A Runtime System for Data-Flow Task Programming on Multicore Architectures with Accelerators

Joao Vicente Ferreira Lima

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

prépareé dans le cadre d'une cotutelle entre l'*Université de Grenoble* et l'*Universidade Federal do Rio Grande do Sul*

Spécialité : **Informatique**

Arrêté ministérial : 6 janvier 2005 – 7 août 2006

Présentée par

João Vicente FERREIRA LIMA

Thèse dirigée par **Bruno RAFFIN et Nicolas MAILLARD**
et codirigée par **Vincent DANJEAN**

préparée au Laboratoire d'Informatique de Grenoble dans le cadre de **l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique** et au Laboratoire de Parallelisme et Distribution dans le cadre du **Programme de Doctorat en Informatique**

A Runtime System for Data-Flow Task Programming on Multicore Architectures with Accelerators

Thèse soutenue publiquement le **5 mai 2014**,
devant le jury composé de :

M. Philippe O. A. NAVAUX
Professeur, Universidade Federal do Rio Grande do Sul, Président
M. Jairo PANETTA
Professeur, CPTEC/INPE, Rapporteur
M. Pierre MANNEBACK
Professeur, Université de Mons, Rapporteur
Mme Andrea CHARÃO
Maître de Conférences, Universidade Federal de Santa Maria, Examinatrice
M. Bruno RAFFIN
Chargé de Recherche, Université de Grenoble, Directeur de thèse
M. Nicolas MAILLARD
Maître de Conférences, Universidade Federal do Rio Grande do Sul, Directeur de thèse

Abstract

In this thesis, we propose to study the issues of task parallelism with data dependencies on multicore architectures with accelerators. We target those architectures with the XKaapi runtime system developed by the MOAIS team (INRIA Rhône-Alpes).

We first studied the issues on multi-GPU architectures for asynchronous execution and scheduling. Work stealing with heuristics showed significant performance results, but did not consider the computing power of different resources. Next, we designed a scheduling framework and a performance model to support scheduling strategies over XKaapi runtime. Finally, we performed experimental evaluations over the Intel Xeon Phi coprocessor in native execution.

Our conclusion is twofold. First we concluded that data-flow task programming can be efficient on accelerators, which may be GPUs or Intel Xeon Phi coprocessors. Second, the runtime support of different scheduling strategies is essential. Cost models provide significant performance results over very regular computations, while work stealing can react to imbalances at runtime.

Keywords: parallel programming, accelerators, task parallelism, data flow dependencies, work stealing.

Resumo

Uma Ferramenta para Programação com Dependência de Dados em Arquiteturas Multicore com Aceleradores

Esta tese investiga os desafios no uso de paralelismo de tarefas com dependências de dados em arquiteturas multi-CPU com aceleradores. Para tanto, o XKaapi, desenvolvido no grupo de pesquisa MOAIS (INRIA Rhône-Alpes), é a ferramenta de programação base deste trabalho.

Em um primeiro momento, este trabalho propôs extensões ao XKaapi a fim de sobrepor transferência de dados com execução através de operações concorrentes em GPU, em conjunto com escalonamento por roubo de tarefas em multi-GPU. Os resultados experimentais sugerem que o suporte a asincronismo é importante à escalabilidade e desempenho em multi-GPU. Apesar da localidade de dados, o roubo de tarefas não pondera a capacidade de processamento das unidades de processamento disponíveis. Nós estudamos estratégias de escalonamento com predição de desempenho em tempo de execução através de modelos de custo de execução. Desenvolveu-se um framework sobre o XKaapi de escalonamento que proporciona a implementação de diferentes algoritmos de escalonamento. Esta tese também avaliou o XKaapi em coprocessodores Intel Xeon Phi para execução nativa.

A conclusão desta tese é dupla. Primeiramente, nós concluímos que um modelo de programação com dependências de dados pode ser eficiente em aceleradores, tais como GPUs e coprocessadores Intel Xeon Phi. Não obstante, uma ferramenta de programação com suporte a diferentes estratégias de escalonamento é essencial. Modelos de custo podem ser usados no contexto de algoritmos paralelos regulares, enquanto que o roubo de tarefas poder reagir a desbalanceamentos em tempo de execução.

Palavras-chave: Programação paralela, aceleradores, paralelismo de tarefas, dependência de dados, roubo de tarefas.

Résumé

Vers un Support Exécutif avec Dépendance de Données pour les Architectures Multicœur avec des Accélérateurs

Dans cette thèse , nous proposons d'étudier des questions sur le parallélism de tâche avec dépendance de données dans le cadre de machines multicœur avec des accélérateurs. La solution proposée a été développée en utilisant l'interface de programmation haute niveau XKaapi du projet MOAIS de l'INRIA Rhône-Alpes.

D'abord nous avons étudié des questions liés à une approche d'exécution totalement asyncrone et l'ordonnancement par vol de travail sur des architectures multi-GPU. Le vol de travail avec localité de données a montré des résultats significatifs, mais il ne prend pas en compte des différents ressources de calcul. Ensuite nous avons conçu une interface et une modèle de coût qui permettent d'écrire des politiques d'ordonnancement sur XKaapi. Finalement on a évalué XKaapi sur un coprocesseur Intel Xeon Phi en mode natif.

Notre conclusion est double. D'abord nous avons montré que le modèle de programmation data-flow peut être efficace sur des accélérateurs tels que des GPUs ou des coprocesseurs Intel Xeon Phi. Ensuite, le support à des différents politiques d'ordonnancement est indispensable. Les modèles de coût permettent d'obtenir de performance significatifs sur des calculs très réguliers, tandis que le vol de travail permet de redistribuer la charge en cours d'exécution.

Mots-clés : Programmation parallèle, accélérateur, parallélisme de tâche, dépendance de données, vol de travail.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Parallelism can be see as an ubiquitous aspect in life. Although our speech is sequential, music has a lot of concurrent elements at same time presented by each instrument. Yet, those instruments play a stream of sound that may not sense anything pleasant if taken isolated. In parallel computing, there are many research areas ranging from architectures to high-level paradigms that can not be taken separately. For instance, parallel programming models abstract underlying architecture details; but it also may be depend on hardware tendencies.

Most programmers may prefer sequential processors as observed in the years predominated by technology advances such as deep pipeline, out-of-order, speculative processors. These architectures maintained the existing sequential programming model and the increasing performance at affordable price for technology at that time. However, this demand for more performance led to inefficient chips in terms of transistor and power (Asanovic et al., 2009).

In the last few years the industry decided to replace the power-inefficient processor with efficient processors on the same chip, providing more and more on-die cores each year. Current microprocessors are homogeneous multicore chips containing from two to sixteen cores, with even higher core counts in the near future. An architectural trend is the emergence of manycore accelerators with many tightly coupled processing units (PU) such as graphics cards (GPU) or Intel Xeon Phi coprocessors for high performance computing (HPC). Although multicore chips seeks to maintain the execution speed of sequential programs while moving into multiple cores, accelerators favor the execution throughput (Kirk and Hwu, 2012). Therefore, such architectures have heterogeneous PUs in terms of computing power and programming model.

Hence, in many cases, familiar and widely used algorithms need to be rethought and rewritten to take advantage of modern multicore and manycore architectures. Parallel algorithms with fine granularity and asynchronicity are essential in order to exploit parallelism and to improve scalability (Buttari et al., 2009).

1.1 Motivation

A challenge in heterogeneous systems is delegate work efficiently to take advantage of all available parallelism since the workload can be affected by the differences in the processing power of each PU. Besides, there are more issues to be considered as dependencies between tasks of different PUs and overhead costs. These costs come from decisions of where actually execute a task, loading its execution code at runtime, costs of library calls from

different PU softwares, and memory transfers required since the PUs do not share the same memory address space.

A well-known load balancing strategy is work stealing. It is a decentralized scheduling algorithm that whenever a processor runs out of work, it steals work from a randomly chosen processor. This strategy achieves provably good performance results for shared-memory systems, but it is not actually applied to heterogeneous systems because of its cache-unfriendly limitations (Acar et al., 2000) and co-processor nature of PU units.

Little research has been done on comparing scheduling strategies on heterogeneous architectures. Augonnet et al. (2009b) compare three strategies in one experiment, which may be insufficient to assert the strengths and weaknesses of each strategy. Most common strategies rely on cost models such as HEFT (Augonnet et al., 2010a) or centralized scheduling (Ayguadé et al., 2009a; Bueno et al., 2011, 2012). We note that both approaches have drawbacks. A cost model scheduling depends on regular computations and does not adapt to load variation at runtime. Besides, a centralized scheduling does not scale as the number of workers increase, and is not suitable for fine-grained computations.

1.2 Hypothesis

In order to deal with the issues on heterogeneous systems, we must consider some hypothesis about runtime systems for accelerators:

- A data-flow task programming model should allow to loose synchronization and exploit parallelism on accelerators;
- A runtime system should improve performance on accelerators by fully asynchronous operations;
- Work stealing scheduler with data locality heuristics should overcome its cache-unfriendly problem;
- Dynamic scheduling strategies should react to unbalances compared to cost models at runtime.

1.3 Objectives

The main objective of this work is to study the issues of task parallelism with data dependencies on multi-CPU architectures with accelerators. We target those architectures with the XKaapi runtime developed by the french team MOAIS (INRIA Rhône-Alpes).

We first propose the use of data-flow task programming model as a solution to program heterogeneous architectures. Task parallelism seems to be a well-suited programming model since parallelism is explicit and processor oblivious, *i.e.*, applications can unfold more parallelism than resources available. In addition, data-flow dependencies provide an explicit memory view and abstract data transfers.

Concerning the work distribution at runtime, we then study the work stealing scheduler for multi-GPU architectures. We overcome the cache-unfriendly problem of work stealing by locality heuristics that, consequently, reduce data transfers substantially.

In order to evaluate our work stealing strategies, we study strategies that predict execution cost at runtime by cost models. They may avoid erroneous decisions since work stealing does not consider the processing power of the available resources. We propose a broader approach in which a scheduling framework is designed to support different scheduling algorithms over multi-CPU and multi-GPU architectures.

In addition, one of our objectives is to evaluate the XKaapi runtime on the Intel Xeon Phi coprocessor. The coprocessor has support for x86 instructions and can run CPU programs entirely inside the accelerator. Our hypothesis is that our runtime may scale in a manycore architecture because of its overhead cost and scheduling by work stealing.

1.4 Contributions

The first contribution of this thesis concerns the proposal of XKaapi extensions in order to support multi-GPU systems. This contribution was published in two papers. The first was published in *IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'12)* (Lima et al., 2012). In this paper we described our concurrent GPU operations to overlap data transfer and execution, along with work stealing scheduler of XKaapi.

The second paper was published in *IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS'13)* (Gautier et al., 2013b). In this paper we described our proposed scheduling heuristics to overcome the cache-unfriendly problem of work stealing.

The second contribution of this thesis concerns a study of scheduling strategies that include dynamic scheduling and cost model with performance prediction. Part of this contribution was submitted to *Parallel Computing* journal. In this paper we evaluated different scheduling strategies over XKaapi scheduling framework on multi-CPU and multi-GPU architectures.

Finally, the third contribution of this thesis concerns the evaluation of a data-flow task programming model over the Intel Xeon Phi coprocessor. The preliminary results of our third contribution were presented in *IEEE 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'13)* (Lima et al., 2013). In this paper, we described preliminary results of data-flow programming and work stealing scheduler of XKaapi on the Intel Xeon Phi coprocessor in native mode.

We give an overview of our contributions in Figure 1.1 on the following page that are highlighted in green. XKaapi version 1.0.1 provided support for the basic building blocks of our contributions: data-flow programming model, multicore support, and work stealing scheduler. Since this version we extended the runtime at our first contribution with multi-GPU support and work stealing heuristics (Chapter 5). Furthermore, our first work originated our two other contributions: version 2.1 with multi-CPU and multi-GPU support and scheduling framework (Chapter 6) and version 2.0 for Intel MIC architecture (Chapter 7).

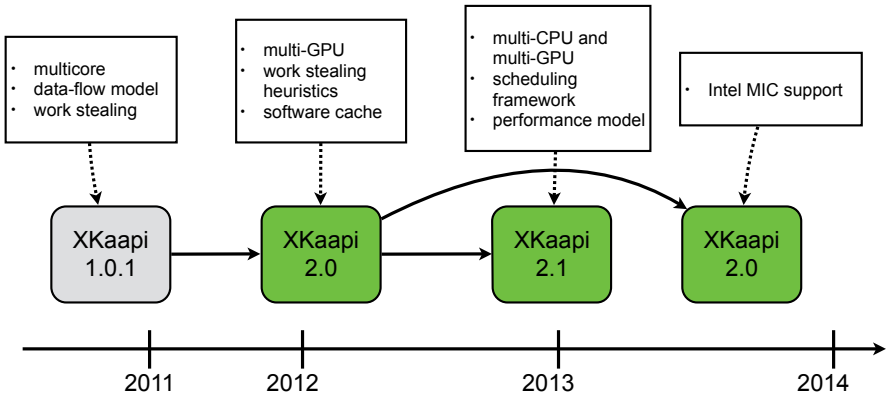


Figure 1.1 – Timeline of the contributions of this thesis (green boxes).

1.5 Context

In my master thesis at UFRGS, advised by Nicolas Maillard, I studied dynamic process creation in MPI in conjunction with system threads. After the master defense, I went to Grenoble for two months in order to establish a collaboration in the context of parallel programming with XKaapi. The initial PhD subject was defined after a number of meetings with Everton Hermann (early PhD student) and Bruno Raffin.

This work has been developed in the brazilian research group GPPD[1] (UFRGS) and the french research team MOAIS[2] (LIG). Besides, it has been involved in the context of the internacional associated lab LICIA[3] composed of research groups from UFRGS and LIG. This work is also part of the efforts for efficient execution of fine-grained algorithms on multicore architectures with the XKaapi environment developed by the MOAIS team. It also continues the work of Everton Hermann's thesis (Hermann, 2010), co-supervised MOAIS/EVASION, that studied the parallelization of SOFA with KAAPI for multi-GPU architectures using work stealing. In addition to my advisors Nicolas Maillard (advisor at UFRGS), Bruno Raffin (*directeur de thése* at LIG), and Vincent Danjean (*encadrant* at LIG), I worked in collaboration with:

- Claudio Schepke about parallel applications, mostly climatological models such as OLAM (Schepke et al., 2013);
- Stéfano D. K. Mór about scheduling algorithms and theoretical aspects;
- Thierry Gautier in XKaapi development and most of experiments;
- Grégory Mounié and Denis Trystram about scheduling algorithms for GPUs.

1.6 Thesis Outline

The text of this thesis is composed of three parts and eight chapters, as follows:

[1] `http://ppgc.inf.ufrgs.br`
[2] `http://moais.imag.fr`
[3] `licia-lab.org`

Part I – Parallel Programming
This Part, composed of three chapters, gives an overview on parallel architectures and programming, along with related works in runtime systems.

Chapter 2 – Background
In this Chapter we provide background for parallel architectures and programming models, as well as scheduling algorithms for multicore and manycore architectures.

Chapter 3 – Programming Environments
This Chapter presents related works on parallel programming environments. It starts with programming tools for multicore and SMP architectures. The Chapter ends with runtime systems that target multi-CPU architectures with accelerators.

Chapter 4 – XKaapi Runtime System
In this Chapter we detail the XKaapi runtime that is the basis of this work. We first describe its goal and programming interface. We then detail its work stealing scheduler from previous works. The Chapter ends with a description of XKaapi data-flow computation.

Part II – Contribution
In this Part, composed of four chapters, we present our contributions of this thesis.

Chapter 5 – Runtime Support for Multi-GPU Architectures
This Chapter describes our first contribution on runtime systems for multi-GPU. We first describe our extensions for multi-GPU systems such as asynchronous execution, concurrent GPU operations, and memory management. We then describe our work stealing heuristics to overcome its cache-unfriendly problem. The Chapter ends with our experimental results in order to evaluate XKaapi extensions and work stealing heuristics.

Chapter 6 – Scheduling Strategies over Multi-CPU and Multi-GPU Systems
In this Chapter we present a broader approach to evaluate different scheduling strategies on multi-CPU and multi-GPU architectures. It starts with a description of the scheduling framework and performance model to design strategies over XKaapi. We then detail the three scheduling strategies implemented on top of our framework. The Chapter ends with experimental results to evaluate the strategies.

Chapter 7 – Runtime Support for Native Mode on Intel Xeon Phi Coprocessor

This Chapter evaluates XKaapi in native execution on an Intel Xeon Phi coprocessor. It describes the modifications in order to optimize XKaapi runtime. The rest of the Chapter details the experimental results in three sets of benchmarks.

Chapter 8 – Conclusion
This Chapter presents the conclusion of this thesis. We also describe our contributions and future perspectives.

Part III – Appendixes
Finally, this Part contains the appendixes of this thesis.

Part I

Parallel Programming

Chapter 2

Background

The widespread usage of multicore processors has put parallel computing into evidence. Its ubiquitous presence let to the popularity of architectures and programming models in which programming is oblivious of hardware details. On the other hand, accelerators offer high throughput parallelism at cost of programming models that expose an explicit memory view. Their success comes from its power processing and low cost, allowing widespread usage through simulation and scientific computing.

In the remainder of this Chapter we will explain the basic concepts on parallel architectures and programming models, as well as scheduling algorithms. Our interest here is to introduce those concepts focusing on heterogeneous architectures composed of multicore processors and accelerators.

We first classify parallel architectures and describe multicore processors and manycore accelerators (Section 2.1). Next, we introduce the concept of parallel programming models and describe four models based on architectural aspects and decomposition strategies (Section 2.2). Finally, we discuss scheduling algorithms with focus on heterogeneous systems (Section 2.3).

2.1 Parallel Architectures

As mentioned earlier, current CPU microprocessors are homogeneous multicore chips containing from two to sixteen cores, with even higher core counts in the near future. A related architectural trend is the emergence of heterogeneous systems with many tightly coupled PUs such as graphics cards (GPUs), heterogeneous processors (Cell BE), or Intel Xeon Phi coprocessors for HPC. Therefore, such architectures have heterogeneous PUs in terms of computing power and programming model. In general, CPUs are serial with a small number of cores, and GPUs are parallel with hundreds of cores. An alternative to these homogeneous multicore chips is an asymmetric multicore chip, in which one or more cores are more powerful than the others (Hill and Marty, 2008).

This Section reviews basic principles in parallel architectures. We start presenting two complementary concepts to classify those architectures. The first is the Flynn taxonomy, and the second is based on the memory access and organization of the memory system. We then describe two big groups of parallel architectures. One is the group of general-purpose processors, which is composed of multicores, and the other is the group of specialized processors, which is composed of accelerators such as Cell BE, GPU, and Intel Xeon Phi. Finally we describe an example of heterogeneous architecture composed of multi-CPU and multi-GPU named Idgraf.

2.1.1 Architecture Models

Although there is no consensus in parallel community, the Flynn taxonomy (Flynn, 1972) is a frequently used classification in parallel systems. It classifies a system according to the number of instruction streams and the number of data streams it can manage simultaneously. The four classes are: Multiple Instruction Single Data (MISD), Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), and Multiple Instruction Multiple Data (MIMD). In parallel computing, the two former classes are not considered since SISD corresponds to a classical Von Neumann system and MISD is an unusual architecture.

