bb`	s^3	addition unit
`mm_raccrow_bb`	$2s^2 + \frac{1}{4}s$	ports 15
`mm_racccol_bb`	$2s^2 + \frac{13}{4}s$	ports 15
`mm_dradmul2_bb`	$2s^2 + s$	addition unit
`mm_dmul2_bb`	$s^3 + 6s^2 + \frac{9}{2}s$	

Table 7.12: Minimum Execution Times (in Cycles) for Blocks of Size s According to Model – Sandy Bridge.

In Table 7.13, we compare the theoretical lower bounds with the experimental measures of execution times for increasing sizes of blocks: 16×16 blocks (left), 32×32 blocks (middle), and 64×64 blocks (right). We chose multiples of a cache line size for the value of the block size s so that all data in the cache line are used in the computation. For each block size, the left-hand side column contains the value T_{model} predicted by the model with the formulas in Table 7.12, and the right-hand side column includes executions times $T_{measured}$ measured on a Sandy Bridge

core. In all cases, the code was compiled with ICC version 13.1.0 with compilation options set to "-O3 -fp-model strict -xSSE3 -vec". The number in parentheses is the ratio $T_{model}/T_{measured}$.

	block size					
	$s = 16$		$s = 32$		$s = 64$	
routine	model	measured	model	measured	model	measured
`mm_dmidmul2_bb`	4096	4997 (82%)	32768	37589 (88%)	262144	299785 (88%)
`raccrow_bb`	516	1505 (35%)	2056	5461 (38%)	8208	21085 (39%)
`mm_racccol_bb`	564	1653 (35%)	2152	5909 (37%)	8400	23397 (36%)
`mm_dradmul2_bb`	528	825 (64%)	2080	2589 (81%)	8256	10293 (81%)
`mm_dmul2_bb`	5704	8116 (71%)	39056	49700 (79%)	287008	346755 (83%)

Table 7.13: Theoretical and Experimental Execution Times (in cycles) – Sandy Bridge.

We will consider $T_{model}/T_{measured}$ as a performance ratio. Effectively, if N is the number of instructions in `mm_mul2_bb` and T its execution time in clock cycles, then $IPC = N/T$ is the average number of instructions executed in a clock cycle, which is a measure of performance. The ratio $IPC_{measured}/IPC_{model}$ indicates the relative performance of the implementation compared to the theoretical one and corresponds to numbers in parentheses in Table 7.13.

We can observe the following phenomena. First, the ratio of performance varies greatly between different routines. The routines `mm_dmidmul2_bb` and `mm_dradmul2_bb` tend to have execution times close to the minimum predicted by the model. On the contrary, the measured execution times for `raccrow_bb` and `mm_racccol_bb` are about three times larger than the predicted minimum. Two explanations are possible: either these two routines are poorly implemented or the model is not adapted to this kind of computation. The latter reason is more plausible because the model does not take into account data dependence and latencies of operations. Yet, `raccrow_bb` and `mm_racccol_bb` contain long chains of dependent computations and divisions, which have large latency. However, the experimental measures show that is possible to get more than 75% of the maximal performance that is theoretically possible for the whole function `mm_dmul2_bb` (last row in Table 7.13). As we have reached our goal regarding the sequential performance for a block computation, we will not try to improve neither the model nor the implementation.

The second observation from the data of Table 7.13 is that the performance increases with the block size s. The only exception is the `mm_racccol_bb` routine with $s = 64$, which is a little bit less efficient than with $s = 32$. Nonetheless, the theoretical minimum seems underestimated for the routines that compute minimal relative accuracies, as we have seen above. What improves the most the overall performance of `mm_dmul2_bb` when s grows is the conjunction of two factors. First, the individual performance of `mm_dmidmul2_bb` is high, above 80% regardless the value of s. Second, the cost of `mm_dmidmul2_bb` becomes more and more dominant in the computation as s grows, since it is cubic in s while the cost of other subroutines is only quadratic in s, according to the model.

Processors prior to Sandy Bridge, like Core 2 processors or some AMD contemporaries, have a level 1 data cache with a capacity of at least 32 KBytes, but their level 2 cache is shared among cores. So, despite the fact that the efficiency is higher for Sandy Bridge when $s = 64$, the performance may be more portable across processors with a smaller size and we will chose $s = 32$ as the value of the block size in further experimental measures. Effectively, the latter value is such that the four blocks read or written by `mm_dmidmul2_bb` just fit into the first level of data cache: 4 blocks $\times$ 32 rows $\times$ 32 columns $\times$ 8 bytes per double = 32.768 bytes. At the same time, our requirement of a performance above 75% of a theoretical maximum is fulfilled with $s = 32$.

7.4.4 Automatic and manual vectorization

Current compilers for x86 platforms are able to vectorize simple loops. With the appropriate compiler options, some optimization phases transform a code acting on scalar values into an executable code using vector instructions, whenever the transformation is possible. What deters automatic vectorization, among other factors, is the presence of complicated data dependencies between iterations, conditional branches in the loop, possibility of aliased pointers (two pointers that reference the same location in memory) or uncountable loops (for instance, a `while` loop whose exit condition cannot be predicted from the value of the loop index).

Listing 7.7 presents an excerpt of the implementation of `mm_dmul2_bb` in plain C, without the use of intrinsics. In the code of the `mm_dmidmul2_bb` subroutine presented here, the number of iterations of each loop is constant and known at compile-time. Different pointers always reference different regions in memory, and the `restrict` keyword informs the compiler of this fact. There are simple data dependencies between iterations in the inner loop, and none in the outer loop. In this example, the inner loop contains a conditional branch for the absolute value computation. The absolute value in the inner loop (line 26) could be computed by a logic operation with a bit-mask and a second version of the implementation of `mm_dmidmul2_bb` has been written in plain C without conditional branch (not shown here).

```
1  static inline void
2  mm_dmidmul2_bb (const dmr_block_ptr a, const dmr_block_ptr b,
3                  dmr_block_ptr c, double * restrict d)
4  {
5  #pragma STDC FENV_ACCESS ON
6    unsigned int i, j, k;
7
8    /* aliases (ease reading) */
9    const unsigned int dim = IBLAS_BLOCK_DIM_DOUBLE;
10   const double * restrict mida = (double*) &a[0];
11   const double * restrict midb = (double*) &b[0];
12   double * restrict midc = (double*) &c[0];
13
14   fesetround (FE_TONEAREST);
15   for (i = 0; i < dim * dim; i++) {
16       d[i] = 0.0;
17   }
18   for (i = 0; i < dim; i++) {
19     for (j = 0; j < dim; j++) {
20       double dot = 0;
21       double absdot = 0;
22       for (k = 0; k < dim; k++) {
23         double p;
24         p = mida[i * dim + k] * midb[k * dim + j];
25         dot += p;
26         absdot += p < 0 ? -p : p;
27       }
28       midc[i * dim + j] += dot;
29       d[i * dim + j] += absdot;
30     }
31   }
32 }
```

Listing 7.7: The `mm_dmidmul2_bb` Function in plain C.