SIMD architectures, which are also known as data-parallel or vector architectures, apply the same instruction to multiple data items. It can also be described as having a single control unit. A "classical" SIMD system must operate synchronously. They are employed on regular computations, which can be expressed as vector or matrix operations. Today, SIMD architectures are often found within mainstream processors, for example the MMX/SSE unit in the Intel Pentium processor line, and in accelerators such as GPUs and Intel Xeon Phi coprocessors.

MIMD systems support multiple simultaneous instruction streams operating on multiple data streams. They typically consist of a collection of independent processing units, each of which has its own control unit. In addition, unlike SIMD systems, MIMD systems are usually asynchronous. The main types of MIMD systems are detailed in Section 2.1.2.

2.1.2 Memory Systems

Parallel systems can also be classified by the memory access and organization of the memory system. There are mainly two types of memory systems: *shared-memory systems* and *distributed-memory systems*.

In shared-memory systems a collection of independent processors are connected to a memory system through an interconnection network, and each processor can access each memory location. The interconnect can either connect all the processors to the main memory, or each processor has direct access to a block of main memory and the processors can access other memory blocks by special hardware. Two types of shared-memory systems are the uniform memory access (UMA) and the non-uniform memory access (NUMA). In an UMA system the access cost of any memory address is uniform for all processors. Systems in this group are symmetric multiprocessors (SMP) and the first multicore processors.

In NUMA systems each processor, or group of processors, has its own local memory block. Similar to UMA, it offers an unified memory address space accessible by all processors. Data access in a local memory is faster than access in a remote memory unit. Most of NUMA systems are cache coherent, and the term CC-NUMA is often employed to emphase. Although the processor design may required more hardware, it has some advantages in respect with UMA architectures. NUMA machines with local memories for each processor reduces the bottleneck of a central memory when several processors attempt to access the memory at the same time. In addition, this architecture is more scalable than UMA enabling the employ of a larger number of processors. Examples of NUMA architecture are the AMD Opteron processors and the most recent Intel Xeon processors (Nehalem

and Sandy-Bridge). Both processors implement Cache Coherency protocol that employs dedicated hardware to control the coherency of cached data by processors.

In distributed-memory systems, each processor is paired with its own private memory and the processor-memory pairs communication over an interconnection network. Therefore, processors may communicate explicitly by sending messages or by using special functions that provide access to the memory of another processor. A well-known example of distributed systems are *clusters*. They are composed of a collection of commodity systems called *nodes* connected by commodity interconnection network, such as Ethernet. In fact, the term cluster can also be applied to a system that includes proprietary interconnects or special hardware designed for a specific parallel computer (Dongarra et al., 2003). These systems with non-commodity components are sometimes denominated *massively parallel processor* (MPP).

2.1.3 General Purpose Processors

Conceptually, the simplest approach to increasing the amount of work performed per clock cycle is to clone a single core multiple times on the chip. Each of these cores may execute independently, sharing data trough the memory system. This design is a scaling down of traditional multisocket SMP systems. However, multicore systems come in different guises, and it can be very difficult to define a core. For example, a mainstream CPU generally includes a wide range of functional blocks such that it is independent of other cores on the chip. However, some models shares functional units between pairs of cores. The aim of such a design is to raise efficiency by improving occupancy of functional units.

The AMD Phenon II represents AMD's current mainstream CPUs. It has up to six cores, allowing a high degree of parallel execution. The CPUs have large caches, up to 512 kB L2/core and 6MB shared. Each core carries a full 128-bit SSE unit that can issue add, multiply, and miscellaneous instructions simultaneously. It has a wide L1 cache interface (128 bits/cycle) and decodes, fetches, and issues six integer operations in parallel.

Intel's Sandy Bridge microarchitecture occupies a similar market segment. Like Phenom, Sandy Bridge supports full 128-bit SSE operations through multiple pipelines and issues up to six operations of mixed types in parallel. In addition, Sandy Bridge supports 256-bit Advanced Vector Extensions (AVX) operations, allowing up to 16 single precision floating point operations per cycle. As in Atom, Intel added multithreading support to Nehalem, and maintained this in Sandy Bridge. In this case, each core can mix operations from a pair of threads in the execution units.

UltraSPARC T1 microprocessor, codename Niagara, is a multithreaded SPARC processor designed by Sun Microsystems. It is optimized for multithreaded performance in commercial servers and increases the application performance by throughput (Kongetira et al., 2005). Niagara supports 32 hardware threads by combining ideas from chip multiprocessors and fine-grained multithreading. Four independent on-chip memory controllers provide about 20 GB/s of bandwidth to memory.

2.1.4 Manycore and Heterogeneous Architectures

In the last years, microprocessors have followed two different designing approaches (Kirk and Hwu, 2012). The *multicore* approach maximizes the execution speed of sequential

programs. A multicore microprocessor has dozens of out-of-order, multiple issue processor cores implementing the full x86 instruction set.

On the other hand, the *manycore* approach emphasis on the execution throughput of parallel programs. Manycore processors have a large number of heavely multithreaded, in-order, single-instruction cores. The hardware takes advantage of the massive number of threads to find work to do when some of them are in a stall on memory accesses. Besides, manycore PUs have been enhanced with arithmetic logical units (ALU) for floating-point operations such as fused multiple-add (FMA) from NVIDIA GPUs and Vector Processing Unit (VPU) from Intel coprocessors. Figure 2.1 illustrates the design differences between multicore (left) and manycore (right) microprocessors.

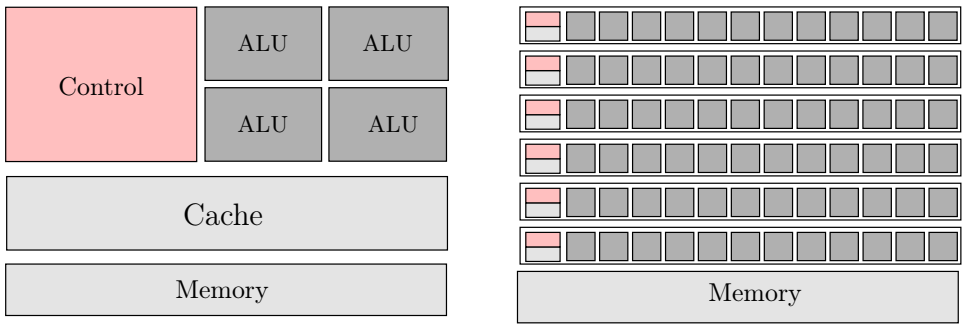


Figure 2.1 – Design differences between a multicore (left) and a manycore (right) microprocessor. This multicore layout is based on a basic model from recent microprocessors, and the manycore layout is similar to a GPU architecture representation.

Cell BE Heterogeneous Processor

The Cell Broadband Engine (Cell BE) architecture is a heterogeneous chip multiprocessor jointly developed by IBM, Sony, and Toshiba. The Cell processor consists of nine processors: a modified 64-bit Power PC core and eight simple SIMD RISC cores (Pham et al., 2005). The Power Processor Element (PPE) is the main core and optimized for control tasks, capable of running an operating system. It controls the eight Synergistic Processor Elements (SPE) optimized for data processing. The PPE instruction set extends the 64-bit Power Architecture with cooperative offload processors (the SPEs), with the direct memory access (DMA) and synchronization mechanisms to communicate with them, and with enhancements for real-time management (Kahle et al., 2005). On the other hand, the SPE implements a new instruction-set architecture optimized for power and performance on computing-intensive and media applications (Gschwind et al., 2006).

Cell BE processors are used from high-performance supercomputers, such as the IBM Roadrunner (Barker et al., 2008), to Playstation 3 game consoles, which may be the cheapest Cell-based system on the market. It contains a Cell processor (with six SPEs), 256 MB of main memory, an NVIDIA graphics card, and a gigabit Ethernet (GigE) network card. The Cell processor in the Playstation 3 is identical to the one found in high-end servers, with exception that two SPEs are not available. In spite of its power, the PlayStation

3 has severe limitations for scientific computing including floating-point problems, memory bandwidth limitations, and disproportional performance between the Cell processor's speed and that of the GigE interconnection (Kurzak et al., 2008).

The Cell BE has been employed in many applications on scientific computing (Cox et al., 2009; Panetta et al., 2009), in addition to a number of tools and environments (Augonnet et al., 2009a; Bellens et al., 2006; Gschwind et al., 2007; Ohara et al., 2006).

Graphics Processing Units and GPU Computing

Graphics Processing Unit (GPU) is a widespread class of accelerator. Over the past few years, GPUs have evolved from a fixed function special-purpose processor to a general-purpose architecture. In the early 2000's, GPU was a fixed-function processor, build around the graphics pipeline, composed mainly of two hardware units: the vertex units to compute geometrical transformations, and the fragment units to process pixels. This fixed-function pipeline lacked of generality and restricted GPU programming to graphics applications. The efforts at this time for general purpose computation, in conjunction with advances in microprocessor design, resulted in unified shader models. All programmable units in the pipeline shared an array of processing units, in which featured the same instruction set.

The programming model of modern GPUs follows a SIMD model with many processing units in parallel applying the same instruction. Each unit operates on integer or floating-point data with a general-purpose instruction set, and can read or write data from a shared global memory. This programming model allows branches in the code but not without performance loss. GPUs devote a large fraction of resources to computation. Supporting different execution paths on each unit requires a substantial amount of control hardware. Today's GPUs group units into blocks, and they are processed in parallel. If some units branch to different directions within a block, the hardware computes both sides of the branch for all units in the block. The size of a block is known as the "branch granularity" (Owens et al., 2008).

In the context of General-Purpose Computing on the GPU (GPGPU), programming for GPUs were not trivial since applications still had to be programed using graphics APIs. General-purpose programming APIs has been conceived to express applications in a familiar programming language. Examples of such APIs are NVIDIA's CUDA and OpenCL.

NVIDIA GPUs

In this direction, NVIDIA launched the G80 series (Lindholm et al., 2008) along with Tesla devices dedicated to HPC. Since the G80, the Compute Unified Device Architecture (CUDA) API is available, from high-end servers to desktops and embedded systems.

The architecture of a NVIDIA GPU is composed of streaming-processor (SP) cores organized as an array of streaming multiprocessors (SM). The number of SP cores and SMs can vary from one generation of GPUs to another. Figure 2.2 on the following page from Kirk and Hwu (2012, p. 9) shows an overview of a NVIDIA GPU architecture. Each SM has a number of SPs that share control logic and instruction cache. In addition, each SP core contains a scalar multiply-add (MAD) unit and floating-point multipliers. A

low-latency interconnect network between the SPs and the shared-memory banks provides shared-memory access.

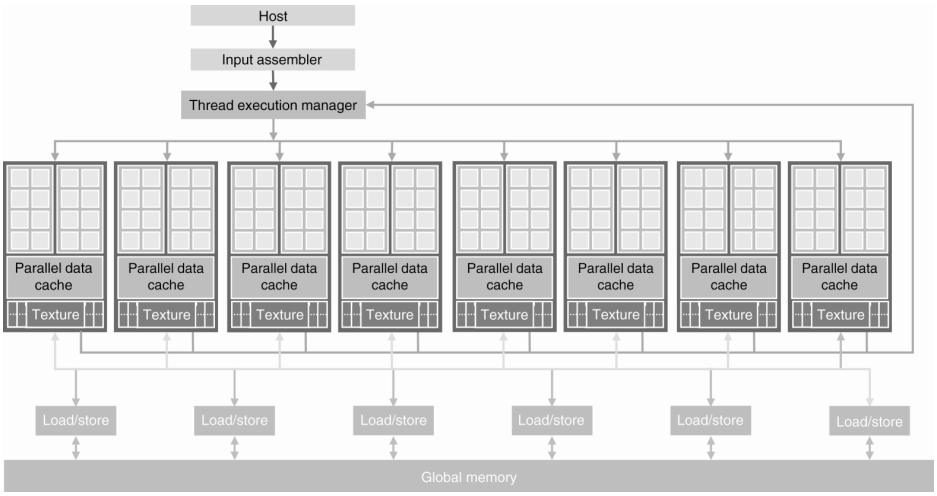


Figure 2.2 – Overview of a CUDA-capable GPU architecture.

An example of GPU architecture is the third-generation Fermi GPUs. It has 16 SMs, each with 32 SP cores, for a total of 512 cores. Each SM has a first-level (L1) data cache, and the SMs share a common 768KB unified second-level (L2) cache. The Fermi also introduces ECC memory protection to enhance data integrity in large-scale GPU computing systems. Each SM may execute up to $1,536$ concurrent threads to help cover long latency loads from global memory.

The SM hardware efficiently executes hundreds of threads in parallel while running several different programs. Each SM thread has its own thread execution state and can execute an independent code path. Concurrent threads of computing programs can synchronize at a barrier with a single SM instruction. The SM uses the single-instruction, multiple-thread (SIMT) processor architecture. The SM's SIMT multithreaded instruction unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*.

Since the G80 architecture, each SM manages a pool of warps. Individual threads composing a SIMT warp are of the same type and start together at the same program address, but they are otherwise free to branch and execute independently. At each instruction issue time, the SIMT instruction unit selects a warp that is ready to execute and issues the next instruction to that warp's active thread. A SIMT instruction is broadcasted synchronously to a warp's active parallel threads; individual threads can be inactive due to independent branching or prediction. A SIMT realizes full efficiency and performance when all 32 threads of a warp take the same execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path. Branch divergence only occurs within a warp; different warps execute independently. Figure 2.3 on the next page illustrates an example of branch divergence on the first warp.

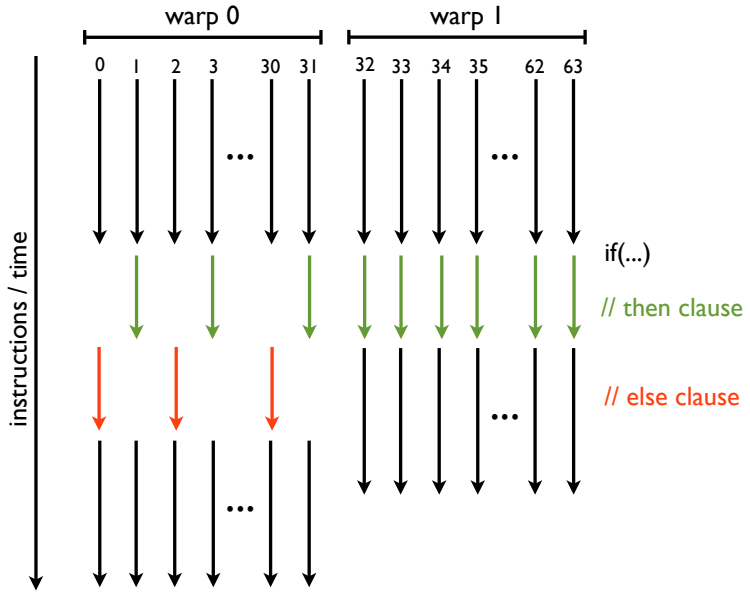


Figure 2.3 – GPU branch divergence example on the first warp.

Intel Coprocessor: Larrabee and MIC

The Larrabee architecture (Seiler et al., 2008) is a GPU based on an array of processor cores running an extended version of the x86 instruction set. Its architecture has multiple in-order x86 processor cores enhanced with a wide vector processor unit (VPU). The L2 cache of Larrabee is shared and connected with the on-die ring network. Unlike modern GPUs, the memory hierarchy supports transparently data sharing across all cores. Larrabee improves general-purpose programmability by supporting legacy x86 ISA. Besides, its programming model supports subroutines, virtual memory, and irregular data structures.

The Intel Many Integrated Core (Intel MIC) is a coprocessor architecture incorporating earlier work on the Larrabee architecture. The first MIC prototype was named Knights Ferry, followed by a commercial release codenamed Knights Corner. The product name of the Knights Corner family is Intel Xeon Phi. The Intel Xeon Phi coprocessor is composed of several cores (up to 61 on the 7100 model) with in-order superscalar architecture. The instruction set is based on the classical x86 instruction set enhanced with 64-bit instructions and specific extensions to address SIMD capabilities with large vector operations. Each core has a 512-bit wide VPU and four independent thread contexts, which execute in round-robin. A core has 32 KB of L1 instruction cache and 32 KB of data cache, as well as a private L2 cache of 512 KB with access to all other L2 caches.

Cores are interconnected by a high speed ring-based bidirectional on-die interconnect (ODI). Cache accesses are coherent using a full MESI coherency protocol. Figure 2.4 on the following page gives an overview of the bidirectional DOI of MIC and the memory controllers onto the ring (up to eight).

The Intel Xeon Phi can be seen as a set of hyperthreaded cores that share a global memory organized by chunks, which is not very far from a SMP with shared UMA sys-

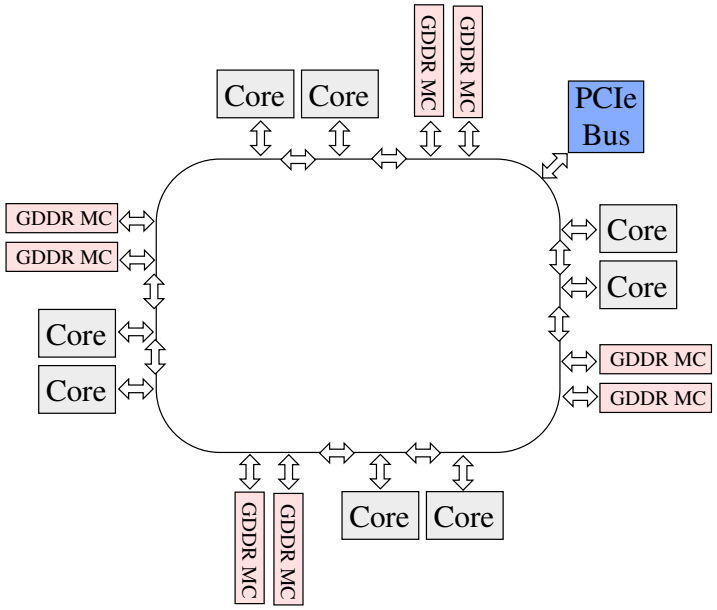


Figure 2.4 – Microarchitecture diagram of Intel Knights Corner, simplified to illustrate the cores interconnected by a bidirectional ring.

tem (Jeffers and Reinders, 2013). Communication between chunks is managed solely by the hardware and transparent to the programmer. In order to illustrate the transparency of MIC architecture, we show the output topology of *hwloc*[1] program for the Intel Xeon Phi coprocessor model 5110P in Figure 2.5 on the next page. Clearly, at user mode level, the system seems a general-purpose SMP platform.

The Intel Xeon Phi runs a modified Linux OS version based on a minimal embedded Linux environment. The libraries include the Linux Standard Base (LSB) Core libraries and a Busybox minimal shell (Jeffers and Reinders, 2013).

2.1.5 A Heterogeneous Machine: Idgraf

Idgraf is composed of two hexa-core Intel Xeon X5650 CPUs (12 CPU cores total) running at 2.66 GHz with 72 GB of memory. It is enhanced with 8 NVIDIA Tesla C2050 GPUs (Fermi architecture) of 448 GPU cores (scalar processors) running at 1.15 GHz each (2688 GPU cores total) with 3 GB GDDR5 per GPU (18 GB total). Figure 2.6 on the facing page illustrates the hardware topology of Idgraf. The machine has 4 PCIe switches to support up to 8 GPUs. When 2 GPUs share a switch, their aggregated PCIe bandwidth is bounded to the one of a single PCIe 16x.