The other subroutines mm_raccrow_bb, mm_racccol_bb, and mm_dradmul2_bb of the mm_dmul2_bb function have been also written without the use of intrinsics. We then have three implementation versions of mm_dmul2_bb: in plain C with conditional branches, in plain C with bitwise logical operations, and with intrinsics. Next, the three versions have been compiled with different compilers or different versions of a compiler, asking to vectorize the code.

As discussed in Section 7.4.1, the code produced with GCC is not safe because the rounding mode is changed. However, the measures of Table 7.10 demonstrate that a compiler can produce a correct code for a small performance loss. This inspires us with hope that GCC could be fixed so that the efficiency of the correct code would be similar to the current optimized, and unsafe, code. Keeping in mind that the produced executable code is not safe, we will use GCC in the following measures in order to compare the vectorization capability of ICC and GCC.

Table 7.14 shows the execution times on the Sandy Bridge platform. The second column presents the execution times when the code is compiled with the Intel compiler, specifically the version 13.1.0 with compiling options "-std=c99 -O3 -xSSE3". The timings for the same code compiled with two versions of GCC and executed on the same platform are in the last two columns. The third column relates to version 4.7.2 and the fourth to version 4.6.3. The compiling options "-std=c99 -O3 -msse3 -mfpmath=sse -frounding-math" are used in both cases. The fourth line, labeled *manual vectorization*, refers to the version written with intrinsics.

mm_dmul2_bb	icc 13.1.0	GCC 4.7.2	GCC 4.6.3
plain C with branches	117447	317723	267867
plain C with bit-masks	117345	155581	154574
manual vectorization	47433	47608	47627

Table 7.14: Automatic and Manual Vectorization: Execution Times (in cycles).

The experimental measures displayed in Table 7.14 are instructive. Firstly, compiler optimizers show a large variability in efficiency. A newest version of a compiler being sometimes less efficient than an older one, as it is the case for the compilation of the plain C version with conditional branches by GCC 4.7.2 and GCC 4.6.3. Secondly, concerning the automatic vectorization of the different plain C versions, the presence of conditional branches has no measurable effect on the Intel compiler, while it perturbs GCC compilers, which produce a code that is about twice less efficient than in the case of bit-masks. Lastly, and more importantly, the code that has been manually vectorized shows the best execution times and the timings are independent from the compiler.

As a conclusion, these measures confirm the validity of our choice to write the computing kernels using SSE intrinsics. Actually, the manual vectorization fulfills two of our implementation objectives that are not satisfied by the auto-vectorized versions:

- Sequential performance. As it is apparent in Table 7.14, compilers have some difficulty to vectorize code for interval arithmetic and the most efficient executable code comes from the version written with intrinsics.

- Portability. Execution times for the code that is manually vectorized are small and remarkably constant across compilers.

In addition, the performance is predictable because the source code is close to the executable code, as shown in Section 7.4.3.

Unfortunately, the goal of the utmost importance, namely correctness, rules out the use of the GCC compiler for codes that modify the rounding mode.

7.5 Conclusion

This chapter demonstrates the advantages of implementing block kernels directly with intrinsic functions and with an explicit level of loop unrolling. This strategy avoids compiler limitations in auto-vectorization and loop unrolling.

As an interesting by-product, implementing algorithms with a low level language reflects more closely the executable code and allows one to better predict performance. The simple model presented in this chapter gives accurate predictions of execution times, within 30%, for the whole block kernel function. One limitation of the model comes from the fact that it requires a detailed knowledge of the micro-architecture. The relevant information is easily available for recent processors, starting from the Sandy Bridge model, but harder to find for previous cores with out-of-order engine. Besides, the model also emphasizes the effect of the size of the block on the overall performance. The criteria for our choice have been given, but, admittedly, the chosen value is only relevant to the underlying architecture we had considered. As an improvement towards more portability of the performance across hardware platforms, the block size could be determined at compile-time by testing several block sizes and selecting the one with the best performance, in the spirit of the ATLAS library [WPD00].

Finally, note that the block kernel `mm_dmidmul2_bb`, described in Section 7.3.1, can be reused for the other interval matrix multiplications. Indeed, both `MMMul3` and `MMMul5` involve the computation of products of matrices and products of their absolutes values.

Chapter 8

Multi-core and multi-threading

The previous chapter describes an implementation of the computation kernels used in the algorithm for interval matrix multiplication MMMul2. It is shown there that this implementation reaches the first two goals that were assigned in Chapter 6: *correctness* and *sequential performance*. The main purpose of this chapter is to study the behavior of the implementation of the whole MMMul2 in a multi-threaded execution and to check that *scalability*, the last measurable goal, is attained.

The chapter is organized as follows. Section 8.1 details how the data parallelism in the loops of MMMul2 is exploited at the thread level. The variations of the measured execution times when the matrix dimension increases are analyzed in Section 8.2 for sequential executions and in Section 8.3 for multi-threaded runs on a multi-core multi-processor platform. We give in the last section our general conclusions for the second part of this document, which deals with parallel implementation of interval matrix multiplications.

8.1 Multi-threaded implementations of block algorithms

Listing 8.1 shows the implementation of the matrix product algorithm. The parameters are pointers to the input block matrices for a and b and a pointer to the product matrix for c. We choose to implement the classical triple nested loops algorithm for the matrix multiplication. Instead of iterating over matrix components, we iterate over matrix blocks. The outer loop iterates over the rows of the left factor, the middle loop over the columns of the right factor, and the inner loop over the common dimension. This order has the advantage that the written block of c does not change in the inner loop, increasing the probability that this block remains in the first levels of cache. The functions iblas_dmr_get_block at lines 9, 11, and 12 return a pointer to the start of a matrix block. The returned block is the one that contains the element of the matrix pointed by the first parameter and whose row index and column index are given by the second and third parameter, respectively.

```
1  int
2  mul_mmd_2000 (const dmr_ptr a, const dmr_ptr b, dmr_ptr c)
3  {
4    unsigned int i, j, k;
5
6  #pragma omp parallel for private(i, j, k) schedule(static)
7    for (i = 0; i < a->nrow; i += IBLAS_BLOCK_DIM_DOUBLE) {
8      for (j = 0; j < b->ncol; j += IBLAS_BLOCK_DIM_DOUBLE) {
```

```
 9        const dmr_block_ptr cij = iblas_dmr_get_block (c, i, j);
10        for (k = 0; k < a->ncol; k += IBLAS_BLOCK_DIM_DOUBLE) {
11          dmr_block_ptr aik = iblas_dmr_get_block (a, i, k);
12          dmr_block_ptr bkj = iblas_dmr_get_block (b, k, j);
13          mm_dmul2_bb (aik, bkj, cij);
14        }
15      }
16    }
17
18    return 0;
19  }
```

Listing 8.1: The mul_mmd_2000 Function with Outer Iterations on Rows.

Let us explain now how the outer loop is parallelized with OpenMP (for more details, see [Ope13]).

The annotation `#pragma omp parallel for` at line 6 indicates that the initial execution thread is entering a new parallel region, that it should create a new team of threads (what is named a *fork*), and that the iterations of the following loop should be executed by the threads of the team. This execution is parallel if the team is not reduced to the single initial thread. The number of threads that belong to the new team may be controlled by several means. In the following, we will simply use the `OMP_NUM_THREADS` environment variable to set the number of threads in a team.

The clause `private(i, j, k)` specifies that the variables `i`, `j`, and `k` correspond to different variables in different threads. The other variables are either shared by the threads of the team if they are declared before the parallel region, so are `a`, `b`, and `c`, or private to each thread if they are declared into the parallel region, so are `cij`, `aik`, and `bkj`. Threads may modify the same memory location concurrently if they access it through shared variables. On the contrary, private variables allow the programmer to use the same name for different variables, each thread having a different interpretation of this name. For the code displayed in Listing 8.1 in particular, the threads iterate over different blocks, so they do not actually modify the same memory cells, but they write into the block assigned to them through the same pointer name, here `cij`. This simplifies the code and, in addition, avoids unnecessary copies.

The clause `schedule(static)` determines how the iterations are distributed across threads. With this instruction, successive iterations are gathered in groups that have approximately the same size. Then, one iteration group is assigned to each thread. Other schedules are possible, but this particular one is appropriate when loops perform the same amount of work.

Finally, there is an implicit barrier at the end of an OpenMP parallel construct, in this case at the end of the outer loop line 16. This means that the threads of a team have to wait for the completion of the parallel tasks before the initial thread can proceed (what is named a *join*).

Note that no change of the rounding mode occurs at this level. The rounding mode management is entirely delegated to the block level. This solves the possible problem of rounding mode violation by the run-time library for thread management, as was discussed in Section 5.1.2.

8.2 Sequential execution time

The previous chapter demonstrates that block kernels respect our implementations goals. Let us see now that the multiplication of complete matrices reaches an acceptable level of performance.

Figure 8.1 presents the measures of the sequential execution times for products of square matrices of increasing dimensions. The measured implementation of `MMMul2` is the `mul_mmd_2000`

function presented in Listing 8.1. At the top of the figure, the execution times of our implementation compiled with ICC 13.1.0 are compared to the ones of the `dgemm` function of the MKL, version 11.0.2. The dimension of square interval matrices for `MMMul2` or square floating-point matrices for `dgemm` is displayed along the x-axis. All matrix components are in double precision, and so is the working precision of the computation. The execution times in processor clock cycles are displayed in logarithmic scale along the y-axis. For a fair comparison, we set the environment variable `MKL_CBWR` to `SSE3`, so that `dgemm` uses SSE instructions. The measured times for the `dgemm` version using AVX instructions are not presented here, they show a two-fold speed-up compared to the SSE version, as expected. The ratio of the two execution times for the same matrix dimension is displayed at the bottom of Figure 8.1.

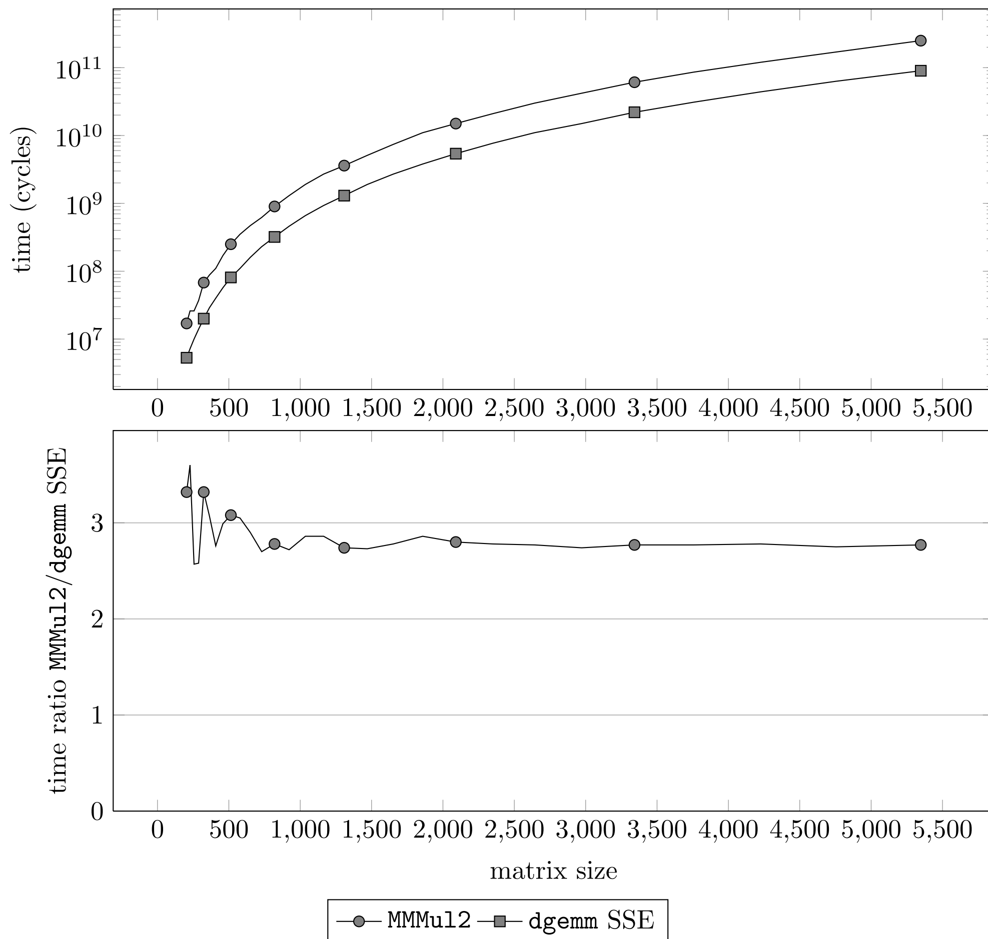

Figure 8.1: Execution time – Sequential.

Apart from very small matrices, the behaviors of the two products are remarkably similar. The timings for `MMMul2` tend to be about 2.8 times larger than the `dgemm` timings. We could have been expecting an overhead of roughly 100% since interval matrix multiplication processes twice the amount of data of a floating-point matrix product. Actually, this last estimation is too optimistic. The simple model presented in the previous chapter can help us analyze more

precisely the minimum overhead, as shown in the following paragraph.