As in Agullo et al. (2011b), we consider the *cumulated peak* from matrix multiplication (`GEMM`) an upper bound on double precision (DP) performance we may obtain using all Idgraf processing units. Our performance results were obtained using ATLAS (version 3.9.39) for CPUs and CUBLAS (CUDA version 5.0) for GPUs. In DP the theoretical peak

[1] `http://www.open-mpi.org/projects/hwloc/`

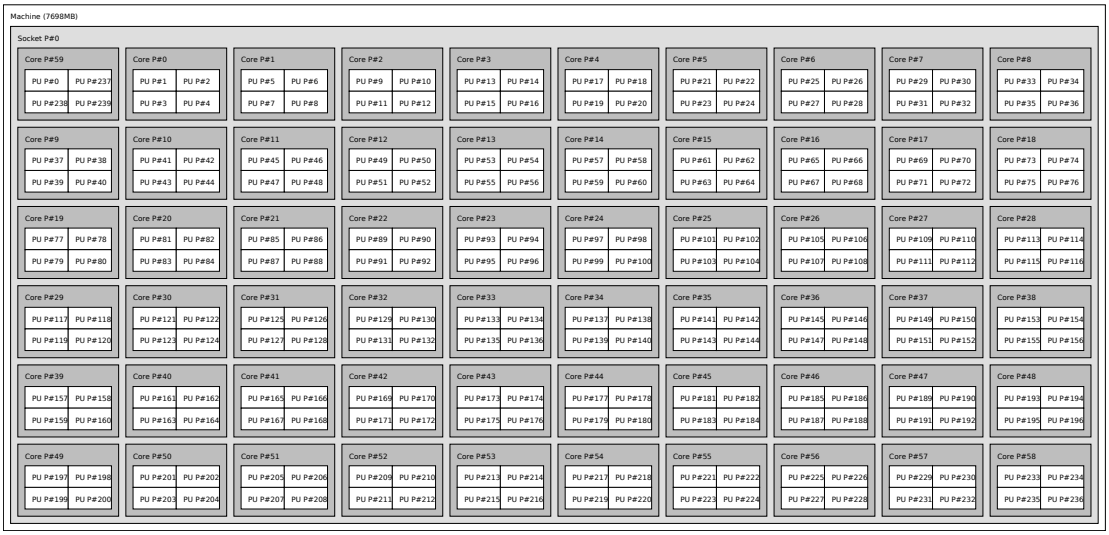


Figure 2.5 – Topology of an Intel Xeon Phi 5110P coprocessor using hwloc.

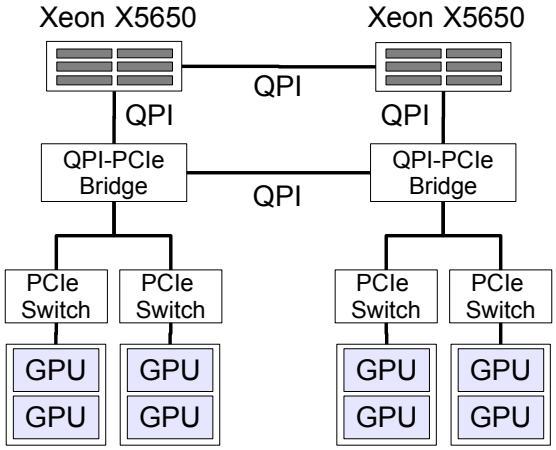


Figure 2.6 – Idgraf hardware topology with two hexa-core CPUs and eight Tesla C2050 GPUs.

of a CPU core is 10.6 GFlop/s (127.2 GFlop/s for all 12 CPU cores) and 515 GFlop/s for a GPU (4.12 TFlop/s for all 8 GPUs). Thus, the DP theoretical peak of Idgraf (all resources) is equal to 4.24 TFlop/s. Furthermore, the DP matrix multiplication peak is 8.9 GFlop/s on a CPU core (`DGEMM` from ATLAS) and 316.4 GFlop/s on a GPU (`DGEMM` from CUBLAS). Hence, the DP *cumulated peak* from `DGEMM` is about 2.6 TFlop/s. Table **??** summarizes the expected and obtained performance results on Idgraf.

2.1.6 Discussion

The trend for more parallelism and power efficient chips was followed by the popularity of accelerators. Recently we can observe the growth of two main types of accelerators for

Table 2.1 – Idgraf peak performance as well as estimated DGEMM performance for all processing units in the system.

System	#units (#cores)	DP peak	DGEMM perf.
Intel Xeon X5650	2 (6)	127.2 GFlop/s	106.8 GFlop/s
NVIDIA Tesla C2050	8 (448)	4120 GFlop/s	2531.2 GFlop/s
Idgraf cumulated (expected)		**4.2** TFlop/s	**2.6** TFlop/s

HPC systems.

The first type is the well-established GPUs for general-purpose programming through programming models CUDA, OpenCL, and OpenACC. It offers hundreds of simple in-order cores and emphasis execution throughput over latency. Despite the evolve of many techniques to reduce the burden of programmers since its earlier models, optimization on GPUs may require a deep knowledge of underling architecture and does not rely only on the usage of large number of lightweight threads. For example, the GPU architecture exposes a memory hierarchy for low latency accesses along with a thread hierarchy in the shape of a grid. Nonetheless, tuning of GPU applications to exploit full potential of such architectural model relies on the programmer.

The second type is the Intel MIC architecture based on x86 ISA enhanced with 64-bit and SIMD instructions. This coprocessor emphasis throughput as well as transparency to the programmer. It provides well-known programming models for general-purpose processors such as OpenMP and Intel Cilk Plus. Thus, the Intel MIC architecture seems to attain transparency for programmers and hide most of architectural details. Regarding optimization techniques, recent experiments on MIC report that performance improvement may come by using classic techniques for CPUs. They include stride-one access, data blocking, data reorganization, false sharing (Dongarra et al., 2003, p. 63) as well as vectorization (Fang et al., 2013a,b; Jeffers and Reinders, 2013; Pennycook et al., 2013).

It seems that the fact of exposing architectural details may dictate the success of a specific accelerator. For instance, Cell processors are a suitable case of an accelerator that may attain significant performance results. However, its programming model restrained its wide acceptance. Successful cases such as GPUs suggest that accelerators play an important role in HPC in order to overcome challenges. They include higher performance results, employment of more applications such as simulation models, and reduce energy consumption without performance lost. The question is if Intel MIC coprocessor would attain such wide acceptance on HPC similar to GPU accelerators.

The rest of this Chapter provides an overview of the main parallel programming models on HPC.

2.2 Programming Models

A parallel programming model is an abstraction of the underlying architecture; in other words, a view of data and execution to developers. A representation of the programming

model's role is the *Parallel Bridge* described by Asanovic et al. (2006) and illustrated in Figure 2.7. It presents the bridge as a balance between *opacity* (right side) and *visibility* of architecture's details (left side) to programmers. Opacity improves architecture's abstraction and obviates the need of programmers to learn the architecture's details; thus, it favors programmer's productivity. On the other hand, visibility shows the key aspects of the underlying architecture for programmers. It allows performance tunning based on architecture's details.

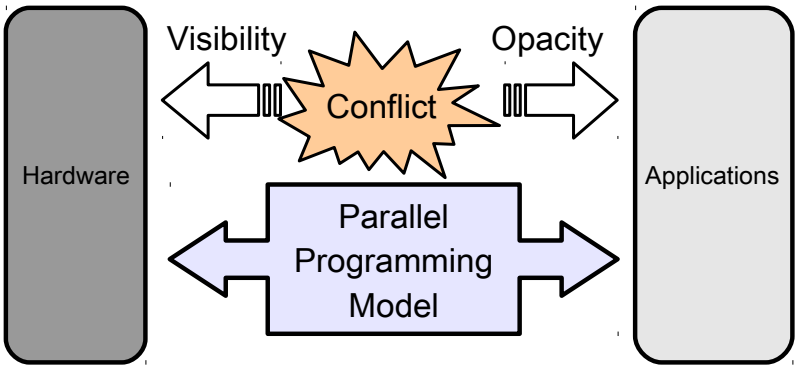


Figure 2.7 – Parallel Bridge between applications and hardware, which the bridge represents the parallel programming models.

Heterogeneous architectures have brought many aspects in the context of parallel programming models. The heterogeneous nature of PUs exposes different parallel programming models at the same time in the context of instructions. Some PUs are efficient at MIMD or task parallelism, in which PUs execute tasks independently and asynchronously. Such PUs are general-purpose processors such as multicore chips. However, other PUs have better performance at SIMD or data parallelism, namely massive-parallel PUs. They execute one single instruction on different pieces of data and do not have task dependencies. Manycore accelerators are a well-known example of such architectures.

The programming model is also responsible to provide a data view of the abstract architecture. There are many data views from different programming models, among them shared-memory and distributed-memory models. Indeed, the advent of accelerators has included a new memory view apart from the previous two classic models. The resulting heterogeneous architecture breaks the shared-memory address space in a different memory space dedicated to the accelerator and accessible by DMA bus requests. Therefore, the memory view from those architectures introduces additional performance parameters and programming directives.

The memory transfers between CPUs and accelerators play a key performance factor, as well as other factors such as bus occupancy and hierarchy. In general, the bus transfer rate is slower than the processing ratio of accelerators resulting a bottleneck to the overall performance. Transfer operations may be reduced to improve the bandwidth and feed accelerators with data to be processed. In the other hand, programming tools for accelerators expose a number of memory characteristics to exploit efficiently the bus and high-speed memory levels as caches.

This Section describes parallel programming models for parallel platforms avoiding spe-

cific programming environments. Five parallel programming models are described based on architectural aspects and decomposing strategy. Details on parallel programming environments and tools are described in Chapter 3 on page 31.

2.2.1 Message Passing

The message-passing model is widely used on parallel computing, which consists of basic mechanisms of send and receive communications. Message-Passing Interface (MPI) and Parallel Virtual Machine (PVM) are the most popular specifications of the message-passing model. Both systems are designed to be portable to a wide range of parallel platforms.

MPI defines a portable interface using either C or Fortran for SPMD and MPMD programs (since MPI-2). It offers since MPI-2 asynchronous messages, collective communications, parallel I/O operations, dynamic process management, multithreaded programs, and remote memory accesses (Gropp et al., 1999). The two basic message-passing primitives are `MPI_Send` and `MPI_Recv` for synchronous messages. The original MPI-1 standard required the number of processes to be specified at startup time as a parameter. The more recent MPI-2 standard supports dynamic task creation, through its dynamic process management, and multithreaded programming.

In addition, Charm++ (Kale and Krishnan, 1993) is a message-driven parallel language implemented as a C++ library. Charm++ parallelism is expressed by collections of objects called *chare* objects, which execute in response to messages received from other chare objects. The adaptive runtime system keeps track of the physical location of chare objects and handles low-level details of the network. Charm++ applications are written to have significantly more chare objects than processors, which allow the overlap between computation and network communication.

2.2.2 Shared Memory

Shared-memory model is widely used in parallel programming for multicore architectures. In this model, it is assumed that all data structures are allocated in a common address space that is accessible from every processor. As described in Section 2.1.2, shared-memory systems may have non-uniform access cost to a memory address.

In shared-memory architectures, communication is implicit since memory is accessible by all processors. For this reason, shared-memory models focus on primitives for expressing concurrency and synchronization along with techniques to minimize overhead (Grama et al., 2003). One exception is data-flow programming models since communication is explicit using data access modes (Gautier et al., 2007). On the other hand, synchronization is implicit and dependencies between tasks are automatically managed by the runtime.

Most of programming tools on this model derivate from multi-threading to express parallelism. Examples of such tools are Pthreads (of Electrical and Electronic Engineers, 1995), OpenMP (Chapman et al., 2007), TBB (Reinders, 2007), Cilk (Frigo et al., 1998) and Cilk++ (Frigo et al., 2009).

2.2.3 Distributed Shared Memory

Distributed shared-memory (DSM) model or Partitioned Global Address Space (PGAS) is a parallel programming model for shared and distributed systems. It is based on a global address space view of the platform. The global memory address space is logically partitioned and a portion of it is local to each thread, called *shared segment*. This shared segment is also accessible by any remote thread, although it may not benefit of data locality. In addition, a thread can use local data named *private segment* that is not included in global address space. PGAS model intends to benefit of the advantages of local-view style and global view of address space. Each memory segment is illustrated in Figure 2.8 where each thread has local access to its private segment and shared segment, which is shared by other threads.

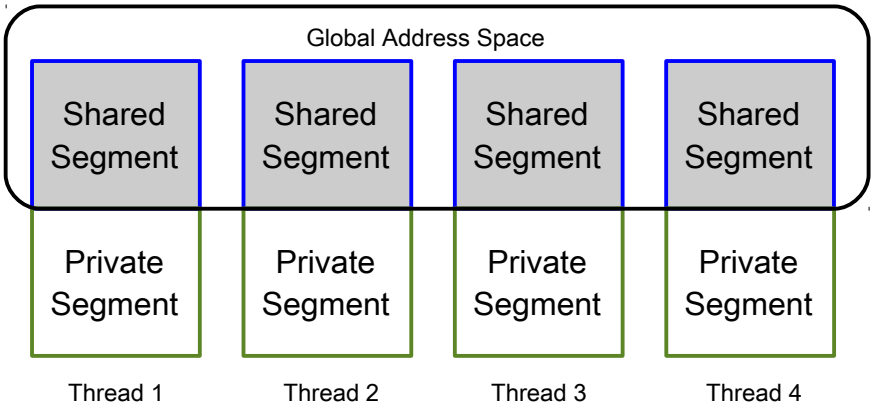


Figure 2.8 – Memory view of PGAS programming model with shared segments composing the global space, and private segments per thread.

A number of PGAS languages are ubiquitous in this model such as Co-array Fortran (Numrich and Reid, 2005), Titanium (Hilfinger et al., 2005), and Unified Parallel C (UPC) (Consortium, 2005).

2.2.4 Data and Task Parallelism

The design of parallel algorithms in general needs to divide the work among the processing elements (PE) (processes or threads) so that each PE gets roughly the same amount of work. Ideally, we also need to arrange the PEs to synchronize and communicate. Foster (1995) describes an outline of four distinct steps to design a parallel program: partitioning, communication, agglomeration, and mapping. *Data parallelism* and *task parallelism* are directly related to the partitioning phase and expose the opportunities for parallel execution. It divides the problem in smaller pieces based on data or computation.

Data parallelism, also named *data decomposition* or *domain decomposition*, is a straightforward and commonly used method to express concurrency in algorithms. In this programming model, the data associated with the problem is partitioned and then mapped to tasks. The decomposed data may be the data input, the output computed, the intermediate data, or owner-computes rule (Grama et al., 2003).

Task parallelism, also *functional parallelism* or *control parallelism*, represents a different and complementary programming model. This strategy decomposes the computation to be performed rather than the manipulated data. This programming model can be employed on tasks that perform different computations and are independent from each other. But task parallelism is commonly used on algorithms that tasks have dependencies and unfold a directed acyclic graph (DAG). When dependency between tasks and data are considered, the algorithm unfolds a data-flow graph (DFG) (Galilee et al., 1998). For instance, recursive algorithms are a straightforward example of task parallelism in which each recursive function call is replaced by a task and a synchronization to wait for produced results if necessary. Another example is parallel loops in which each iteration is mapped to a task without dependencies.

2.2.5 Discussion

The advent of heterogeneous systems brought the use of programming models that expose more aspects of underlying architecture in order to exploit full potential of accelerators. They are independent of the number of processors, but they are based on shared-memory with disjoint address spaces on the same system. While those programming models offer a ways to express large parallelism through multithreading, they impose an explicit memory view.

Asanovic et al. (2006) report some recommendations on designing parallel programming models for parallel systems. They show an interesting analysis concerning various programming models for 5 critical aspects sorted from most explicit (visibility) to most implicit (opacity). The most two implicit models are HPF and OpenMP, which is widely accepted for shared-memory architectures. Hence, it seems that implicit models in most aspects can deliver high performance such as OpenMP (Chapman et al., 2007). Recent versions of OpenMP also include task parallelism (Ayguadé et al., 2009b) from version 3.0 and accelerators (OpenMP Architecture Review Board, 1997-2013) in the new 4.0 version. Still, OpenMP is incipient regarding the memory view because its original goal was shared-memory systems with an uniform memory view, which does not even consider NUMA aspects. Other works similar to OpenMP describe a memory view through data dependencies like Intel Offload Compiler (Newburn et al., 2013), OmpSs (Bueno et al., 2012), and OpenACC (Kirk and Hwu, 2012).

Task parallelism seems to be a well-suited programming model for heterogeneous architectures. Parallelism is explicit, while all other aspects pointed by Asanovic et al. (2006) are implicit and relies on the underlying runtime system. Besides, this model favors fine granularity and asynchronicity that are essential in order to expoit parallelism and improve scalability in modern multicore and manycore architectures (Buttari et al., 2009).

2.3 Scheduling Algorithms

Scheduling is concerned with the allocation of scarce resources to activities, or tasks, with the objective of optimizing one or more performance metrics. Resources may be machines, CPU, memory and I/O devices, etc. There are also many performance metrics to optimize such as minimization of makespan, or minimization of the number of late tasks. This Sec-

tion describes algorithms that relate the following load balancing variables: data locality, heterogeneous computing resouces, and ocupancy.

We will focus on the task parallelism programming model that unfolds parallelism by means of fork/join constructs. We assume that algorithms create tasks by *spawn* construct and synchronization execution by *sync* construct, similar to Cilk programming extensions (Blumofe et al., 1995). We first describe two algorithms over multicore processors (work stealing) and accelerators (HEFT), then we present related works on work stealing on GPUs, and online scheduling for heterogeneous systems.

2.3.1 Work Stealing

Work stealing is a distributed list scheduling algorithm with receiver-initiated load balancing. A worker (the thief) who runs out of work randomly selects another worker (the victim) from whom to steal work. Cilk (Blumofe and Leiserson, 1998; Blumofe et al., 1995; Frigo et al., 1998) is a well-known programming tool that implements work stealing with provably efficiency for a number of P homogeneous processors. Throughout this section, we use the Cilk's work stealing as reference.

In general, each processor, called *worker*, has a *ready deque* (doubly-ended queue) of ready tasks. Each deque has two ends, a *head* and a *tail*, from which tasks can be added or removed. At first, a worker tries to remove a task from the tail of its deque. When a worker runs out of work, it becomes a *thief* and attempts to steal a task from another worker, called its *victim*. The thief steals tasks from the head of the victim's deque, the opposite end from which the victim is working. A worker adds and removes tasks from the deque's tail, while thieves may work at the opposite end.

The work stealing algorithm has theoretical lower bounds to *fully strict* computations, whose each task has a direct edge to its parent task. The execution time is constrained by two parameters: *work* and *critical path*. The work, denoted T_1, is time by one-processor execution. The critical path length, denoted T_∞, is the total execution time required by an infinite-processor execution. The expected execution time, including scheduling overhead, is bounded by

$$T_P = T_1/P + O(T_\infty), \tag{2.1}$$

assuming an ideal parallel computer. Work stealing guarantees with high probability that only $O(PT_\infty)$ steal attempts occur ($O(T_\infty)$ on average for each processor). All these costs are borne on the critical path.