Let us consider blocks of size s. On the one hand, the computation of a product of non interval blocks involves approximately s^3 additions and s^3 multiplications. Thus, it needs a total of $N = 2s^3$ floating-point operations. The Sandy Bridge architecture being able to issue one sum and one product of SIMD vectors per cycle and the SSE vector being made of 2 floating-point values in double precision, the execution time in cycles for the block product is

$$T_{dgemm} = \frac{s^3}{2}.$$

On the other hand, the model for `MMMul2` (see Table 7.12) gives a minimum time of

$$T_{MMMul2} = s^3 + 6s^2 + \frac{9}{2}s.$$

Then, the ratio of theoretical execution times is at least $2 + \frac{12}{s} + \frac{9}{s^2}$. Therefore, for a block size $s = 32$, the minimal execution time for `MMMul2` is at least $2.34\, T_{dgemm}$. Considering the limits of the model, which have been discussed in Section 7.4.3, the measured factor of 2.8 for large matrices seems acceptable.

8.3 Measure of execution time for multi-threaded runs

In this section, we present the timings for the multi-threaded executions of `mul_mmd_2000` when the dimension of the square matrices increases. As said in Section 6.3, the migration of threads between cores is disabled by setting the environment variable `OMP_PROC_BIND` to true. So, in our experiments, each thread is “pinned” to a single core and a given core executes no more than one thread.

Moreover, the experiments are conducted in such a way that the measured times reflect the use of the cores of a machine with an increasing number of octo-core processors. Precisely, the first eight threads are assigned to the cores of the first processor (Socket P#0 in Figure 6.2). Then, for experiments with more than eight threads, the next eight ones are assigned to the cores of the second processor (Socket P#1), the seventeenth to twenty-fourth threads are assigned to the cores of the third processor (Socket P#2), and the last eight ones to the cores of the fourth processor (Socket P#3). We perform this assignment of threads by using the Intel OpenMP library and by setting the environment variable `KMP_AFFINITY` to “compact”.

On the contrary, no attempt is made to control the memory allocation for the matrices. Indeed, each processor is linked to a memory module with a capacity of 96 GBytes. This means that a single memory module may contain three square interval matrices in double precision that have a dimension up to 44721, because 6 floating-point matrices $\times$ 8 Bytes per double $\times 44721^2$ is less than $96 \cdot 2^{30}$. In our experiments, we restrict ourselves to much smaller dimensions, so as to keep execution times in the limit of one half day for the whole set of measures. As a consequence, components of all matrices are likely to be stored together in the same memory module, presumably in the first memory module (M0 in Figure 6.1). This disposition favors the threads that are executed on some core of the first processor and is less favorable to the threads running on processor P#2 (see discussion on non-uniform memory access in Section 6.2.1).

As for sequential measures, the measured times for the `dgemm` function are related to the SSE version.

8.3.1 Timings for 8 threads

The first set of measures aims at characterizing the behavior of the code when the execution threads are dispatched to all cores of a single processor. Figure 8.2 summarizes the measures of execution times with eight threads in a presentation similar to Figure 8.1.

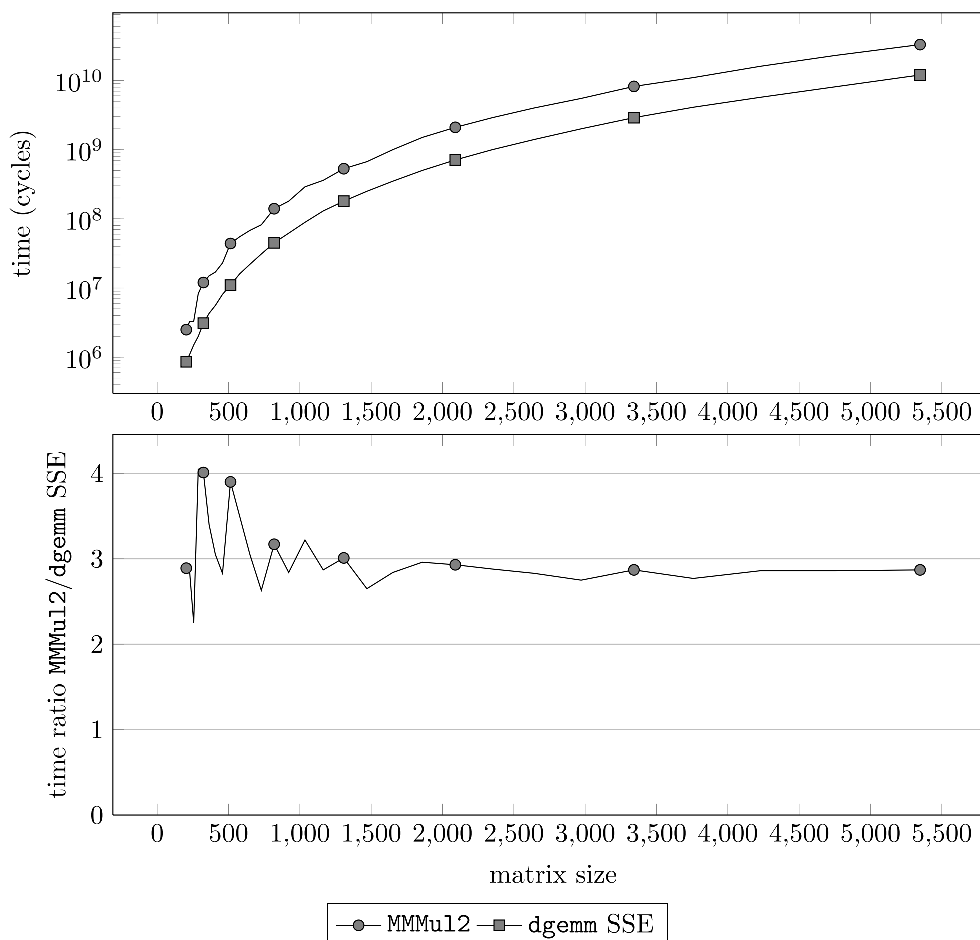

Figure 8.2: Execution time – 8 threads.

The variations of the execution time with the dimension of the matrices are alike for `MMMul2` and `dgemm`. The ratio of the execution times of `MMMul2` and `dgemm` is always between 2 and a little bit more than 4, the maximum ratio in this set of measures being 4.05. It clearly tends to a value around 2.9 when the matrix dimension grows.

8.3.2 Timings for 32 threads

The second set of measures exposes the behavior of the MKL `dgemm` and of our implementation of interval matrix product when all the processing power of the test machine is used. The measured execution times for `MMMul2` and `dgemm` with 32 threads are displayed in Figure 8.3.