The described work stealing applies a depth-first execution of tasks for two main reasons. First, steals of top level tasks may contain larger amounts of work than lower levels. Stealing large amounts of work tends to lower synchronization costs, because fewer steals are necessary. Second, tasks at the tail of the deque are also the ones in lower levels in the DAG. Therefore, if processors are idle, the work they steal tends to make progress along the critical path.

Work stealing is efficient for recursive and fine-grained algorithms. However, classical work stealing does not consider locality (Acar et al., 2000; Guo et al., 2010) due to randomized stealing nor heterogeneous processors (Bender and Phillips, 2007).

The Work-First Principle

Frigo et al. (1998) describe the following principle:

> *The work-first principle*: Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path.

Let us denote by T_s the running time of a sequential program and the *work overhead* by $c_1 = T_1/T_s$. According to the work-first principle, incorporating critical path and work overheads into Equation 2.1 yields

$$T_P \leq c_1 T_1/P + c_\infty T_\infty \approx c_1 T_s/P, \tag{2.2}$$

under the condition of "parallel slackness", that is, the average parallelism exceeds the number of processors. The work-first principle states that minimize c_1 is important, even at the expense of a larger c_∞, since c_1 has direct impact on performance. The principle may affect scalability if c_∞ is too large.

Work-First *versus* Help-First

The work-first implementation states that a worker executes a newly spawned task and leaves the continuation (current task) to be stolen by another worker (Frigo et al., 1998). On the other hand, the help-first policy dictates that a worker executes the continuation and leaves the spawned tasks to be stolen.

Guo et al. (2009) report that the work-first policy is designed for scenarios in which steal is a rare event, and its steal overhead becomes significant as the number of workers increases. They give an analysis for iterative computations with each iteration creating a new task. Using the total amount of work in one iteration of a parallel loop T, the time to migrate a task from one worker to another t_{steal} and the time to save the continuation for each iteration t_{cont}. Under the work-first policy, one worker will save and push the continuation onto the local deque and another idle worker may steal it. Thus, distributing P chunks among P workers will require $P-1$ steals and these steals must occur sequentially, considering the classic work stealing. In the THE protocol (Frigo et al., 1998), a push operation on the deque is lock-free but a thief must acquire the lock on the victim's deque, thus $t_{cont} \ll t_{steal}$. As P increases, the actual work for each worker T/P decreases and the total time will be dominated by the time to distribute tasks, which is $O(t_{steal}P)$.

A help-first policy can be space-efficient for stack size assuming that the required space to save a continuation is greater than for a spawned task. Besides, let us assume t_{task} the time to create and push a task onto the local deque, the inequality $t_{task} \ll t_{steal}$ holds. However, the work-first policy can be more space-efficient than help-first in Cilk "two-clone" strategy (Frigo et al., 1998). Each parallel procedure has two versions: a "fast" clone and a "slow" clone. The fast clone executes in sequential semantics and has little support to parallelism. When a task is spawned, the fast clone runs. Whenever a thief steals a task, it runs the slow clone.

We will use two cases to describe the space requirements of help-first and work-first: a parallel loop with each iteration creating a new task, and a recursive computation. In

the parallel loop, work-first is space-efficient because it will not push the created task at each iteration. Instead, it will execute the fast clone in sequential semantics. A help-first strategy will push all tasks of the loop and may cause a stack overflow. On the other hand, a recursive computation with work-first will push the continuation at each recursion level and may cause stack overflow.

The analysis from Guo et al. (2009) only considers the implied overhead of each strategy, in which it concludes that help-first has less overhead than work-first. However, it lacks of an accurate analysis of space requirements, which depends on the parallel algorithm.

2.3.2 Heterogeneous Earliest-Finish-Time

Topcuoglu et al. (2002) describe the Heterogeneous Earliest-Finish-Time (HEFT) algorithm, which is a scheduling algorithm for a bounded number of heterogeneous processors. We describe in this Section the problem scope of HEFT, and the scheduling algorithm.

Scheduling Problem

The HEFT scheduling defines the application, the target computing environment, and the performance criteria of scheduling. The application is represented by a DAG $G = (V, E)$ where V is the set of v tasks and E is the set of e edges between the tasks. Each edge $(i, j) \in E$ represents the precedence constraint. The computing environment consists of a set Q of q heterogeneous processors connected in a fully connected topology.

Before presenting the objective function, it is necessary to define the EST and EFT attributes, which are derived from a given partial schedule. $EST(n_i, p_j)$ is the *earliest start time* and $EFT(n_i, p_j)$ is the *earliest finish time* of task n_i on processor p_j. For the entry task n_{entry} of a given task graph,

$$EST(n_{entry}, p_j) = 0. \tag{2.3}$$

Given $w_{i,j}$ the estimated execution time of task n_i on processor p_j, and $c_{i,k}$ the communication cost of the edge (i, k) which is to transfer data from task n_i (scheduled in p_m) to task n_k (scheduled in p_n). For each task in the graph, the EFT and EST values are computed recursively starting from the entry task, as shown in 2.4 and 2.5. In order to compute the EFT of a task n_i, all immediate predecessors tasks of n_i must have been scheduled.

$$EST(n_i, p_j) = max\{avail[j], \max_{n_m \in pred(n_i)}(AFT(n_m) + c_{m,i})\}, \tag{2.4}$$

$$EFT(n_i, p_j) = w_{i,j} + EST(n_i, p_j), \tag{2.5}$$

where $pred(n_i)$ is the set of predecessor tasks of task n_i and $avail[j]$ is the earliest time at which processor p_j is available for task execution. If n_k is the last assigned task on processor p_j, the $avail[j]$ is the time that processor p_j completed the task n_k and is available to execute another task. The inner max block in the EST equation returns the ready time, *i.e*, the time when all data needed by n_i is available at processor p_j.

The objective of HEFT is to minimize the makespan (C_{max}), or the schedule length, of a task graph. After a task n_m is scheduled on a processor p_j, the earliest start time and the earliest finish time of n_m on p_j is equal to the actual star time, $AST(n_m)$, and the actual finish time, $AFT(n_m)$, of task n_m, respectively.

HEFT Algorithm

The algorithm has two major phases: a *task prioritizing* phase for computing the priorities of all tasks and a *processor selection* phase for select the tasks in the order of their priorities and scheduling each task on its "best" processor, which minimizes the task's finish time.

The search of an appropriate idle time slot of task n_i on a processor p_j stars at the time equal to the ready time of n_i on p_j, which is the time when all input data of n_i have arrived at processor p_j. HEFT has an $O(eq)$ time complexity for e edges and q processors. The algorithm is depicted in Algorithm 1.

Algorithm 1: Heterogeneous Earliest-Finish-Time.

```
1 Compute task prioritizing for all tasks by transversing graph upward, starting from
  the exit task
2 Sort the tasks in a scheduling list by nonincreasing order of priority values
3 while there are unscheduled tasks in the list do
4 |   Select the first task, n_i, from the list of scheduling
5 |   foreach processor p_k in the processor set (p_k ∈ Q) do
6 |   |   Compute EFT(n_i, p_k) value using the insertion-based scheduling policy
7 |   end
8 |   Assign task n_i to the processor p_j that minimizes EFT of task n_i
9 end
```

2.3.3 Algorithms for Heterogeneous Systems

Many works in the literature propose scheduling strategies evolving accelerators. We split this Section in two groups of related works: strategies restricted to GPUs, and online scheduling on runtime systems.

Work Stealing on GPUs

Cederman and Tsigas (2008) compared four load balacing strategies to distribute thread blocks among workers (streaming processors) of a GPU. The strategies are based on a centralized queue with blocking operations, centralized queue with no-blocking operations, centralized queue with static distribution, and task stealing. Task stealing has modifications regarding the work stealing scheduler. The authors assume an initial work distribution among workers and when they run out of work they try to steal from others. The victim selection is performed in a round-robin fashion. According to their findings, the task stealing with non-blocking operations was able to scale and outperformed static distributions.

Zhou et al. (2009) designed a Reyes renderer, called RenderAnts, using work stealing on multi-GPU. In order to reduce inter-GPU transfers, they duplicate some computations and send all scene data to all GPUs at the beginning of the pipeline. Task stealing follows a recursive data splitting scheme with adaptive granularity. When a subregion has more than a given threshold of computing primitives and any other GPU is idle, the subregion is split.

Toss and Gautier (2012) study a lock-free work stealing scheduler embedded into a GPU. It exploits two levels of parallelism: the first level with independent and coarse subproblems, and the second level with finer threads that cooperate for the same computation. Each worker is a SM with a work queue of first-level tasks. When a worker runs out of work, it tries to steal randomly from another worker.

Online Scheduling

We relate here the works on scheduling algorithms for heterogeneous architectures. Most of the strategies are designed in conjunction with runtime systems, whose we detail in Chapter 3 on page 31.

Augonnet et al. (2010a) study strategies based on the HEFT scheduling algorithm. They compare the impact of execution time (*heft-tm*) in addition to data transfers (*heft-tmdp*) and and data prefetch (*heft-tmdp-pr*). The prefetch policy makes input transfers in advance at the moment when a task is scheduled to a worker.

Ayguadé et al. (2009a) describe the StarSs runtime system for multi-GPUs, named GPUSs. It schedules tasks by centralized list scheduling according to the available GPUs. From those available GPUs, the scheduling strategy maps tasks to the GPU that minimizes data transfers. Recent versions of the GPUSs runtime called OmpSs (Bueno et al., 2011, 2012) extend scheduling strategies by two policies: centralized with first-in first-out (FIFO) strategy, and locality-aware. The locality-aware strategy of OmpSs computes a locality score for each GPU at task creation. This score is based on the data on each worker for execution of the new task. The main difference of these strategies is the use of a local queue and a global queue, which contains tasks with no affinity score. An idle worker looks for tasks into the local queue, then the global queue, and last it tries to steal work from another worker. The strategies proposed by Ayguadé et al. (2009a), Bueno et al. (2011), and Bueno et al. (2012) have the limitation of strict usage of GPUs, while CPUs are involved only in runtime routines.

Planas et al. (2013) study a scheduling strategy with task versioning similar to StarPU on multi-CPU and multi-GPU systems. The scheduler at first distributes tasks in round-robin to calibrate the performance of tasks for each worker. Next, it uses a HEFT approach to reduce the finish time of tasks. The only novelty of this work is the use of multi-version implementation for a target architecture. The programmer can give multiple task implementations for a given architecture and the runtime will choose the most efficient depending on the target worker.

Hermann (2010) proposes a static and dynamic scheduling for iterative computations on multi-CPU. Based on the work of Laurent Pigeon (Pigeon, 2007), it expresses a loop in a task graph, partitioned and redeployed at each iteration. First it applies a static partitioning to group related tasks in the same partition and to create partitions that are weakly dependent. After this initial partitioning, a work stealing approach is applied. The stealing of tasks is decomposed in two phases: a partition level stealing, and a task level stealing. An idle worker visits each worker trying to steal a ready partition to execute. If it fails, the worker will try to steal a ready task.

The static and dynamic scheduling enforce data locality and reduce the total number of steals. From this multi-CPU strategy, the multi-GPU has few changes (Hermann et al.,

2010). The initial partitioning tries to minimize communications between workers. In addition, to overcome the processing power differences of CPUs and GPUs, it considers the execution time of each partition. Each partition has a ratio T_{CPU}/T_{GPU} that is constantly updated taking into account the execution time from the previous iteration. Partitions with T_{CPU}/T_{GPU} ratio below a given threshold execute on CPUs, otherwise on GPUs. Since task stealing between partitions can lead to expensive memory transfers, the second phase (task level stealing) is replaced by a locality guided work stealing based on the work of Acar et al. (2000). Each partition has a *affinity list* of workers: a partition owned by a given worker has an other distant worker in its affinity list if and only if this distant worker holds at least one task that interacts with the partition.

2.3.4 Discussion

Several authors have studied work stealing for shared-memory (Acar et al., 2000; Blumofe and Leiserson, 1998; Blumofe et al., 1995; Frigo et al., 1998; Gautier et al., 2013a; Hermann, 2010; Roch et al., 2006; Tchiboukdjian et al., 2010a,b; Traoré et al., 2008) and distributed (Blumofe and Lisiecki, 1997; Dinan et al., 2009; Gautier et al., 2007; Guo et al., 2009; Min et al., 2011; Nieuwpoort et al., 2000, 2001; Quintin and Wagner, 2010; Ravichandran et al., 2011) systems. In addition, some scheduling strategies based on work stealing are used for GPU and multi-GPU load balancing.

On the other hand, scheduling on heterogeneous systems does not have studies on dynamic strategies. Most common strategies rely on cost models such as HEFT (Augonnet et al., 2010a) or centralized scheduling (Ayguadé et al., 2009a; Bueno et al., 2011, 2012). We note that both approaches have drawbacks. A cost model scheduling depends on regular computations and does not adapt to load variations at runtime. Besides, a centralized scheduling does not scale as the number of workers increase, and is not suitable for fine-grained computations since a single queue of tasks may become a bottleneck.

2.4 Summary

Despite the evolve of many techniques to reduce the burden of programmers, tuning of GPU applications to exploit full potential of such architectural model relies on the programmer. Successful cases such as GPUs suggest that accelerators play an important role in HPC in order to overcome challenges. Nonetheless, the Intel Xeon Phi coprocessor, based on x86 ISA, emphasis throughput as well as transparency to the programmer. It provides well-known programming models for general-purpose processors such as OpenMP and Intel Cilk Plus.

Task parallelism seems to be well-suited programming model with explicit parallelism. Its association with data-flow dependencies offers a memory view independent of underlying architecture. In addition, it allows to loose synchronization that is an essential aspect to exploit parallelism and improve scalability on architectures with accelerators (Buttari et al., 2009; Hermann, 2010).

In the context of scheduling, little research has been done on dynamic scheduling for heterogeneous architectures. Although work stealing is an efficient strategy with theoretical

performance guarantee, it has also been known to be cache-unfriendly for some applications due to randomized stealing.

Chapter 3

Programming Environments

The increasing usage of multicore processors and manycore accelerators in HPC evidences the need to rely on programming models that abstract the underlying architecture, in memory and processor view. Ideally, an application may unfold parallelism oblivious of memory levels and available processors to compute. A programming model well-suited for this purpose is task parallelism, described in the previous Chapter, that has explicit parallelism and may have uniform memory view considering data dependencies to construct a data-flow graph (DFG) (Galilee et al., 1998). Therefore, a task will execute only when data input is produced.

To exploit the potential of heterogeneous architectures, a runtime system plays an essential role. It may provide a programming model and balance workload among processing units at runtime. Besides, if tasks have data dependencies, a parallel program may be oblivious to data transfers between disjoint address spaces since it describes intrinsically the necessary transfer by data access modes.

We are interested here in programming environments for parallel programming over heterogeneous machines improved with manycore accelerators. This Chapter first overviews programming tools for shared-memory systems whose programming model includes tasks parallelism (Section 3.1). We then present programming tools that target GPUs or Intel Xeon Phi accelerators (Section 3.2).

3.1 Shared Memory Programming

3.1.1 OpenMP

OpenMP is a standard API for shared-memory parallel programming. It consists of a set of compiler directives, library routines, and environment variables for building multithreaded parallel applications (Chapman et al., 2007). A main concept of OpenMP is incremental parallelism through the addition of *parallel regions*. The programmer adds parallel directives to the sequential code. In parallel regions, the program forks additional threads to form a team of threads. The threads execute in parallel across a region of code and, at the end, wait until the full team reaches this point and then join back together.

A parallel region does not distribute the work by itself, being necessary the use of *work-sharing* directives to specify how the work is to be shared among the executing threads. Probably, the most common work-sharing directive is the `for` loop. Version 3.0 of OpenMP introduced dynamic task creation through `task` construct, which specifies an unit of parallel work as an *explicit task* (Ayguadé et al., 2009b), and synchronization by `taskwait` construct. Figure 3.1 on the following page shows an example of the Fibonacci sequence using the `task` directive to create tasks dynamically.

```
1 void TaskFibo ( int* ptr, int n ) {
2   if (n < 2) {
3     *ptr = n;
4   } else {
5     int res1 =0;
6     int res2 =0;
7 #pragma omp task untied shared(res1) firstprivate(n)
8     TaskFibo( &res1, n-2);
9 #pragma omp task untied shared(res2) firstprivate(n)
10    TaskFibo( &res2, n-1);
11 #pragma omp taskwait
12    *ptr = res1 + res2;
13  }
14 }
```

Figure 3.1 – Example of the Fibonacci computation with OpenMP tasks.

In addition, OpenMP specifies data-sharing clauses to describe data access by all concurrent threads such as `shared` (shared by all threads) and `private` (local to each thread). The `untied` directive tells to the runtime a new task can be executed by any thread worker.

3.1.2 Cilk and Cilk++

Cilk is a multithreaded runtime system that extends the C language with simple keywords to create and synchronize tasks. The programmer is responsible to expose parallelism and exploit locality, and the runtime system is in charge of scheduling tasks on the target platform.

The Cilk language has three keywords: `cilk` identifies a *Cilk procedure* that is the parallel version of a C function; `spawn` executes a Cilk procedure in parallel; and `sync` is a local barrier that suspends the procedure until all of its children have completed. Data dependencies are implicit through return values and other values sent from one task to another, which is called *fully strict* computation (Blumofe et al., 1995).

One classic example of a Cilk code is the recursive implementation of the nth Fibonacci number computation. The sequential reference algorithm is shown in Figure 3.2a, and Figure 3.2b shows the Cilk version. It is not the optimal version of this algorithm, yet it is a simple way to show how we can profit of parallelism with few changes in the code. A Cilk program has the same semantics as the C program when Cilk keywords are deleted, whose this C program is called *serial elision* or *C elision*.

In a similar way, Cilk++ is a C++ extension to express parallelism through keywords `cilk_spawn`, `cilk_sync`, and `cilk_for` (Leiserson, 2009). The last keyword (`cilk_for`) is an additional feature over classic Cilk extensions allowing parallel loops that are automatically parallelized through a recursive divide-and-conquer strategy. Besides, Frigo et al. (2009) describe the concept of Cilk++ *hyperobjects* to avoid determinacy races in code with non-local variables. An example of hyperobject type is reducers capable to provide a reduction mechanism without explicit locking of atomic updating.

(a) A sequential C program.

```
1 int fibonacci( int n )
2 {
3   int x, y;
4   if( n < 2 )
5     return n;
6
7   x = fibonacci( n-1 );
8   y = fibonacci( n-2 );
9
10
11  return (x+y);
12 }
```

(b) A parallel Cilk program.

```
1 cilk int fibonacci( int n )
2 {
3   int x, y;
4   if( n < 2 )
5     return n;
6
7   x = spawn fibonacci( n-1 );
8   y = spawn fibonacci( n-2 );
9   sync;
10
11  return (x+y);
12 }
```

Figure 3.2 – Recursive implementation of the Fibonacci number computation (left) compared to its Cilk version (right). Each recursive call with `spawn` creates a new task asynchronously. The `sync` keyword is used to wait the previously created tasks and sums their result.

Cilk and Cilk++ follow the same implementation philosophy described by Frigo et al. (1998). The compiler generates two clones of the same Cilk procedure: a *fast* clone and a *slow* clone. The fast clone is similar to a C procedure with few modification, while the slow clone has full support for parallelism. When a task is spawned, the fast clone runs. Whenever a thief steals a task, however, the task is converted into its slow clone. Cilk runtime expects that the number of steals is small and the fast clones to be executed most of the time.

The Cilk scheduler uses a work stealing strategy for load balancing. Idle processors, called thieves, "steal" threads from busy processors, called victims. Each worker owns a *spawn deque* – a double-ended queue (Cormen et al., 2009, p. 236) – in that the worker can insert or remove tasks on the *tail* end of its deque, but other workers ("thieves") can only remove from the *head* end. In its implementation, Frigo et al. (1998) propose the THE protocol to resolve the race condition when a thief tries to steal the same task that its victim is attempting to pop. Its key idea is that operations by the worker on the tail of its deque contribute to the work overhead, while operations by thieves on the head contribute only to the critical path overhead, in accordance to the work-first principle introduced in Section 2.3.1 on page 23.

3.1.3 Threading Building Blocks

Intel *Threading Building Blocks* (TBB) (Reinders, 2007) is a C++ runtime library without compiler support or language extensions. Its parallelism can be expressed in terms of tasks, which are represented as instances of a task class, or using concurrent container classes through generic interfaces. Most of TBB programming interface is focused on data-parallel programming relying on generic programming. The development of Intel TBB is strongly inspired by Cilk, mostly on its scheduling strategy. TBB uses work stealing to redistribute

the work across processors.

An Intel TBB implementation of the nth Fibonacci number is shown in Figure 3.3. The `execute` method represents the task code executed by the library, which spawns two new tasks and waits for their results. Its C++ programming interface requires much more code to express the same functionality than compiled languages such as Cilk or OpenMP.

```
1 class TaskFibo: public tbb::task {
2 public:
3   const int n;
4   int* const sum;
5
6   TaskFibo ( int n_, int* sum_ ) : n(n_), sum(sum_) {}
7
8   task* execute() {
9     if ( n < 2 ) {
10      *sum = n;
11    } else {
12      int x, y;
13      TaskFibo& a = *new (allocate_child())TaskFibo( n-1, &x );
14      TaskFibo& b = *new (allocate_child())TaskFibo( n-2, &y );
15      set_ref_count(3);
16      spawn (b);
17      spawn_and_wait_for_all(a);
18
19      *sum = x+y;
20    }
21
22    return NULL;
23  }
24 };
```

Figure 3.3 – Example of an Intel TBB task of the Fibonacci sequence.

3.1.4 Athapascan/KAAPI

Athapascan is a parallel programming interface developed by the MOAIS project from the INRIA/LIG. It relies on task parallelism and data dependencies expressed using explicitly shared data types (by default all data are local). Like Cilk, it supports recursive parallelism to express divide and conquer algorithms. Additionally, Athapascan can exploit non-recursive parallelism by means of data dependency analysis. The Athapascan syntax and semantic are described by Galilee et al. (1998) and the technical report of Roch et al. (2003) gives the complete programming API.

A runtime environment, called KAAPI, offers an abstraction of the hardware architecture, providing an uniform interface for parallel programming and communication (Gautier et al., 2007). The load balancing uses a work stealing approach like done by Cilk,

where an idle processor steals work from busy processors. It also offers fault tolerance and distributed-memory support.

Programming Model

The Athapascan programming model relies on a DFG of task dependencies using the notion of tasks and shared data. The high level API used to express data dependencies is inspired by the Jade Parallel Programming Language (Rinard et al., 1993). Athapascan has basically two keywords: `Shared` and `Fork`. The `Shared` keyword declares an object in the global memory, which can be accessed by any processor. The `Fork<`*Task*`>` keyword creates a new parallel task *Task*, similarly to the spawn construct of Cilk.

A task in Athapascan is a function whose signature contains all the data it shares with other tasks. This signature also contains the access mode of each parameter : a read-only parameter is declared using `Shared_r` while a write-only data is declared as `Shared_w`. The program execution is driven by the data availability, according to these access modes. A task requesting a read access must wait for all the previous writers before starting the execution.

Combining recursive and data driven execution one can express more complex structures and better exploit the parallelism of algorithms. One example using the Fibonacci number computation is shown in Figure 3.4. As done in Cilk, the Fibonacci number is computed recursively, and tasks are spawned to compute each branch. However, the execution is controlled by the DFG and not by the program recursion structure of the full strict model (Section 3.1.2 on page 32).

The unrolling of the DFG is illustrated in Figure 3.5 on page 37 through the Fibonacci number computation (Figure 3.4 on the next page). The initial task is called by the main thread and receives the output buffer as parameter (Figure 3.5(a)). When the initial tasks are executed they create other tasks recursively (Figures 3.5(b) and 3.5(c)). Inside the `Fibonacci` function the tasks are linked by data dependencies. For instance the `Sum` tasks wait the end of both preceding `Fibonacci` functions.

Using DFG offers a more flexible parallel programming model compared to Cilk and Intel TBB fully strict model. The Athapascan model allows a synchronization between tasks that are at the same recursion level. For instance, tasks `Sum` and `Fibonacci` have a data dependency to compute the sum of both `Fibonacci` tasks. To support synchronization between tasks at a same level, TBB and Cilk provides a synchronization task, which forces the application to wait for the end of all the previous spawned tasks. This synchronization is used to wait for all the previous `Fibonacci` tasks before summing them. In Fibonacci computation, this synchronization call does not impair performance since in all the cases the summing depends on all the previous Fibonacci tasks.

KAAPI runtime structure

KAAPI is a runtime environment that implements the Athapascan API. It offers an abstraction layer for the underlying hardware, along with data sharing, fault tolerance, and task scheduling. Among the KAAPI functionalities this Section will be restrained to the programming model and scheduling aspect.

```
1 struct Sum {
2   void operator()( a1::Shared_w<int> res,
3                    a1::Shared_r<int> a,
4                    a1::Shared_r<int> b) {
5     res.write( a.read() + b.read() );
6   }
7 };
8
9 struct Fibonacci {
10  void operator()( a1::Shared_w<int> res, int n ) {
11    if( n < 2 )
12      res.write( n );
13    else {
14      a1::Shared<int> res1, res2;
15      a1::Fork<Fibonacci>()(res1, n-1);
16      a1::Fork<Fibonacci>()(res2, n-2);
17      a1::Fork<Sum>() (res, res1, res2);
18    }
19  }
20 };
21
22 void fibonacci( unsigned int n ) {
23   a1::Shared<int> res;
24   a1::Fork<Fibonacci>()( res, n );
25   a1::Fork<Print>()( res ); /* Print the result */
26 }
```

Figure 3.4 – Recursive version of the Fibonacci sequence with Athapascan interface.

Figure 3.6 on the facing page shows the hierarchical structure of KAAPI. One KAAPI process is launched per computing node. Then, each KAAPI process is composed by system threads called *K-Processors*. Usually there is one K-Processor per processor in the machine. The K-Processors execute the user level threads implemented by KAAPI. These lightweight threads are called *K-Threads*, and contain a stack of tasks spawned by the user. The K-Threads are non preemptive, *i.e.*, their execution cannot be interrupted by the runtime environment. This kind of hierarchical structure is used because creating and switching context among KAAPI threads is much faster than switching context on system threads.

3.2 Heterogeneous Architectures

This Section describes parallel programming models available in the literature for heterogeneous architectures. It begins with the primarily programming models for GPUs CUDA, OpenCL, and OpenACC. After, we show an overview of runtime tools such as Charm++, StarPU, OmpSs, KAAPI, and Intel Xeon Phi software.

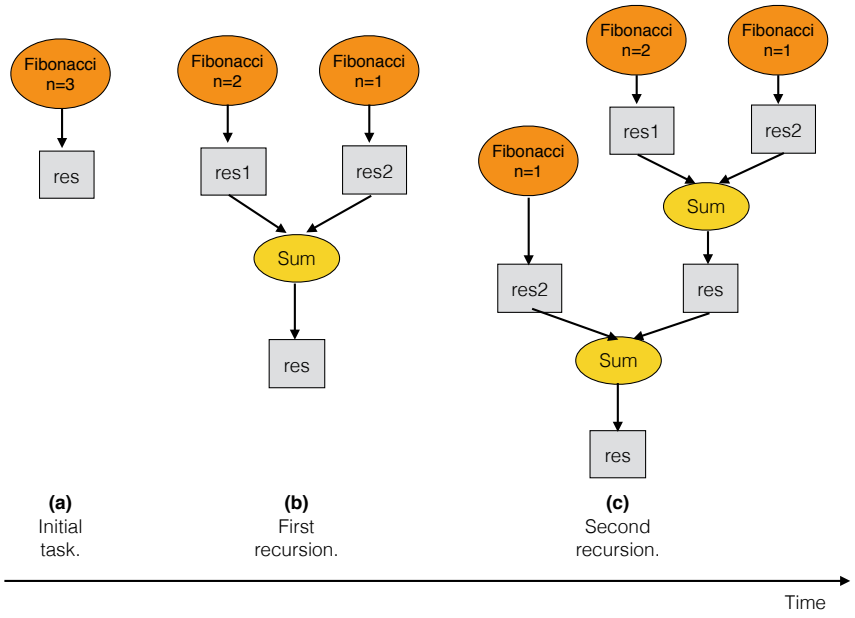


Figure 3.5 – Fibonacci dependency graph on Athapascan model.

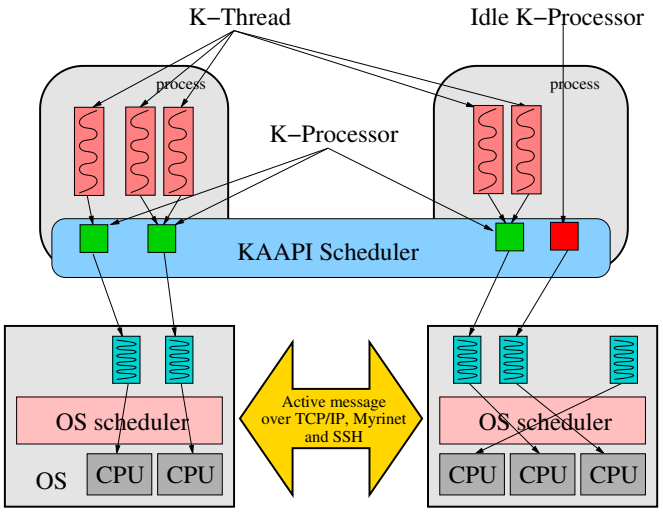


Figure 3.6 – KAAPI runtime structure.

3.2.1 CUDA, OpenCL, and OpenACC

In this Section we overview three well-known programming tools for parallel programming on GPU accelerators. While NVIDIA CUDA is a proprietary tool for NVIDIA GPUs, OpenCL and OpenACC are open specifications that may have multiple implementations.

The **Compute Unified Device Architecture (CUDA)** is a general-purpose parallel programming model for NVIDIA GPUs. A CUDA program consists of one or more phases that are executed on either the host (CPU) or a GPU device in a fork-join strategy like

OpenMP parallel regions (Chapman et al., 2007, p. 24). The phases that exhibit little or no data parallelism are implemented in the host code, while whose phases rich in data parallelism are implemented in the device code (Kirk and Hwu, 2012). The NVIDIA C Compiler (`nvcc`) separates the two at compile time. The host code is ordinary C/C++ code, while the device code is an extended ANSI C with keywords for labeling data-parallel functions, called *kernels*, and their associated data structures.

The parallelism is expressed by a hierarchy of thread groups. A *CUDA thread* is a lightweight unit of execution that is identified using a thread index that can be one-dimensional up to three-dimensional forming a *thread block*. There is a limit to the number of threads per block; however, a kernel can be executed by multiple equally-shaped thread blocks. Blocks are organized into a one-dimensional up to three-dimensional *grid* of thread blocks.

In CUDA, the host and devices have separate address spaces. The host allocates GPU memory with `cudaMalloc()` calls and transfers data to and from the GPU using `cudaMemcpy` calls. The GPU address space has three memory levels:

- *Local memory* is read-write and private to each thread;
- *Shared memory* is read-write and shared per-block thread;
- *Global memory* is read-write and accessible to all threads.

It also offers a subset of the texturing hardware that the GPU uses for graphics to access texture and surface memory.

CUDA supports a number of features such as asynchronous execution, concurrency with streams, event monitoring, unified virtual address space (UVA), and multi-device support.

The **Open Computing Language (OpenCL)** is a standard framework that is managed by the Khronos Group and offers both a device-side language and a host management layer for devices in a system (Gaster et al., 2012). It provides task-based and data-based parallelism, whose device-side language is designed to efficiently map to a wide range of devices. This device-language, OpenCL C, is a restricted version of the C99 language with extensions appropriate to execute data-parallel code on specialized processor devices.

The unit of concurrent execution in OpenCL is a *work-item*, which can be mapped to dimensions of input or output data as a *n-dimensional range* (NDRange). Work-items can be organized in groups of equally-sized *workgroups*. An index space of N dimensions requires workgroups of N dimensions; thus, a three-dimensional index space requires three-dimensional workgroups. Work-items within a workgroup can perform barrier operations to synchronize and have access to a shared memory address space, in a similar way to the CUDA thread blocks. On the host side, the *context* container acts as a mechanism for host-device interaction. In addition, communication with a device occurs by submitting commands to a *command queue* on the host.

Due to its support to multi-platform and multi-vendor portability, OpenCL has a more complex programming API and device management (Kirk and Hwu, 2012, p. 298). While the OpenCL standard is designed to support portability across different devices, such portability does not come for free. A portable OpenCL code will need to avoid optional features that may allow applications to achieve more performance.

The **OpenACC** API provides a set of compiler directives and library routines to parallel programs on accelerators (Kirk and Hwu, 2012, chap. 15). Its programming model relies on compiler pragmas (`#pragma acc`) with directives for incremental development of parallel programs. An OpenACC program has an execution strategy similar to OpenCL and CUDA with a *parallel region* or *kernel region* to create and execute code in an accelerator device.

Parallelism in an OpenACC device is organized as a hierarchy where a *worker* is an execution thread and a group of workers is a *gang*. The OpenACC memory model treats host and device memory as separated. Unlike CUDA, programmers do not need to code explicitly data movements, they can just annotate which memory object need to be transfered. The *data clauses* in OpenACC are `copy_in` (from host to device), `copy_out` (from device to host), and `copy` (to copy in both directions).

More recently, OpenACC is in process of integration with OpenMP 4.0 to provide OpenMP support for accelerators. The OpenACC founding members are part of the OpenMP working group for accelerators (Kirk and Hwu, 2012, p. 337).

3.2.2 Charm++

Jetley et al. (2010) extended the Charm++ runtime system to support GPUs with the *Charm++ GPU Manager*. It allows the overlap of chare objects with the use of different CUDA streams for every chare executing on the GPU and delaying transfers from CPU to GPU as far as possible. Users of GPU Manager define *work requests* that specify the GPU kernel and any data transfer operations. The system controls the execution of work requests to maximize overlapping of computations and data transfers. The system returns control to the user upon completion of a work request through a callback object specified by the user per work request. Figure 3.7 shows a simple example of a GPU request on Charm++. The runtime calls `kernelSelect` at task execution passing a `workRequest` with necessary arguments to the kernel execution. The `workRequest` object at `kernelSetup` has more parameters such as input and output data, which are omitted for the sake of space.

```
1 void kernelSetup(void *cb) {
2   workRequest *wr= (workRequest*) malloc(sizeof(workRequest));
3   wr->callbackFn = cb; /* Callback function */
4   enqueue(wrQueue, wr); /* Enqueue this task to be executed */
5 }
6
7 void kernelSelect(workRequest *wr) {
8   helloKernel<<<wr->dimGrid,wr->dimBlock,wr->smemSize>>>();
9 }
```

Figure 3.7 – A code example of the Charm++ GPU Manager with a task submission to a GPU.

Vasudevan et al. (2013) improve the Charm++ GPU Manager by a new framework on top of Charm++ called G-Charm. It schedules at runtime chare objects between CPUs and GPUs based on the current loads and the estimated execution time provided by previous

executions. Its scheduling strategy, however, seems to not consider data transfer costs in its decision. The memory layer of G-Charm keeps track of chare buffers in order to reduce memory transfers. If a chare buffer is already present on a GPU memory it will be reused in the future. In addition, it dynamically combines multiple GPU kernels to reduce the cost of kernel invocations. Combining depends on adjacent chares with adjacent memory regions at application level.

3.2.3 StarPU

StarPU is a runtime system providing a data management facility and an unified execution model over heterogeneous architectures including GPUs and Cell BE processors (Augonnet, 2011; Augonnet et al., 2009a,b, 2011). It offers a runtime API and C language annotations designed into a GCC plug-in (Coutér, 2013). StarPU programming model relies on explicit parallelism by tasks with data dependencies and a memory layer to abstract transfers among disjoint address spaces.

StarPU execution model relies on the use of *codelet*, which is a task description with the input and output data. A codelet provides the means to express task dependencies and implementations of one task to different accelerators, being only necessary a specific function to each accelerator. The execution of codelets are asynchronous and with no order guarantee.

The data management facility provides a high level API that hides the complexity of transfers from and to accelerators. StarPU introduces the notion of *filters*, an interface to describe the data layout or to manipulate the sub-data. Each accelerator has a buffer that describes the parameters of a task and the access mode.

The StarPU runtime also provides a set of scheduling policies from dynamic work balance (*eager* policy) or based on performance models such as HEFT. On one hand, its runtime does not provide performance guarantees using an eager policy that has no extensions or improvements to heterogeneous systems. On the other hand, research on StarPU clearly focus on performance models for task prediction and scheduling through HEFT based algorithms. Previous works study the impact of different task prediction strategies along with HEFT extensions such as data prefetch. Nonetheless, StarPU transfer prediction lacks of more studies over its accuracy. It predicts data transfers based on an offline sampling of bandwidth transfer between PUs.

Figure 3.8 on the facing page illustrates an example of vector scaling using the StarPU API. First it uses the structure `starpu_perfmodel` to select the history-based performance model along with the entry name for the scaling task. Next, it creates a `starpu_codelet` structure, composed of two implementations (CPU and CUDA) and their function pointers. The vector scaling task is created at line 26.

3.2.4 StarSs and OmpSs

StarSs proposes a programming model to exploit task-level parallelism by OpenMP-like pragmas (Ayguadé et al., 2009a) and a runtime system to schedule tasks while preserving dependencies. The StarSs extensions have the advantage to simplify the parallelization of sequential programs to GPUs (GPUSs) and Cell BE (CellSs). The runtime is divided in three actors: main thread, helper thread and workers. The main thread generates tasks and

```
1 /* Here it selects the history-based performance model */
2 static struct starpu_perfmodel vector_scal_model = {
3   .type = STARPU_HISTORY_BASED,
4   .symbol = "vector_scale_model"
5 };
6
7 static starpu_codelet cl = {
8   .modes = { STARPU_RW },    /* access modes */
9   .where = STARPU_CPU | STARPU_CUDA,
10  .cpu_func = scal_cpu_func, /* CPU */
11  .cuda_func = scal_cuda_func, /* GPU */
12  .nbuffers = 1, /* number of arguments */
13  .model = &vector_scal_model
14 };
15
16 void compute(float* vector, int n){
17  starpu_data_handle vector_handle; /* vector handle */
18  starpu_vector_data_register(&vector_handle, 0, (uintptr_t)vector, n,
        sizeof(float));
19
20  struct starpu_task *scal_task = starpu_task_create();
21  scal_task->cl = &cl; /* task codelet */
22  scal_task->callback_func = NULL;
23  scal_task->buffers[0].handle = vector_handle;
24  scal_task->buffers[0].mode = STARPU_RW;
25
26  int ret = starpu_task_submit(scal_task);
27  starpu_task_wait_for_all();
28  starpu_data_unregister(vector_handle);
29 }
```

Figure 3.8 – StarPU program example for vector scaling.

the helper thread consumes them, mapping to the most suitable device. Worker threads wait for available tasks and perform the necessary data transfers between the main memory and the GPU memory. The StarSs scheduler is a variation of the work pushing strategy where the master generates tasks while the helper thread pushes tasks on workers' pool.