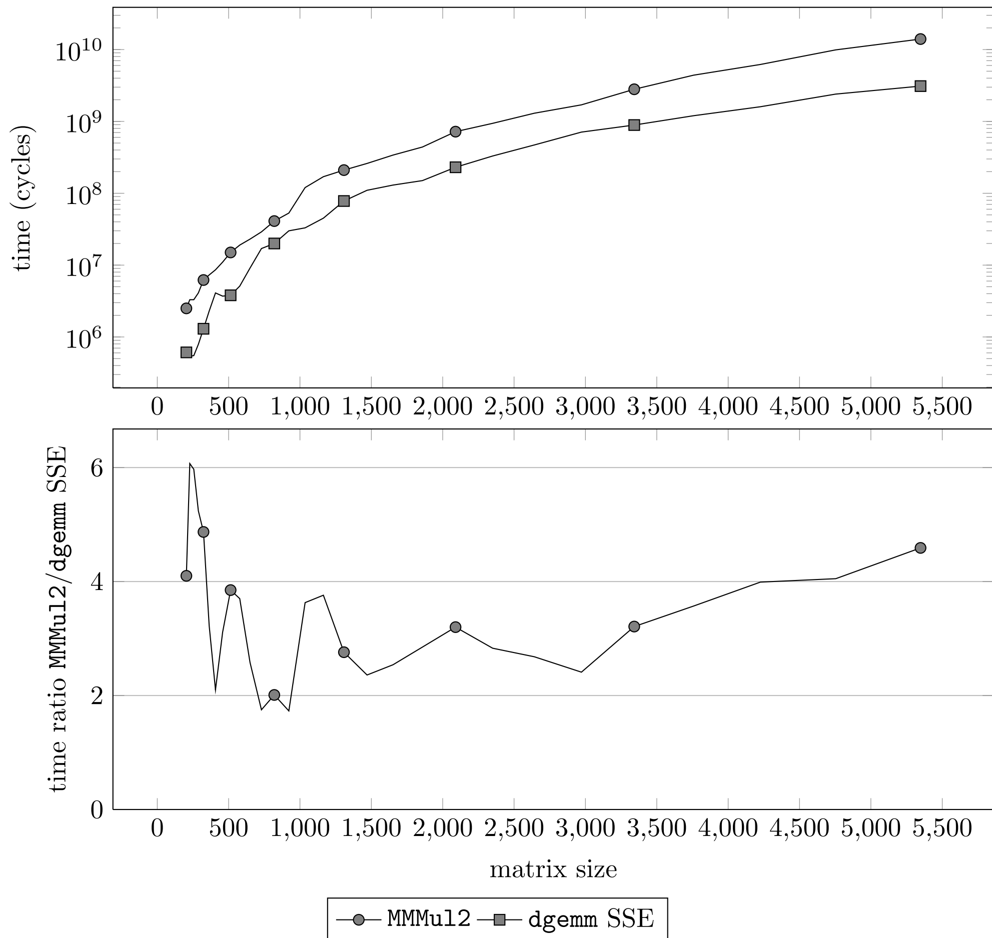

Figure 8.3: Execution time – 32 threads.

Here, the behaviors of `dgemm` and of `MMMul2` are less regular than in the sequential execution or with 8 execution threads. Their ratio varies with a greater amplitude, between 1.75 and 6.07 in this experiment. Moreover, the trend seems to be an increase in the ratio with increasing matrix dimension.

Note that some of the 32 threads are idle if the matrix dimension is too small. Precisely, the execution times for matrix dimensions less than 1,024 do not correspond to parallel executions of `mul_mmd_2000` with 32 threads. In fact, the code assigns the computation of rows of block matrices to the threads (see line 7 in Listing 8.1). So, we need at least 32 block rows for being able to assign an iteration over the block rows to each thread. The block dimension being 32, the row matrix dimension should be greater than $32 \times 32 = 1,024$. With smaller dimensions, the work is distributed to a smaller number of threads in the team. However, when the dimension of the square matrices is greater than or equal to 1,024, each thread processes a substantial amount of computation: it has to compute at least 32768 interval components[1] in the result matrix.

[1] 32 blocks of size 32×32.

8.3.3 Scalability

We now analyze the *strong scalability* of `dgemm` and `MMMul2`, that is, their behavior when the number of threads rises while the matrix dimension remains constant. The relevant metric in this context is the measure of *efficiency*. By definition, the efficiency of a parallel implementation executed with p threads is the ratio $T_1/(p \times T_p)$, where T_1 is the sequential execution time, and T_p is the execution time using p execution threads. It equals one when the parallelism is perfect.

Figure 8.4 represents the variation of the measured execution times when the matrix dimension[2] is fixed to 1024×1024 and the number of threads increases.

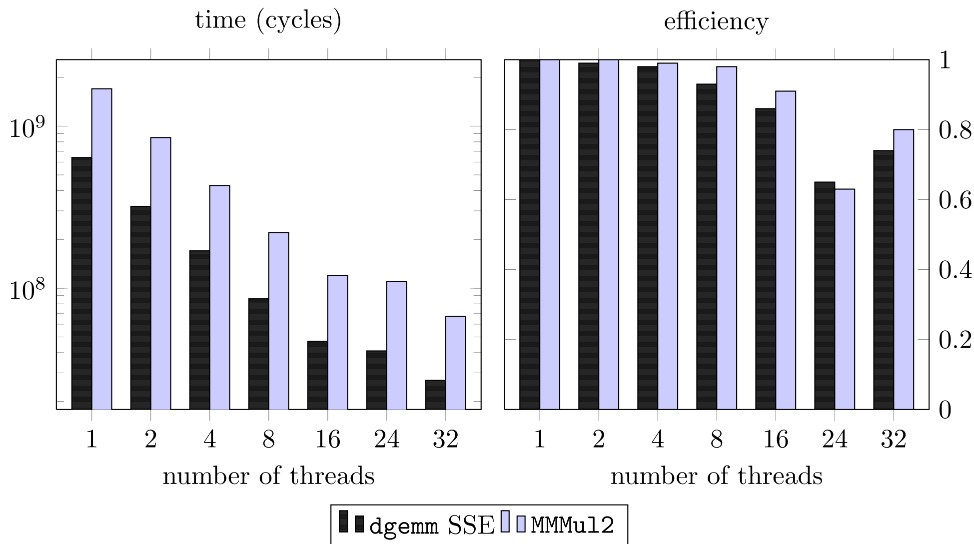

Figure 8.4: Scalability of `MMMul2` and `dgemm` – 1024-by-1024 Matrices.

The measured times and ratios corresponding to Figure 8.4 are also displayed in Table 8.1 below.

thread	`dgemm`		`MMMul2`		time ratio
number	time	efficiency	time	efficiency	`MMMul2` / `dgemm`
1	$6.4 \cdot 10^8$	1.00	$1.7 \cdot 10^9$	1.00	2.7
2	$3.2 \cdot 10^8$	0.99	$8.5 \cdot 10^8$	1.00	2.7
4	$1.6 \cdot 10^8$	0.98	$4.3 \cdot 10^8$	0.99	2.5
8	$8.6 \cdot 10^7$	0.93	$2.2 \cdot 10^8$	0.98	2.6
16	$4.7 \cdot 10^7$	0.86	$1.2 \cdot 10^8$	0.91	2.6
24	$4.1 \cdot 10^7$	0.65	$1.1 \cdot 10^8$	0.63	2.7
32	$2.6 \cdot 10^7$	0.76	$6.7 \cdot 10^7$	0.80	2.5

Table 8.1: Measured Times (in cycles) and Efficiency of `MMMul2` and `dgemm` – 1024-by-1024 Matrices.

The middling efficiency with 24 threads can be easily explained. With 1024-by-1024 matrices,

[2]The behavior of the matrix multiplication with square matrices of dimension up to 5,000 is similar to the one with 1024×1024 matrices, which is described here as a representative example.

we have 32 block rows to distribute among threads. If the thread team comprises 8, 16, or 32 threads, then each thread processes 4, 2, or 1 block row(s) respectively. If the thread team comprises 24 threads, then the first 8 ones process 2 block rows and others process only 1 block row. This explains why the computing times for 16 and 24 threads are comparable: some threads have the same amount of work in both cases.

As it can be seen from this data, the efficiency of our `MMMul2` implementation is close to the efficiency of the `dgemm` implementation of the MKL, sometimes even superior.

8.3.4 The goal of strong scalability is reached

The experimental measures presented in this section show that the `MMMul2` implementation reaches the scalability goal as defined in Section 6.1. In particular, the efficiency is very high, more than 90%, with 8 threads, that is, when the number of threads is equal to the number of cores of a processor. This high percentage is an indication that the implementation takes advantage of the multi-core architecture, ending up as an interval matrix multiplication that costs no more than 3 times the cost of a well-optimized floating-point matrix product.

With more than 8 threads, the timings show a worrisome increasing trend when matrix dimension grows. More investigation would be needed to determine the cause of this increase. Nevertheless, the ratio of execution times still remains less than 6 for matrix dimensions less than 5,000.

8.4 Conclusion

We conclude this part on the parallel implementation by comparing our implementation of the interval matrix product, on the one hand, with the implementation where a multi-threaded BLAS library is called, what was intended by previous authors, and, on the other hand, with the implementations techniques of numerical, i.e. non-interval, matrix products.

As shown in Chapter 5, the use of external libraries is problematic because the rounding mode may not be taken into account. The use of a parallel BLAS library makes the problem even worse: the underlying multi-thread library may not respect the rounding mode. These concerns about the correctness of the corresponding executable code exclude such an implementation for a computation that one claims to be certified. Moreover, implementing the interval matrix multiplications with the parallel BLAS also reduces the scalability of the program. In [RT13], we compared the implementation of `MMMul5`, where some of the numerical matrix products are combined in the same loop, with the one where each product is performed with a call to a parallel BLAS function. The conclusion was that the former implementation is more scalable than the latter. The reason is that the latter case imposes five global synchronization points, one after each call to a BLAS function, whereas the version where a custom parallel function computes in the same loop the four numerical matrix products of the lines 3 and 4 of Algorithm 7 page 45 only requires two global synchronization points: one after this custom function and one after the last numerical matrix multiplication (line 5 of Algorithm 7). This lack of strong scalability of the parallelization of a linear algebra operation through calls of some parallel BLAS functions is well-known and is discussed in the case of matrix factorization in [BLKD09] or [BDK$^+$07], for example.

On the contrary, correctness and acceptable levels of sequential performance and of strong scalability may be obtained by classical implementation techniques for numerical matrix multiplications. The key idea is the use of a specific data structure, where small square sub-matrices of the interval matrix are stored in contiguous arrays. First, it improves the spatial and temporal locality of memory accesses without needing to copy data. As a result, the performance is better.

Second, such block structures can be processed by small and efficient block kernels, while the overall computation is spread over the several cores with multiple threads. This distinction of two levels in the computation matches the distinction of the inter- and intra-core levels in the hardware. It also affords opportunity to use classical algorithms at the higher level with slight modifications, the difference being that the algorithm operates on blocks instead of components. At the multi-thread level, the parallelization is done with OpenMP threads, by simply the adding an OpenMP annotation. At the single thread level, the result block is computed in a sequential way, setting the correct rounding mode systematically, without making any assumption on the current rounding status. So, the correctness of the interval computation also depends on this distinction of two levels, provided that it is correctly handled by the compiler. And finally, it is possible to use loop fusion, loop unrolling, vectorization and removal of conditional branches for the implementation of block kernels. Due to the higher level of complexity of interval computation, current compilers may be unable to exploit these optimizations automatically and, unfortunately, it is sometimes necessary to apply them manually. The resulting implementation of the `MMMUl2` algorithm then shows guaranty on the computed product, a small three-fold increase in computation time and a comparable strong scalability, with respect to a product of mere numerical matrices of the same size.

Conclusion

Our contributions

First, we quantified the numerical quality. Former error analyses of interval matrix products establish bounds on the radius overestimation by neglecting the roundoff error. We discussed several possible measures for interval approximations. We then bounded the roundoff error and compared experimentally this bound with the global error distribution on several random data sets. This approach enlightens the relative importance of the roundoff and arithmetic errors depending on the value and homogeneity of relative accuracies of inputs, on the matrix dimension, and on the working precision. This also leads to a new algorithm that is cheaper yet as accurate as previous ones under well-identified conditions.

Second, we exploited the parallelism of linear algebra. Previous implementations use calls to BLAS routines on numerical matrices. We showed that this may lead to wrong interval results and also restrict the scalability of the performance when the core count increases. To overcome these problems, we implemented a blocking version with OpenMP threads executing block kernels with vector instructions. The timings on a 4-octo-core machine show that this implementation is more scalable than the BLAS one and that the cost of numerical and interval matrix products are comparable.

Perspectives and applications

The work presented here could be extended in several directions.

Interval BLAS library

The efficient implementation of interval matrix multiplication is the first step toward an interval BLAS library that would provide the set of matrix operations that is defined in the BLAS standard. The same approach of block algorithms and the same optimization techniques may be employed to implement other arithmetic operations on interval matrices with the guaranty that the inclusion property is verified.

It would be much more difficult to implement, with our approach, parallel interval matrix factorizations or a certified parallel solution of triangular system of linear equations. Actually, the corresponding floating-point algorithms involve subtractions and divisions that are detrimental to the accuracy of interval computations.

Other targets – Sparse Matrices

The data structure we have chosen use a decomposition of interval matrices into small blocks. By choosing a block size adapted to the characteristics of the underlying platform, our approach

should give good results on other targets like GPUs and other many-core accelerators. Another favorable aspect of our interval matrix multiplication is that it uses only a static scheduling of the threads. This is particularly important for the execution on current GPUs that do not have enough capabilities to handle complex conditional code.

The structure in small blocks is also suitable to sparse matrices that contains a high percentage of zero values. By storing only the block that contains non-zeroes in a compressed row (or column) storage structure, one could adapt the sparse algorithms into block algorithms in the same manner that dense matrix product has been adapted here to interval block matrix product.

Adaptive algorithm for interval matrix multiplication

At last, let us envisage an interesting improvement of numerical matrix computations in the light of the conclusions of our accuracy analysis.