OmpSs (Bueno et al., 2011, 2012, 2013; Planas et al., 2013) is a continuation of StarSs and extends SMPSs (Badia et al., 2009) and GPUSs (Ayguadé et al., 2009a) by providing simpler code annotations with the capacity to have recursive tasks. OmpSs does not offer any library API to write a program and the user depends of the Mercurium compiler (Bueno et al., 2012).

Figure 3.9 on the next page illustrates an example of the matrix multiplication on top of OmpSs. The algorithm calls the method `matmult_tile` at line 21 and the runtime identifies two task versions of this method by directive `task` at lines 2 and 10. Note

that this directive also describes data-flow dependencies on created tasks. Lines 1 and 8 identify the task's target (*smp* or *cuda*), in addition to directive `implements` that gives an alternative version for the specified target device.

```
1 #pragma omp target device (smp) copy_deps
2 #pragma omp task input([BS*BS]A, [BS*BS]B) inout([BS*BS]C)
3 void matmul_tile(float *A, float *B, float *C, int BS) {
4   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
5     BS, BS, BS, 1.0, A, BS, B, BS, 1.0, C, BS);
6 }
7
8 #pragma omp target device (cuda) implements(matmul_tile) \
9     copy_deps
10 #pragma omp task input([BS*BS]A, [BS*BS]B) inout([BS*BS]C)
11 void matmul_tile_cuda(float *A, float *B, float *C, int BS) {
12   cublasSgemm('T', 'T', BS, BS, BS, 1.0, A, BS, B, BS, 1.0, C, BS);
13 }
14
15 void matmul(int mb, int nb, int kb,
16     float **A, float **B, float **C, int BS) {
17   int i, j, k;
18   for(i = 0; i < mb; i++)
19     for(j = 0; j < nb; j++)
20       for(k = 0; k < kb; k++)
21         matmul_tile(A[i*mb+k], B[k*kb+j], C[i*mb+j], BS);
22 }
```

Figure 3.9 – Example of matrix multiplication using multi-versioning with OmpSs annotations.

The OmpSs runtime, called Nano++, offers different scheduling strategies, and most results are reported on a centralized scheduling strategy with data locality on multi-CPU, multi-GPU, and clusters. This centralized strategy may compromise scalability on fined-grained algorithms. Its runtime provides memory coherence protocols such as write-back and write-through, but the write-back is on average always the best, as reported by Quintana-Ortí et al. (2009). Recent versions of OmpSs include specific multi-versions of the same task (Planas et al., 2013) and regions of strided and/or overlapped data (Bueno et al., 2013).

3.2.5 KAAPI Extensions for Iterative Computations

Everton Hermann's thesis studies a parallelization approach to tightly coupled computing units composed of multi-CPU and multi-GPU (Hermann, 2010). The work relies on the open source SOFA physics simulation library, which supports various types of differential equation solvers for single objects as well as complex scenes of different kinds of interacting physical objects (rigid objects, deformable solids, fluids).

One of his contributions was the **Multi-GPU Abstraction Layer** introducing multi-architecture data types. The multi-GPU implementation for standard data types intends to hide the complexity of data transfers and coherency management among multiple GPUs and CPUs. In the context of shared-memory multiprocessors all CPUs share the same address space and data coherency is managed by hardware. In opposite, even when embedded in a single board, GPUs have their own local address space. It describes the development of a distributed shared-memory (DSM) like mechanism to release the programmer from the burden of moving data between a CPU and a GPU or between two GPUs.

When accessing a variable, the proposed data structure first queries the runtime environment to identify the PU trying to access the data. Then it checks a bitmap to test if the accessing PU has a valid data version. If so, it returns a memory reference that is valid in the address space of the PU requesting data access. If the local version is not valid, a copy from a valid version is required. It happens when a PU accesses a variable for the first time, or when another PU has changed the data. This detection is based on dirty bits to flag the valid versions of the data on each PU. These bits are easily maintained by setting the valid flag of a PU each time the data is copied to it, and resetting all the flags but that of the current PU when the data is modified.

Another contribution is the **Multi-Implementation Task** definition. It requires an interface to hide the task implementation that is very different if it targets a CPU or a GPU. It provides a high level interface for architecture specific task implementations as demonstrated in Figure 3.10. First a task is associated with a signature that must be respected by all implementations. This signature includes the task parameters and their access modes (read or write). This information will be further used to compute the data dependencies between tasks. Each CPU or GPU implementation of a given task is encapsulated in a functor object, which provides a clear separation between a task definition and its various architecture specific implementations.

```
struct TaskName: Task::Signature<double*>{};

template<>
struct RunCPU<TaskName>{
  void operator()( double* a )
  { /* CPU code */ }
};
```

```
template<>
struct RunGPU<TaskName>{
  void operator()( double* a )
  { /* GPU code */ }
};
```

Figure 3.10 – C++ example of KAAPI multi-implementation task definition. In left is the CPU implementation, and in right the GPU implementation.

A scheduler approach is also implemented. It first relies on a task partitioning that is executed every time the task graph changes. Between two task graph partitioning, work stealing is used to reduce the load imbalance that may result from work load variations due to the dynamic behavior of the simulation. We detail its mixed approach in Section 2.3.3 on page 26.

3.2.6 Intel Xeon Phi Coprocessor Programming

The Intel Xeon Phi coprocessor is a throughput-oriented accelerator offering a broader variety of programming models unlike other HPC accelerators. The card runs a Linux-based operating system, called *uOS*, and has a x86 instruction set (see coprocessor details in Section 2.1.4 on page 15). Programming for this coprocessor has basically two approaches:

- **Offload** model where a (host) program offloads work to a coprocessor;
- **Native** model where a program natively runs on processors or coprocessors and may communicate by various methods.

Its software stack has combinations for both models (Jeffers and Reinders, 2013, chap. 7). The Offloading model is supported by different programming tools such as Intel Cilk Plus, OpenMP, and Intel TBB. Besides, these three tools have native execution support in conjunction with MPI applications, in which MPI processes may run on processors and coprocessors communicating by message-passing.

The Offload compiler and runtime of Intel Xeon Phi software (Newburn et al., 2013) offer language pragma directives `#pragma offload` to offload computations from a host processor to a coprocessor. The Offload compiler hides parallelism from programmer and manages data transfers and code execution, which rely on its runtime. Figure 3.11 shows an example of vector scaling using the Offload compiler. Line 2 indicates that the `for` loop will be offloaded to an Intel MIC coprocessor (directive `target(mic)`) and it has a data in read-write access mode (directive `inout`). Inside the offload code block the program can use OpenMP directives such as parallel loops and task parallelism (see Section 3.1.1 on page 31).

```
1 void scaling_vector(float* vector, int n, float factor) {
2 #pragma offload target(mic) inout(vector:length(n))
3 #pragma omp parallel for
4   for(int i= 0; i < n; i++) {
5     vector[i] = vector[i] * factor;
6   }
7 }
```

Figure 3.11 – Example of a scaling vector program using the Intel Offload compiler.

Recently, many other experiments have been done on the Intel Xeon Phi coprocessor. Cramer et al. (2012) evaluated the overhead of the offload directives and compared with a 16-socket 128-core system composed of Intel Xeon X7550 processors. It concluded that the overhead was low compared to large multicore systems.

The OmpSs technical report of Labarta and Beltran (2013) from the EU FP7 project DEEP reported preliminary experiments on Cholesky factorization on pre-release prototypes of the Intel Xeon Phi (Intel Knights Corner and Intel Knights Ferry) for matrix of size 8192×8192.

Pennycook et al. (2013) reported the use of the SIMD instruction sets of Intel Xeon processors and Intel Xeon Phi coprocessors to accelerate molecular dynamics (MD) simu-

lations. The use of an Intel Xeon Phi coprocessor with SIMD was $5.2x$ faster than scalar execution.

Eisenlohr et al. (2012) reported comparisons on a dense linear QR factorization in Intel Cilk Plus and OpenMP on a dual Intel Knights Ferry coprocessors connected through PCIe to a dual 6-cores Intel Westmere X5680. They observed some degradation of Intel Cilk Plus when the number of threads was higher than the hardware cores.

MAGMA (Tomov et al., 2010) implements static scheduling for linear algebra algorithms on heterogeneous systems composed of GPUs. Recently it has included some hybrid methods that use a Sandy Bridge processor and an Intel Xeon Phi coprocessor (Dongarra et al., 2013a).

Heinecke et al. (2013) designed three versions of the Linpack benchmark for nodes with Intel Xeon Phi coprocessors: native, single-node hybrid, and multi-node hybrid version.

3.2.7 Other High Level Tools

This section presents programming tools with a higher level programming model in order to hide architecture details.

HMPP (Dolbeau et al., 2007) is a set of compiler directives similar to OpenMP, tools and software runtime that support parallel programming in C for hybrid platforms. HMPP introduces the codelet concept that declares a computation or task to be executed on the accelerator, being the only code block with architecture details.

Harmony (Diamos and Yalamanchili, 2008) is runtime system with dynamic scheduling, out-of-order (OOO) execution, and programming model composed of compute kernels and control decisions. Sequoia (Fatahalian et al., 2006) is a memory hierarchy aware programming language to express explicitly locality and communication on top of clusters of hybrid machines. Merge (Linderman et al., 2008) is a high-level parallel programming language based on the map-reduce pattern for hybrid architectures. The Merge framework abstracts most of the architectural characteristics by a compiler and runtime support. As in Sequoia programming language, each work unit is a task, the principal parallel construct.

Furthermore, there are many other solutions concerning stream processing as Brook (Buck et al., 2004), RapidMind (McCool et al., 2006), PeakStream (Papakipos, 2007), and StreamIt (Thies and Amarasinghe, 2010).

3.3 Summary

In the context of multicore, shared-memory programming, several tools have studied task parallelism and dynamic scheduling by work stealing (Blumofe et al., 1995; Gautier et al., 2007; Leiserson, 2009; Reinders, 2007). This programming model favors granularity and asynchronicity that are essential to exploit parallelism and scalability in modern multicore architectures. Nevertheless, little research has been done on task parallelism with data dependencies with the exception of related works on Athaspascan/KAAPI (Galilee et al., 1998; Gautier et al., 2007).

On accelerators data parallelism has been studied by several programming languages (Gaster et al., 2012; Kirk and Hwu, 2012). Most of parallelism is expressed through fork-join in which a host application offload a computational block to an accelerator. Besides, there

are efforts on data dependencies to provide means of automatically transfer data such as OpenACC and Intel Offload Compiler.

On the other hand, several runtime systems have studied task parallelism with data dependencies systems over heterogeneous architectures (Augonnet et al., 2009a; Bueno et al., 2011). They provide a programming model able to describe a DAG of tasks to run over PUs. These runtime systems schedule tasks based on cost models or centralized strategies derived from list scheduling. However, there are few studies on the effects of data-flow programming models with dynamic scheduling strategies on heterogeneous architectures equipped with accelerators. Hermann et al. (2010) study the combination of work stealing and graph partitioning. But their approach is restricted to the domain of iterative applications.

Chapter 4

XKaapi Runtime System

Contents

XKaapi[1] is a novel implementation of the KAAPI runtime developed by the INRIA MOAIS[2] team. More than a runtime, XKaapi is a fully featured software stack to program parallel architectures. The core stack is written in C and is designed using a bottom up approach: each layer is kept as specialized as possible to fit a specific need.

Currently, the runtime stack includes: a runtime supporting multicores and multiprocessors; a set of ABIs (QUARK (YarKhan et al., 2011), OpenMP runtime libGOMP (Broquedis et al., 2012)); a set of high level APIs such as C++ API **Kaapi++** and C API **Kaapic** (Lementec et al., 2011a); and a source-to-source compiler (Lementec et al., 2011b) based on the ROSE compiler framework.

In this Chapter, we introduce the XKaapi programming model and runtime system, that are the basis of our contributions on heterogeneous systems. Section 4.1 overviews XKaapi's semantic and runtime. In Section 4.2, we introduce the Kaapi++ programming interface that is used throughout the rest of this dissertation. Section 4.3 details the runtime scheduler, the work stealing algorithm. In Section 4.4, we describe the XKaapi data-flow mechanism.

4.1 Overview

The XKaapi semantic remains sequential like in its predecessor Athapascan (Galilee et al., 1998), but the KAAPI runtime has been redesigned (Gautier et al., 2007) and then specialized for multi-CPU/multi-GPU iterative applications (Hermann et al., 2010). The proposal of XKaapi runtime is to target multicore architectures and accelerators such as GPU and Intel Xeon Phi. Figure 4.1 on the next page shows the scope of KAAPI (see Section 3.1.4 on page 34) and XKaapi on parallel systems.

[1] `http://kaapi.gforge.inria.fr`
[2] `http://moais.imag.fr`

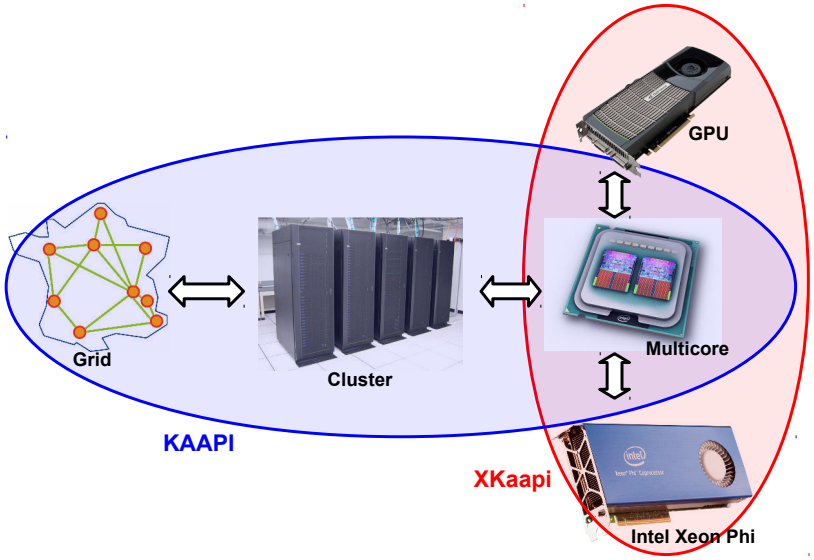


Figure 4.1 – Comparison between KAAPI and XKaapi runtime.

The XKaapi task model (Gautier et al., 2007), as in Cilk (Frigo et al., 1998), Intel TBB (Reinders, 2007), OpenMP-3.0 (Chapman et al., 2007) or StarSs (Ayguadé et al., 2009a; Bueno et al., 2012), enables non-blocking task creation: the caller creates tasks and continues the program execution. A XKaapi program is composed of sequential code and some annotations or runtime calls to create tasks. The parallelism in XKaapi is explicit, while the detection of synchronizations is implicit: the dependencies between tasks and the memory transfers are automatically managed by the runtime. The execution of a XKaapi program generates a sequence of tasks that access data in a shared memory. From this sequence, the runtime extracts independent tasks to dispatch them to the available PUs.

A task in XKaapi is a function call that returns no value except through the shared memory and the list of its effective parameters. Depending of the API, tasks are created using code annotation (`#pragma kaapi task` directive) if the XKaapi compiler is used (Lementec et al., 2011b), or by library function (`kaapic_spawn` call using XKaapi's C API, or by calling the template function `ka::Spawn`), or by low level runtime function calls.

4.2 Kaapi++ Interface

The Kaapi++ API is a C++ interface derived from the Athapascan API but with modifications at data sharing and task signature. Here we detail the Kaapi++ interface by mapping a C++ function to a Kaapi++ code with tasks.

First, a task is associated with a signature that includes the number of effective parameters and their access modes. The code from Figure 4.2 on the next page illustrates an example of task signature based on a C++ function signature. The `F` function has two effective parameters in which the first is an input (`n`) and the second an output (`result`).

C++ function signature	Kaapi++ task signature

`void F(`
`  double n,`
`  double* result`
`);`
	`struct TaskF: public ka::Task<2>::Signature<`
	`  double,  /* input parameter */`
	`  ka::W<double> /* output parameter */`
	`> {};`

Figure 4.2 – Example of a Kaapi++ task signature from a C++ function.

Second, the implementation of task `TaskF` from Figure 4.2 corresponds to a C++ function, *i.e.*, an object from a class having the `operator()` function specialized to a specific architecture. The implementation for a given architecture corresponds to a template specialization of the `TaskBodyCPU` or (not exclusive) `TaskBodyGPU` classes. The template classes must be specialized with the `UserTaskName`, *i.e.* the name of the class that has been used to defined the signature. The types and the number of effective parameters must match the task' signature. Each shared data of the effective parameters has to be enclosed in a `ka::pointer` type according to its access mode. The mapping from C++ function to a Kaapi++ task is shown in Figure 4.3. The input parameter `n` does not change, while the output argument `result` from a pointer type `double*` requires a `ka::pointer_w` class type.

C++ function definition	Kaapi++ task body specialization

`void F (`
`  double n,`
`  double* result`
`){`
`  *result = n*n+1;`
`}`
	`template<> struct TaskBodyCPU<TaskF> {`
	`  void operator() (`
	`  double n,`
	`  ka::pointer_w<double> result`
	`  ){`
	`  *result = n*n+1;`
	`  }`
	`};`

Figure 4.3 – Mapping of a C++ function to a Kaapi++ task.

Similarly to Athapascan (Roch et al., 2003), Kaapi++ has two access types: *immediate* and *postponned*. By default the access type of an object is *immediate* meaning that a task may directly access the object at execution time. On the other hand, an access type is *postponned* (access mode suffixed by "P") if the task will not directly perform an access on the object. Instead, it will create other (recursive) tasks that may access this object. The available access modes in Kaapi++ are:

- `ka::W<T>` for write access mode, meaning that the task will write a new value of type `T`.
- `ka::R<T>` for read access mode, meaning that the task will only read the previous value of type `T`, without possibility to modify the data.

- `ka::RW<T>` for exclusive access mode, meaning that the task will read or write a value of type `T`.
- `ka::CW<T>` for concurrent write access mode, meaning that the task will write a value of type `T` with accumulation law.

Figure 4.4 shows the conversion rules between task's signature definition and Kaapi++ task implementation.

type of *signature parameter*	*body definition's parameter* type
`T`	`T`
`T`	`const T&`
`ka::R[P] < T >`	`ka::pointer_r[p] < T >`
`ka::W[P] < T >`	`ka::pointer_w[p] < T >`
`ka::RW[P] < T >`	`ka::pointer_r[p]w[p] < T >`
`ka::CW[P] < T >`	`ka::pointer_cw[p] < T >`

Figure 4.4 – Kaapi++ Conversion rules between task's signature and task body arguments.