Starting from the point of view that the users may know the accuracy of a matrix product that would fit their needs, it could be possible to express the wanted accuracy as an upper bound on the relative accuracies of the result. If this bound is passed as a parameter to an interval matrix multiplication function, an adaptive algorithm could try to compute a result within the given accuracy bound. Let us sketch such an adaptive algorithm:

1. Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be the input interval matrices, maybe reduced to thin matrices or matrices with a uniform relative accuracy of one ulp if the initial problem is to compute floating-point matrix products. Let E be the wanted maximum relative accuracy for the result.

2. In a first step, an approximate product $\boldsymbol{C}_2 \supseteq \boldsymbol{AB}$ is computed using the `MMMul2` algorithm.

3. If all maximum relative accuracies for components of $\boldsymbol{C}_2$ are less than E, then the problem is solved and $\boldsymbol{C}_2$ is an acceptable solution.

4. Otherwise, the components that satisfy the requirement are filtered out and the other one are recomputed as follows. The maximum relative accuracies in $\boldsymbol{A}$ and $\boldsymbol{B}$ being already computed, one can use the decision tree of Figure 4.16 and the program chooses the best algorithm between `MMMul3` and `MMMul5` to compute a tighter enclosure for the remaining components.

The choice between `MMMul2`, `MMMul3`, and `MMMul5` could also be made at the block kernel level, after the vector of maximum relative accuracies are computed.

Bibliography

[BCD⁺01] S. Blackford, G. Corliss, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, C. Hu, W. Kahan, L. Kaufmann, B. Kearfott, F. Krogh, X. Li, Z. Maany, A. Petitet, R. Pozo, K. Remington, W. Walster, C. Whaley, J. Wolff von Gudenberg, and A. Lumsdaine. *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard*, 2001. http://www.netlib.org/blas/blast-forum/.

[BDK⁺07] Alfredo Buttari, Jack J. Dongarra, Jakub Kurzak, Julien Langou, Piotr Luszczek, and Stanimire Tomov. The impact of multicore on math software. In Bo K*r*agström, Erik Elmroth, Jack J. Dongarra, and Jerzy Waśniewski, editors, *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 2007.

[BLKD09] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, January 2009.

[FHL⁺07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[gcc08] GCC bug 34678 – optimization generates incorrect code with -frounding-math option. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=34678, January 2008.

[gcc11] GCC bug 47617 – sse rounding mode works -g, not -o3. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=47617, February 2011.

[GFMR13] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Boston, Massachusetts, USA, May 2013. http://hal.inria.fr/hal-00799904/PDF/ipdps2013.pdf.

[Gou14] Frédéric Goualard. How do you compute the midpoint of an interval? *ACM Transactions on Mathematical Software*, 40(2):11:1–11:25, March 2014.

[Gus06] Fred G. Gustavson. New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms. In *Proceedings of the 7th international conference on Applied Parallel Computing: state of the Art in Scientific Computing*, PARA'04, pages 11–20. Springer-Verlag, 2006.

[Gus08] Fred G. Gustavson. The relevance of new data structure approaches for dense linear algebra in the new multi-core/many core environments. In *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, PPAM'07, pages 618–621. Springer-Verlag, 2008.

[Gus12] Fred G. Gustavson. Cache blocking for linear algebra algorithms. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Proceedings of the 9th International Conference On Parallel Processing And Applied Mathematics*, volume 7203 of *Lecture Notes in Computer Science*, pages 122–132. Springer, 11–14 September 2012.

[Gv08] Kazushige Goto and Robert A. van. de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12:1–12:25, May 2008.

[Hig02] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, second edition, 2002.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, 4th edition, 2007.

[IEE08] *IEEE Standard for Floating-Point Arithmetic*, Aug 2008.

[Int] Intel. *Intel Xeon Phi Coprocessor Developer's Quick Start Guide.* `http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide`.

[Int13a] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2013. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`.

[Int13b] Intel. *Intel C++ Compiler XE 13.1 User and Reference Guide*, 2013. `https://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm`.

[Int14a] Intel. *Intel 64 and IA-32 Architectures Software Developers Manuals*, 2014. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`.

[Int14b] Intel Corporation. *Intel Threading Building Blocks Reference Manual*, 2014. `https://software.intel.com/en-us/node/506130`.

[ISO99] ISO. The ANSI C standard (C99). Technical report, ISO/IEC, 1999. `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf`.

[IT13] IEEE and The Open Group. *The Open Group Base Specifications*, 7 edition, 2013. `http://pubs.opengroup.org/onlinepubs/9699919799/`.

[Khr11] Khronos OpenCL Working Group. *The OpenCL Specification.* Khronos Group, November 2011. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[KLY+13] Jakub Kurzak, Piotr Luszczek, Asim YarKhan, Mathieu Faverge, Julien Langou, Henricus Bouwmeester, and Jack Dongarra. *Multithreading in the PLASMA Library*, chapter Multicore Computing, pages 119–141. Computer & Information Science Series. Chapman & Hall/CRC, 2013.

[KM03] Zenon Kulpa and Svetoslav Markov. On the inclusion properties of interval multiplication: A diagrammatic study. *BIT Numerical Mathematics*, 43:791–810, 2003.

[KNN+10] R. Baker Kearfott, Mitsuhiro T. Nakao, Arnold Neumaier, Siegfried M. Rump, Sergey P. Shary, and Pascal van Hentenryck. Standardized notation in interval analysis. *Computational Technologies*, 15(1):7–13, 2010.

[Knü94] Olaf Knüppel. PROFIL/BIAS–a fast interval library. *Computing*, 53(3-4):277–287, 1994.

[Kre13] Vladik Kreinovich. How to define relative approximation error of an interval estimate: A proposal. *Applied Mathematical Sciences*, 7(5):211–216, 2013.

[Lam06] Branimir Lambov. Interval arithmetic using sse-2. In Peter Hertling, Christoph M. Hoffmann, Wolfram Luther, and Nathalie Revol, editors, *Reliable Implementation of Real Number Algorithms: Theory and Practice*, volume 5045 of *Lecture Notes in Computer Science*, pages 102–113. Springer-Verlag, 2006.

[LDB+02] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software*, 28(2):152–205, June 2002.

[LMM12] Christoph Lauter and Valérie Ménissier-Morain. There's no reliable computing without reliable access to rounding modes. In Institute of Computational Technologies, editor, *SCAN 2012 Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics*, pages 99–100, 2012.

[MBdD+10] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[MZG+07] Bryan Marker, Field Van Zee, Kazushige Goto, Gregorio Quintana-Orti, and Robert van de Geijn. Toward scalable matrix multiply on multithreaded architectures. *Euro-Par 2007 Parallel Processing*, pages 748–757, 2007.

[Neu90] Arnold Neumaier. *Interval methods for systems of equations*. Cambridge University Press, Cambridge, 1990.