The mapping of C++ function call to Kaapi++ task creation is illustrated in Figure 4.5. The template function `ka::Spawn<T>` is a non-blocking call that creates a XKaapi task in a help-first policy. Note that in this case we do not need to modify the argument's types at creation time. Next, Kaapi++ function `ka::Sync` blocks execution until all tasks execute respecting the tasks' precedence. In this case both tasks can not execute in parallel because of a true dependency on `results` object (read-after-write dependency).

C++ function calls	Kaapi++ task creation

`F(n, &result);`
`Print(&result);`
	`ka::Spawn<TaskF>()(n, &result);`
	`ka::Spawn<TaskPrint>()(&result);`
	`ka::Sync();`

Figure 4.5 – Example of Kaapi++ task creation.

Kaapi++ also provides multi-versioning of a task implementation based on the work of Hermann et al. (2010). Each CPU or GPU implementation is encapsulated in a functor object, which must respect its task signature. This concept of multi-versioning and task implementation allows a clear separation between the task definition and its implementations. Moreover, the access modes in task's signature allow the runtime to automatically take care of memory transfers. Figure 4.6 shows an example of a task with CPU (`TaskBodyCPU`) and GPU (`TaskBodyGPU`) implementations conforming to its signature (Figure 4.2).

The runtime represents multi-versioning through a task format containing signature information and task implementations. The first consists of access modes and parameter types, while the second has a pointer to each task implementation. XKaapi expects at least the CPU implementation of a task signature. At task execution the runtime will execute the signature's implementation based on the current worker type, *i.e.* CPU or GPU.

Kaapi++ CPU version	Kaapi++ GPU version

`template<>`
`struct TaskBodyCPU<TaskF> {`
`  void operator() (`
 	
`    double n,`
`    ka::pointer_w<double> res`
`  )`
`  {`
`    /* CPU implementation */`
`  }`
`};`
	`template<>`
	`struct TaskBodyGPU<TaskF> {`
	`  void operator() (`
	`    ka::gpuStream stream,`
	`    double n,`
	`    ka::pointer_w<double> res`
	`  )`
	`  {`
	`    /* GPU implementation */`
	`  }`
	`};`

Figure 4.6 – Example of Kaapi++ multi-versioning with CPU and GPU implementations.

We note, however, that the code inside a GPU implementation is not compiled nor verified by XKaapi. Kaapi++ interface allows to include GPU specific code, such as CUDA, if and only if the application is compiled by a GPU compiler. An application with CUDA extensions within a GPU task should be compiled by NVIDIA CUDA compiler `nvcc` since it also support C++ code.

4.3 Scheduling by Work Stealing

The XKaapi runtime implements work stealing inspired by Cilk design described by Blumofe et al. (1995) and Frigo et al. (1998). Thanks to Cilk, the work stealing technique has become popular and is often considered when it comes to dynamically balance the work load among PUs. The work stealing principle can be synthesized as follows. An idle thread, called a thief, initiates a steal request to a random selected victim. On reply, the thief receives a copy of one ready task, leaving the original task marked as stolen. Coherency between a thief and its victim is ensured by a Dijsktra-like protocol from Frigo et al. (1998).

The XKaapi runtime creates a system thread for each worker, which is in general a processor core. A thread creates tasks and pushes them on its own work queue, which is represented as a stack. The enqueue operation is very fast, typically about ten cycles on the last x86/64 processors (Broquedis et al., 2012). As for Cilk, a running XKaapi task can create child tasks, which is not the case for the other data-flow programming tools previously mentioned (Augonnet et al., 2009b; Ayguadé et al., 2009a; YarKhan et al., 2011), except the recent StarSs extension OmpSs (Bueno et al., 2012). Once a task ends, the thread executes its children following a FIFO order by popping tasks from its own work queue. During task execution, if a thread finds a stolen task, it suspends its execution and switches to the work stealing scheduler that waits for dependencies to be met before resuming the task. Otherwise, and because sequential execution is a valid order of execution (Galilee et al., 1998; Gautier et al., 2007), tasks are performed in FIFO order without

computation of data-flow dependencies.

The main difference between XKaapi and other software (Augonnet et al., 2009b; Bueno et al., 2012; YarKhan et al., 2011) is that XKaapi computes data-flow dependencies only when an idle thread searches for a ready task. Computing data-flow dependencies during steal operations reduces the overhead of normal task execution in recursive programs where the number of steals is dependent of the critical path. This concept follows the work-first principle (Frigo et al., 1998): at the expense of a larger critical path, XKaapi moves the cost of computing ready tasks from the work performed by the victim during task's creation to the steal operations performed by thieves.

Thanks to our approach, the classical fine-grained recursive Fibonacci in a data-flow implementation shows an overhead T_1/T_{seq} of about 10 (Broquedis et al., 2012), which is of the same order as Cilk or TBB that do not handle data-flow dependencies. In XKaapi, the cost of task creation is several orders of magnitude lower than in StarPU (Augonnet et al., 2009b), StarSs (Ayguadé et al., 2009a) or OmpSs (Bueno et al., 2012).

4.3.1 Runtime Data Structures

The XKaapi runtime represents a worker stack by *activation frames* inspired by the Cilk runtime (Frigo et al., 1998, 2009). Each instance of a XKaapi task creates an *activation frame* at runtime. A running task creates an instance of an activation frame and pushes new tasks and their arguments to it. An activation frame is composed of:

- **data stack pointer (sp_data)** – the next available position in the data stack, which grows from the bottom to the top;
- **stack pointer (sp)** – the next available task position at the bottom of the task stack, which grows from the top to the bottom;
- **current task (pc)** – it points the next ready task to execute, from the point of view of the current worker. It executes from the top (least pushed task) to the bottom (last pushed task).
- **task list (tasklist)** – a list of tasks with calculated dependencies. This mechanism is explained in Section 4.3.3.

Each XKaapi worker has a *thread context* allocated at execution time. The thread context provides storage for task stack, which is a set of activation frames, and data stack in a contiguous block of memory. Its fields are:

- **data stack (sp_data)** – storage for data arguments;
- **activation frame stack (stackframe)** – storage for the activation frames of tasks;
- **stack frame pointer (sfp)** – it contains information about the current frame (*pc*, *sp*, etc).

Both stacks grow in opposite directions inside a continuous memory block, provided by the thread context. The task stack grows from the top to the bottom, and the data stack grows from the bottom to the top. A XKaapi thread context is full when the stack pointer (*sp*) is equal to the data stack pointer (*sp_data*). The runtime guarantees that an allocation request in one stack (data or task) returns a contiguous block of memory.

4.3.2 Concurrent Steal Requests

If a program is highly parallel, i.e. $T_\infty \ll T_1$, then the number of steal operations per thread remains in order $O(T_\infty)$ which is low. In that case, the cost of computing data-flow, perhaps multiple times if several idle threads iterate over the same queue, is negligible with comparison to systematic computation on task creation. Otherwise, if the frequency of steal operations increases, XKaapi tries to aggregate multiple requests to the same victim.

XKaapi aggregation protocol elects one "thief" worker to reply to all steal requests. This aggregation strategy reduces the number of steal requests and the computing of data-flow dependencies from k steal requests to a less costly operation requesting k ready tasks. Hendler et al. (2010) give a theoretical analysis and show a reduction of the total steal request number. In addition, Tchiboukdjian et al. (Tchiboukdjian et al., 2012) propose a theoretical analysis of work stealing with task dependencies, considering a XKaapi-like protocol.

4.3.3 Reduction of Steal Overhead

In order to reduce the cost of computing ready tasks, XKaapi implements an optimization to compute ready tasks at steal operation. The runtime attaches to the worker's victim stack an accelerating data structure for steal operations. The runtime switches to this new structure when the cost becomes important, especially when the victim's stack contains many tasks, as for instance in blocked linear algebra algorithms (Buttari et al., 2009).

In a steal operation, the scheduler computes a list of tasks' successors from data-flow dependencies and attaches it to the worker's stack. The successors of a task are tasks with true dependencies with the task. Subsequently, using the successors list, activated tasks are pushed directly into a ready task list. If new tasks are created, the scheduler computes a new list of their successors. Therefore, the search for a ready task, which would be proportional to the number of tasks in the stack (and to the number of task arguments k in order to computer their dependencies), switches to constant time (access to the first element of the list of ready tasks). Consequently, subsequent steal operations in a thread with a ready task list have lower cost. This optimization moves the overhead of computing ready tasks from each steal operation to the steal operation that detects new tasks in the stack.

This optimization is also possible by the Kaapi++ interface that provides a task attribute `ka::SetStaticSched()`. Indeed, XKaapi creates an embedded task that will calculate dependencies and attaches the list of ready tasks using the same algorithm from the steal operation. In Appendix A on page 139 we detail an example of the task attribute in the Cholesky algorithm.

4.4 Data Flow Computation

In XKaapi runtime, tasks share data if they have access to the same memory region. A memory region is defined as a set of addresses in the process virtual address space. The user is responsible for indicating the mode each task uses to access memory: the main access modes are *read*, *write*, *reduction* or *exclusive* (read and write).

During a steal operation, the thief thread computes true dependencies (*read after write*) between tasks according to the access modes. At the expense of memory copy, the scheduler may solve false dependencies (*write after read* and *write after write*) through variable renaming. Algorithm 2 shows a simplified version of XKaapi data-flow computation. It has as input the stack of a worker's victim p_i^{stack}, and output a ready task t. The algorithm maintains the last access mode to a specific shared variable, defined as a_{last}, through a hash table indexed by its memory address. Each access in write or exclusive mode produces a new data version, which creates a true dependency.

Algorithm 2: Data-flow computation in XKaapi.

Input : victim stack p_i^{stack}
Output: ready task t

```
1  for frame ← top frame of p_i^stack to bottom frame of p_i^stack do
2      for t ← top task of frame to bottom task of frame do
3          foreach shared data d_i of task t do
4              a_i ← access mode for d_i
5              a_last ← last access mode to d_i
6              if a_i is postponned access type then
7                  mark a_i as ready dependency
8              end
9              if a_last is void or a_i is write-only access then
10                 mark a_i as ready dependency
11             end
12             if a_i == a_last and a_i is read access then
13                 mark a_i as ready dependency
14             end
15             if a_last is void or a_i has write access then
16                 a_last ← a_i
17             end
18         end
19         if all shared arguments of t are ready then
20             return t
21         end
22     end
23 end
24 return nil
```

4.4.1 DFG Example

Figure 4.7 on the facing page illustrates a code fragment to compute the matrix multiplication using the Kaapi++ API. Each parameter `ri, rj, rk` of type `ka::rangeindex` corresponds to a range of indexes. The data type `ka::range_2d` is an abstraction to view a memory region as a 2D array. A construction such as `A(ri,rk)` represents the

sub-matrix of elements from position `(i,k)` to `(i`blocksize,k+blocksize)+ with the same leading dimension size of `A` (by method `A->lda()`).

```
1 template<> struct TaskBodyCPU<TaskGEMM> {
2   void operator() (
3     ka::range2d_r<double> A, ka::range2d_r<double> B,
4     ka::range2d_rpwp<double> C
5   ){
6     for( i=0; i < N; i+= blocksize ){
7       ka::rangeindex ri(i, i+blocksize);
8       for( j=0; j < N; j+= blocksize ){
9         ka::rangeindex rj(j, j+blocksize);
10        for( k=0; k < N; k+= blocksize ){
11          ka::rangeindex rk(k, k+blocksize);
12          ka::Spawn<TaskGEMM>()( A(ri,rk), B(rk,rj), C(ri,rj) );
13        }
14      }
15    }
16  }
17 };
```

Figure 4.7 – Simplified example of a Kaapi++ blocked matrix product.

Figure 4.8 on the next page shows an example of a XKaapi DFG from the blocked matrix product algorithm of Figure 4.7. Each object of the DFG represents:

- **orange diamond** – a shared data instance in which we use labels to illustrate the matrix tile, such as `A(i,j)`;
- **pink square box** – an access to a shared object creating a data version, displayed by *v. 0* for initial version;
- **green ellipse** – a task, in this example the `GEMM` for `TaskGEMM`.

Dependencies between objects are represented as arrows in the DFG. Standard arrows into tasks are read dependencies, while arrows with a diamond tail have write access. Dotted lines indicate the sequence of data versions; in our example, they connect nodes for `C` tiles.

4.5 Summary

In this Chapter we detailed the XKaapi runtime and programming model. As successor of Athapascan/KAAPI, XKaapi has different design choices at runtime level in order to efficiently run on multicore architectures and to reduce overhead on fine-grained algorithms.

The programming model is similar to Athapascan, but it dictates that tasks share data having access to the same memory region. The Kaapi++ API adds the concept of task signature that specifies the number of arguments and access modes. In addition, Kaapi++

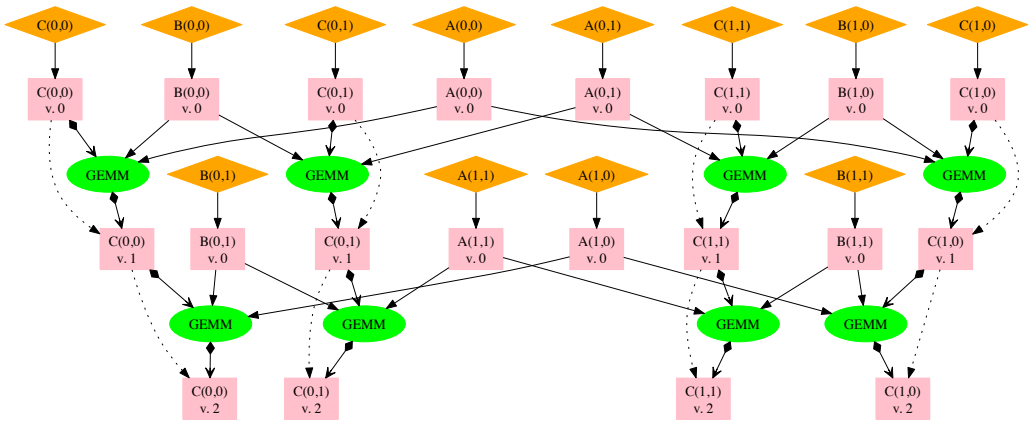


Figure 4.8 – A XKaapi DFG of the blocked matrix product.

concept of multi-versioning allows a clear separation between the task definition and its implementations.

The XKaapi runtime reduces costs mainly using activation frames and work stealing optimizations. Concurrent steal requests and on-demand ready task list provide provably optimizations to our work stealing scheduler (Tchiboukdjian et al., 2012). Besides, the allocation of one stack for data and activation frame in a continuous block of memory allows constant time operations, instead of queues of memory blocks like in KAAPI. However, the current XKaapi mechanism of stack allocation does not support on-demand allocation when the stack is full.

The standard version of XKaapi does not have support for heterogeneous systems, even if the Kaapi++ API predicts task multi-versioning. Hermann et al. (2010) report the first experiments on data-flow computations on multi-GPU for iterative computations. However, its strategy does not provide a generic design of work stealing with multi-GPU scheduling.

Part II

Contribution

Chapter 5

Runtime Support for Multi-GPU Architectures

Contents

The works in this Chapter were published in Lima et al. (2012) and Gautier et al. (2013b).

In heterogeneous systems with accelerators such as GPUs, it is essential a runtime system to abstract hardware details and offers a programming model able to express parallelism with few architecture details. The runtime has to overcome challenges such as a programming model, a memory view of the system, and scheduling.

In this Chapter, we describe our first contribution of this thesis. We detail the XKaapi runtime extensions for heterogeneous architectures composed of multi-CPU and multi-GPU. The extensions implement a programming model that offers asynchronous execution of GPU tasks and abstracts memory details. Algorithms on top of XKaapi describe the execution flow through the task dependencies and the runtime decides the target resource (CPU or GPU) and performs memory transfers as necessary. Our current version supports NVIDIA CUDA and it relies on the features of recent GPUs such as Fermi and Kepler models.

The remainder of this Chapter details the features to support multi-GPUs in XKaapi by Kaapi++ annotations (Section 5.1), runtime workers with GPUs and asynchronous execution (Section 5.2), concurrent GPU operations (Section 5.3), memory management (Section 5.4), and scheduling (Section 5.5). We conclude with the experimental results (Section 5.6) on the Idgraf heterogeneous architecture.

5.1 Kaapi++ User Annotations

XKaapi provides for programmers the concept of *user annotation* to pass scheduling hints in the Kaapi++ API. Its goal consists in advising the scheduler that a task is more efficient on a certain processor type, since CPUs and GPUs have different processing power. The main annotation for scheduling strategies on heterogeneous architectures is the `SetArchitecture` (or `SetArch`). The `SetArch` annotation restricts a task to a specific architecture type (CPU or GPU) and the runtime shall comply with this condition. Thus, a task with attribute CPU (`ka::ArchHost`) or GPU (`ka::ArchCUDA`) will not be executed by a worker of different type, allowing CPU-only and GPU-only tasks.

An example of scheduling hints by user annotations is illustrated in Figure 5.1. It creates two independent tasks in which the first executes on any CPU (line 2) and the second on any CUDA GPU (line 5). Appendix A on page 139 gives a complete example of the Cholesky factorization with user annotations.

```
1 /* CPU-only task */
2 ka::Spawn<TaskOnlyCPU>( ka::SetArch(ka::ArchHost) )( /* */ );
3
4 /* GPU-only task */
5 ka::Spawn<TaskOnlyGPU>( ka::SetArch(ka::ArchCUDA) )( /* */ );
```

Figure 5.1 – Example of scheduling hints in the Kaapi++ API.

We note that only CPU tasks have support for recursive task creation. Recursive support from GPU tasks would demand a finer control of memory regions, which was previously studied by Bueno et al. (2013).

5.2 GPU workers and Task Execution

Our runtime for multi-GPU dedicates a CPU core to manage a target GPU in the same way as other runtime tools such as OmpSs and StarPU. Figure 5.2 illustrates the execution mechanism of XKaapi over three computing units (workers): two CPUs and one GPU. The CPU cores become CPU workers and execute CPU computations (`TaskBodyCPU` code), while the GPU core becomes a GPU worker and does not compute CPU tasks. This worker is dedicated to find ready tasks and to send computations to the GPU. Besides, the core executes all host code to manage the GPU such as memory management and execution control.

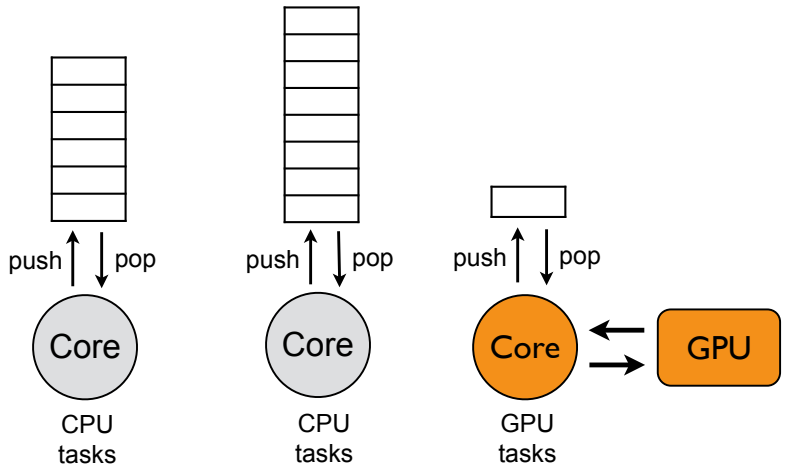


Figure 5.2 – Runtime structure of XKaapi with multi-GPU. A core is dedicated to control a target GPU.