[Ngu11] Hong Diep Nguyen. *Efficient algorithms for verified scientific computing: numerical linear algebra using interval arithmetic*. PhD thesis, École Normale Supérieure de Lyon – Université de Lyon, 2011. `http://hal-ens-lyon.archives-ouvertes.fr/ensl-00560188`.

[OO05] Takeshi Ogita and Shin'ichi Oishi. Fast inclusion of interval matrix multiplication. *Reliable Computing*, 11(3):191–205, 2005.

[OOO11] Katsuhisa Ozaki, Takeshi Ogita, and Shin'ichi Oishi. Tight and efficient enclosure of matrix multiplication by using optimized BLAS. *Numerical Linear Algebra with Applications*, 18(2):237–248, 2011.

[OORO12] Katsuhisa Ozaki, Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Fast algorithms for floating-point interval matrix multiplication. *Journal of Computational and Applied Mathematics*, 236:1795–1814, 2012.

[Ope13] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. OpenMP Architecture Review Board, July 2013. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[OR02] Shin'ichi Oishi and Siegfried M. Rump. Fast verification of solutions of matrix equations. *Numerische Mathematik*, 90(4):755–773, 2002.

[QOQOG+09] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. Van De Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3):14:1–14:26, July 2009.

[Rat72] H. Ratschek. Teilbarkeitskriterien der intervallarithmetik. *Journal für die reine und angewandte Mathematik*, 252:128–138, 1972.

[RS82] H. Ratschek and W. Sauer. Linear interval equations. *Computing*, 28(2):105–115, 1982.

[RT13] Nathalie Revol and Philippe Théveny. Parallel implementation of interval matrix multiplication. *Reliable Computing*, 19(1):91–106, 2013.

[RT14] Nathalie Revol and Philippe Théveny. Numerical reproducibility and parallel computations: Issues for interval algorithms. *IEEE Transactions on Computers*, 2014. (to appear).

[Rum99a] Siegfried M. Rump. Fast and parallel interval arithmetic. *BIT Numerical Mathematics*, 39:534–554, 1999.

[Rum99b] Siegfried M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. `http://www.ti3.tuhh.de/rump/`.

[Rum10] Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.

[Rum12] Siegfried M. Rump. Fast interval matrix multiplication. *Numerical Algorithms*, 61(1):1–34, 2012.

[RZBM] Siegfried M. Rump, Paul Zimmermann, Sylvie Boldo, and Guillaume Melquion. Interval operations in rounding to nearest. `http://www.ti3.tu-harburg.de/paper/rump/RuZiBoMe09.pdf`.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[Tod12] R. Todd. Introduction to conditional numerical reproducibility (CNR). http://software.intel.com/en-us/articles/introduction-to-the-conditional-numerical-reproducibility-cnr, 2012.

[VCv+09] Field G. Van Zee, Ernie Chan, Robert A. van. de Geijn, Enrique S. Quintana-Orti, and Gregorio Quintana-Orti. The libflame library for dense matrix computations. *IEEE Computing in Science & Engineering*, 11(6):56–63, November 2009.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC 1998)*, pages 1–27. IEEE, 7–13 November 1998. www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps.

[WPD00] R. Clint Whaley, Antoine P. Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. Technical Report 147, LAPACK Working Note, September 2000. http://www.netlib.org/lapack/lawnspdf/lawn147.pdf.

[YKD11] Asim YarKhan, Jakub Kurzak, and Jack J. Dongarra. QUARK users' guide: QUeueing And Runtime for Kernels. Technical report, University of Tennessee Innovative Computing Laboratory, 2011. http://icl.cs.utk.edu/projectsfiles/plasma/pubs/56-quark_users_guide.pdf.

[ZJH09] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Accuracy of performance counter measurements. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–32. IEEE, 26–28April 2009.

Résumé

Algèbre Linéaire d'Intervalles : Qualité Numérique et Haute Performance sur Processeurs Multi-Cœurs

L'objet de la thèse est de comparer des algorithmes de multiplication de matrices à coefficients intervalles et leurs implémentations.

Le premier axe est la mesure de la précision numérique. Les précédentes analyses d'erreur se limitent à établir une borne sur la surestimation du rayon du résultat en négligeant les erreurs dues au calcul en virgule flottante. Après examen des différentes possibilités pour quantifier l'erreur d'approximation entre deux intervalles, l'erreur d'arrondi est intégrée dans l'erreur globale. À partir de jeux de données aléatoires, la dispersion expérimentale de l'erreur globale permet d'éclairer l'importance des différentes erreurs (de méthode et d'arrondi) en fonction de plusieurs facteurs : valeur et homogénéité des précisions relatives des entrées, dimensions des matrices, précision de travail. Cette demarche conduit à un nouvel algorithme moins coûteux et tout aussi précis dans certains cas déterminés.

Le deuxième axe est d'exploiter le parallélisme des opérations. Les implémentations précédentes se ramènent à des produits de matrices de nombres flottants. Pour contourner les limitations d'une telle approche sur la validité du résultat et sur la capacité à monter en charge, je propose une implémentation par blocs réalisée avec des threads OpenMP qui exécutent des noyaux de calcul utilisant des instructions vectorielles. L'analyse des temps d'exécution sur une machine de 4 octo-coeurs montre que les coûts de calcul sont du même ordre de grandeur sur des matrices intervalles et numériques de même dimension et que l'implémentation par blocs passe mieux à l'échelle que l'implémentation avec plusieurs appels aux routines BLAS.

Mots-Clés : algèbre linéaire numérique, multiplication de matrices, implémentation parallèle, processeurs multi-cœurs, memoire partagée, virgule flottante, analyse d'erreur, arithmétique d'intervalles.

Abstract

Numerical Quality and High Performance in Interval Linear Algebra on Multi-Core Processors

This work aims at determining suitable scopes for several algorithms of interval matrices multiplication.

First, we quantify the numerical quality. Former error analyses of interval matrix products establish bounds on the radius overestimation by neglecting the roundoff error. We discuss here several possible measures for interval approximations. We then bound the roundoff error and compare experimentally this bound with the global error distribution on several random data sets. This approach enlightens the relative importance of the roundoff and arithmetic errors depending on the value and homogeneity of relative accuracies of inputs, on the matrix dimension, and on the working precision. This also leads to a new algorithm that is cheaper yet as accurate as previous ones under well-identified conditions.

Second, we exploit the parallelism of linear algebra. Previous implementations use calls to BLAS routines on numerical matrices. We show that this may lead to wrong interval results and also restrict the scalability of the performance when the core count increases. To overcome these problems, we implement a blocking version with OpenMP threads executing block kernels with vector instructions. The timings on a 4-octo-core machine show that this implementation is more scalable than the BLAS one and that the cost of numerical and interval matrix products are comparable.

Keywords: Numerical linear algebra, matrix multiplication, parallel implementation, multicore processors, shared memory, floating-point number, error analysis, interval arithmetic.