For each task executed on a GPU, the runtime first requests memory from our software cache and transfers the input data. XKaapi basically inserts *prologue* and *epilogue* hooks before and after task execution, respectively, to perform those requests. The software cache request may result in a memory allocation under the availability of GPU memory for new data. The runtime assumes that the GPU task implementation launches the GPU kernels asynchronously.

XKaapi has a mechanism of software cache to handle memory allocation and consistency (Section 5.4). Data transfers and kernels on GPU are handled asynchronously by an extension of CUDA streams in order to abstract the control flow of GPUs (Section 5.3). Once a task implementation (`TaskBodyGPU`) launched computations on a GPU, XKaapi scheduler will select the next ready task to execute by sending its input data in advance in order to overlap data transfer with kernel execution (Section 5.3). Data transfers and kernel invocation on a GPU are handled asynchronously as well as the completion of these operations.

5.3 Concurrent Operations between CPU and GPU

Recent GPUs such as NVIDIA's Fermi allow new techniques to explore asynchronism in multi-GPU systems. Fermi GPUs have one execution engine and two copy engines capable of concurrent execution and transfers (two-way host-to-device and device-to-host), under the condition that no explicit nor implicit synchronization occur. This Section details how we exploit these capabilities.

XKaapi has an execution strategy for GPUs that avoids CUDA's implicit synchronizations and exploits concurrent memory transfers in the two ways along with kernel execution. It splits the execution of a GPU task in three basic operations: host-to-device input transfers (H2D), `TaskBodyGPU` execution (*i.e.* launch of CUDA kernels) (K), and device-to-host output transfers (D2H).

5.3.1 Kstream Structure

Since concurrency between data transfers and kernel launches must use CUDA streams, we define a new data structure, called **kstream**, that encapsulates three types of CUDA streams: a stream for host-to-device transfer, a stream for kernel execution and a stream for device-to-host transfer. The kstream structure allows to insert a request for one of the three types it handles (H2D, K, or D2H).

A callback function and its argument can be specified for each request insertion. Moreover, after each request insertion, the kstream inserts a CUDA event to detect the completion of the asynchronous operation. Once the kstream detects the event completion, it calls the callback function with its argument as parameter. It is the responsibility of the client of the kstream structure to regularly poll for the completion of asynchronous requests by calling a specific function.

This design allows concurrent execution between CUDA streams of each type. The kstream represents three flows of FIFO ordered GPU operations whose execution are independent from each other. The FIFO order is only respected among operations of the same type (H2D, K or D2H). The callback mechanism permits to compose a sequence of operations and it is typically used by the GPU work stealing algorithm: first to insert data transfers for inputs of a task, and then to invoke the `TaskBodyGPU` when the transfer ends. Our callback strategy permits H2D-K-D2H execution order and uses events to ensure consistency. In addition, CUDA events are a lightweight mechanism to closely monitor the device's progress and avoid implicit synchronization.

The runtime does not require synchronization points in the `TaskBodyGPU` code; indeed, the programmer must be aware that synchronous operations can lead to "holes" in the kstream pipeline so he must avoid them. It is important to note that implicit synchronizations, such as device and page-locked memory allocations and GPU operations to the default stream (named as 0), block other streams of the same device.

5.3.2 Sliding Window

Figure 5.3 on the next page illustrates the way our kstream structure allows to pipeline concurrent operations on a Fermi GPU. XKaapi uses a *sliding window* strategy to limit

the number of enqueued operations at each stream in the GPU. In our example, the sliding window of Figure 5.3 has two operations for each computing stream.

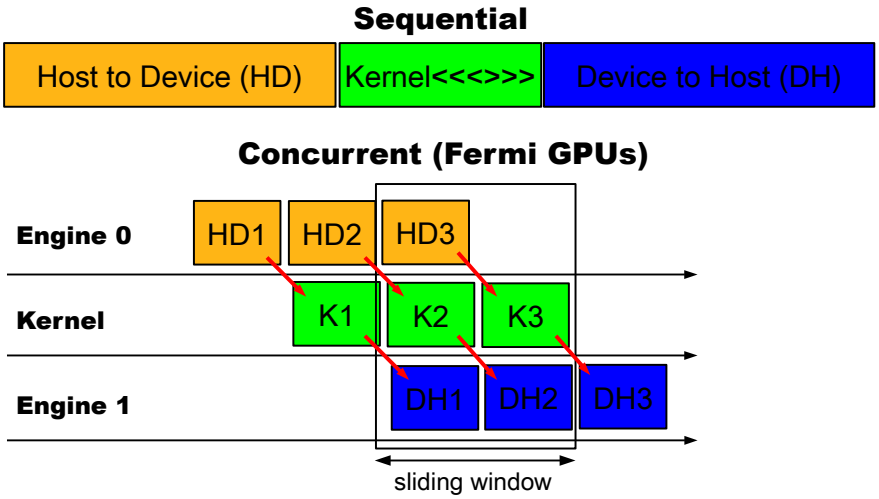


Figure 5.3 – Sequential and concurrent operations in a recent GPU card.

We empirically found that the best performance gain is obtained when having two tasks being processed per GPU. Starting more tasks do not increase performance significantly and reduce the capacity to balance the work load, because GPU tasks can not be aborted neither stolen after the start of a GPU transfer. In addition, a sliding window of two operations by stream would guarantee the overlap of transfer and execution minimizing pipeline gaps.

5.4 Memory Management

Similar to a DSM, XKaapi memory management enables the use of different address spaces, which are transparent to the programmer, and divides a heterogeneous system in main memory (or host memory) and GPU device memory of each card. The runtime keeps track of physical addresses of each data through a structure called *Kaapi Memory Data* (kmd).

Each instance of a kmd associates one memory address on each address space, *i.e.*, one CPU memory address and one GPU memory address for each card, or *null* if the data is not present in the corresponding address space. Each address space maintains the *kmd*s that belong to it, and performs *kmd* queries in linear time $O(n)$ through a hash table indexed by the memory pointers contained in this space. Optimize this query is essential to reduce data consistency's overheads.

Figure 5.4 on the following page illustrates a simplified version of *kmd* structure for a shared data argument of a hypothetical XKaapi task. It has basically three fields to track the pointer values for each address space (*data*), a bitmap to track which address space has a valid valid copy (*valid_ bits*), and a bitmap to track which address space has a pointer allocated previously (*add_ bits*). In this example the host and GPU 1 have a pointer to this

shared data. We protect bitwise operations on the bitmaps by atomic built-in functions of C compilers such as GCC.

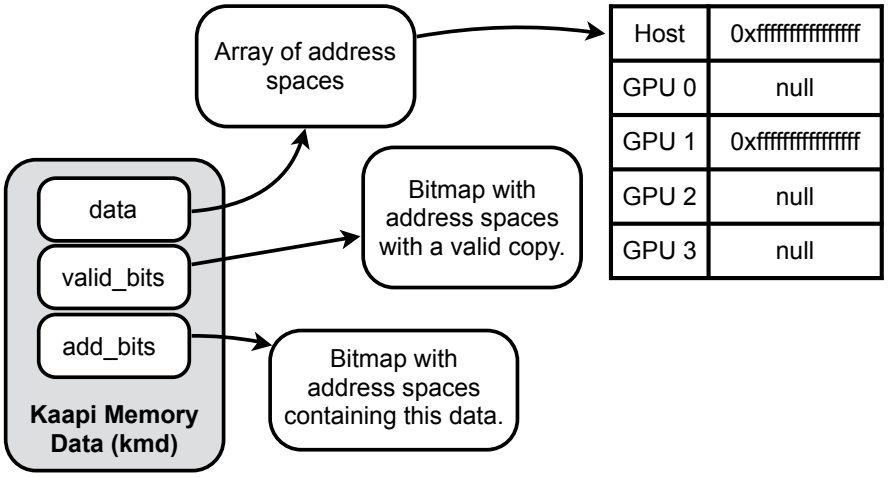


Figure 5.4 – XKaapi kmd structure to track valid replicas of a shared data.

The designed memory management is divided in two main components: software cache and consistency mechanism.

5.4.1 Software Cache

XKaapi manages GPU memory through a software cache, based on the Least Recently Used (LRU) replacement policy. Each GPU worker maintains two FIFO queues in order to keep track of allocated blocks. One queue stores blocks in read-only (RO) mode and the other stores blocks in read-write (RW), write-only (WO) modes or concurrent write (CW) modes. The first positions of the RO and RW/WO queues contain the last recently used blocks, and the last positions the least recently used blocks. When a GPU task requires to access a host memory block that is not present on the GPU, the runtime will allocate memory and insert it in one of the two queues, based on its access mode, after it has initiated the data transfer (for data in read access mode).

If the GPU memory is full, the software cache tries to evict the least recently used memory block of its own queue (LRU policy). If possible, unused blocks are reused without being freed. It verifies first at the end of the RO queue and, then, into the RW/WO queue, respectively, if a memory block bigger or equal than the requested size is not accessed anymore. Otherwise, it may free blocks from RO and RW/WO/CW queues, respectively, as needed. This optimization avoids unnecessary CUDA calls. Furthermore, the use of two queues (RO and RW/WO/CW) ensures that data produced by one task in write mode has more chance to remain on the GPU than read-only data.

In order to enable asynchronous memory transfers with CUDA, user data is page-locked through specific CUDA library function (`cudaHostRegister`). Similar strategy exists for StarPU (Augonnet et al., 2011), but it requires the use of strict data structures provided by the StarPU runtime such as vector and matrix.

5.4.2 Consistency

Consistency is guaranteed by a lazy strategy using a write-back policy. Data transfers to or from GPU occur only when a task accesses data and when the data is in an invalid state in the target address space. This policy avoids unnecessary transfers, unlike write-through policy (Augonnet et al., 2011; Bueno et al., 2012; Quintana-Ortí et al., 2009).

All transfer operations are asynchronous and rely on the use of our kstream data structure to signal the completion of operations. In the case of GPU-to-GPU transfers, the runtime first performs a transfer device-to-host from a GPU with a valid copy, followed by a host-to-device transfer to the GPU that owns the task.

CUDA 4.1 or later includes transfers between two GPUs directly by DMA called peer-to-peer device access. This feature is available when the function `cudaDeviceCanAccessPeer` returns true. The current version of XKaapi does not make use of this feature, because of its unpredictable behaviour concerning the GPU copy engines: with it, we could not guarantee the coherence of concurrent data copies, nor their overlapping with a kernel execution.

5.5 Runtime Scheduling

The XKaapi version for multicore architectures implements the work stealing scheduler based on Cilk and has specific optimizations for fine-grained parallel algorithms, which have been sketched in Chapter 4 on page 47. For each used GPU, or GPU worker, the runtime launches a thread on the host machine that runs a modified work stealing algorithm.

In addition, each CPU or GPU worker has a local queue named *mailbox* in which remote workers can push tasks. Scheduling heuristics make use of this additional queue by worker to improve locality based on different parameters. Yet, the local queue is not restricted to work stealing scheduling and can be employed to schedule static distributions.

We designed in XKaapi the classic work stealing algorithm, inherited from previous works on multicore architectures, and two scheduling heuristics for local optimization based on meta-data information of memory consistency: data-aware (H1) and locality-aware (H2).

5.5.1 Work Stealing

In comparison with original multi-CPU work stealing, multi-GPU work stealing adds a new state in the task state diagram that corresponds to a task for which input data are under transfer. The GPU worker polls regularly the completion of previous asynchronous GPU operations.

A task that completes its execution, when the asynchronous kernel launch has completed, activates the successor tasks (according to the data-flow dependencies) that become ready. These new ready tasks are pushed on the tasks' queue attached to the current GPU and they may be stolen by one CPU or another GPU.

Since GPU task execution is asynchronous, the end of a `TaskBodyGPU` code does not guarantee the conclusion of its launched kernels or data transfers to the host memory. The sliding window of concurrent operations also helps our work stealing scheduler because the

number of enqueued GPU tasks is reduced and does not retain all ready tasks, which can be stolen by other GPUs or CPUs. Activation of successor tasks occurs only after the end of all launched kernels from a GPU task. However, modified data in the GPU is not transfered back to the host memory. This mechanism favors our write-back consistency policy detailed in Section 5.4 on page 63.

5.5.2 Data-Aware Work Stealing (H1)

The goal of our data-aware heuristic, also here named **H1**, is to reduce memory transfers between host and devices in order to execute a ready task. It is similar to the classic work stealing but considers meta-data information to reduce memory transfers. Bueno et al. (2012) proposed a related scheduling strategy named locality-aware, but over a centralized scheduler.

In our strategy, for each ready task to be pushed, the algorithm first goes through every shared data argument of the task and searches for the workers where this data is in valid state. If the argument is valid, it keeps track of the amount of valid data (bytes) in this worker. The worker that owns the maximum number of data bytes in valid state for this task is then chosen as target to run the task. The ready task will be pushed onto the *mailbox* of the target worker, which would execute tasks from its mailbox before becoming a "thief". We note that a ready task pushed into the mailbox of a worker may be stolen if a worker becomes idle.

In Algorithm 3 we illustrate our designed algorithm. It accumulates for each shared argument the total number of valid bytes on each worker p_j. At line 8 it sorts workers by decreasing order of bytes and selects worker p_j that minimizes data transfer to execute task n.

Algorithm 3: Data-aware work stealing to reduce data transfers.

Input : ready task n
Output: target worker p_i

```
1 foreach shared argument d_i of task n do
2 |   foreach worker p_j do
3 |   |   if d_i is in valid state on the memory node of worker p_j then
4 |   |   |   total_j ← total_j + size in bytes of d_i
5 |   |   end
6 |   end
7 end
8 p_i ← worker j from total_j that has the maximum number of bytes in valid state
9 return p_i
```

5.5.3 Locality-Aware Work Stealing (H2)

The goal of our locality-aware, also named here **H2**, is to reduce invalidations of data replicas based on an owner-computes rule (OCR). This heuristic is similar to the approach proposed by Acar et al. (2000), but with an automatic scheme to (locally) reduce the

number of cache invalidations instead of the explicit code annotation. Guo et al. (2010) also propose a similar locality heuristic.

Our locality-aware strategy searches a shared data argument that has *write* or *exclusive* access mode. It pushes a ready task to the *mailbox* of a worker (CPU or GPU) that has a valid copy of this argument (*i.e* output argument). If more than one worker is eligible, then the scheduler simply selects a worker at random. We note that a ready task pushed into the mailbox of a worker may be stolen if a worker becomes idle.

We show our algorithm in Algorithm 4. It goes through each shared argument of a task n and tests the access mode. If the argument has *write* access, it queries to the memory management which worker has a valid copy of d_i. The algorithm returns the local (current) worker p_{local} if no worker or no write argument are found.

Algorithm 4: Locality-aware work stealing to reduce cache invalidations.

Input : ready task n
Output: target worker p_i

```
1 foreach shared argument d_i of task n do
2 |   a_i ← access mode for d_i
3 |   if a_i has write access then
4 |   |   p_i ← a worker with d_i in valid state
5 |   |   return p_i
6 |   end
7 end
8 return p_local
```

5.6 Experiments

The goal of our experiments is to evaluate the XKaapi runtime extensions for heterogeneous architectures composed of multi-CPU and multi-GPU. We evaluate:

- XKaapi ability to overlap communication with task execution (Section 5.6.3);
- Single-GPU and multi-GPU performance (Section 5.6.4);
- Comparison of scheduling heuristics (Section 5.6.5);
- The impact of CPUs to improve performance (Section 5.6.6).

In each experiment, we show in the x-axis the number of resources as the number of GPUs or the number of CPUs and GPUs for each execution. We employ this notation to clearly distinguish the number of computing CPUs and GPUs at runtime. Since XKaapi dedicates a CPU to manage a GPU, the number of computing CPUs is the number of total CPUs minus the number of GPUs. Each result is a mean of 30 executions. The 95% confidence interval is represented on the graphs.

5.6.1 Platform and Environment

All experiments have been conducted on a heterogeneous, multi-GPU system, named "Idgraf" (see Section 2.1.5 on page 16). Figure 5.5 illustrates the hardware topology of Idgraf with two hexa-core CPUs and eight Tesla C2050 GPUs.

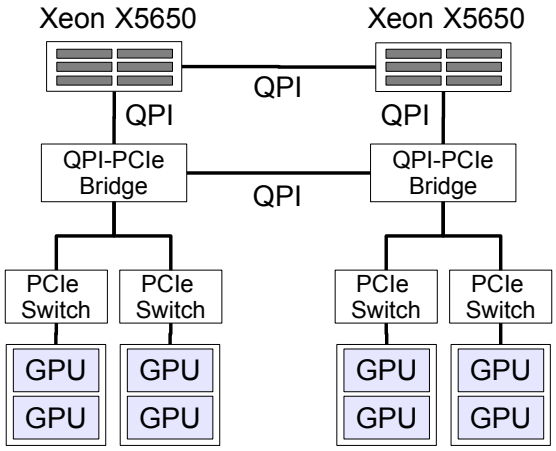


Figure 5.5 – Idgraf hardware topology for multi-GPU experiments.

We used as software environment GNU/Linux Debian squeeze x86/64, the compiler GCC 4.4, CUDA 5.0, and the library ATLAS 3.9.39 (BLAS and LAPACK). We also used MAGMA 1.1.0 for linear algebra algorithms and StarPU 1.0.1 (with its HEFT scheduling algorithm) as performance references.

5.6.2 Benchmarks

Our experiments use the same parallel version of the dense linear algebra problems matrix product ($C \leftarrow \beta C + \alpha AB$) and Cholesky factorization, as found in PLASMA (Buttari et al., 2009). The matrix data layout is the same as in PLASMA (tile data layout).

In XKaapi version we designed two versions of the parallel Cholesky algorithm: default algorithm as found in PLASMA, and a parallel-diagonal version. The parallel-diagonal Cholesky is a two level algorithm in order to exploit XKaapi capacity for CPU recursive tasks. The first level uses the PLASMA algorithm with 1024×1024 tiles, and the second level unfolds the panel factorization (`DPOTRF`) in sub-tiles of size 128×128 using the same parallel algorithm of the first level. The details of both Cholesky algorithms on XKaapi are depicted in Appendix A on page 139. We have not used auto-tuning to select the sizes of the tile and sub-tile, but an empirical approach: after a few experiments showing their average good performance, we have decided to use theses values.

We calculate the number of Flops according to PLASMA (Buttari et al., 2009) algorithms. All results, except when specified, are in double precision floating-point operations.

5.6.3 Concurrent Operations

This Section presents our experiments to evaluate the capacity of XKaapi to exploit asynchronous data transfers in concurrence with GPU kernel executions. Our experiment mea-

sures the performance of the matrix product algorithm. Matrices A and B of dimension $N \times N$ are decomposed into tiles (or blocks) of size $s \times s$. We devised our implementation such that all computations are performed on the GPU. Matrix computation is done with double precision, each block-matrix product launches CUDA kernels using the CUBLAS `DGEMM` routine.

We compared the performance of three versions:

- **CUBLAS (no copy)** – CUBLAS when the time to copy input and output matrices is not considered. We note that *CUBLAS (no copy)* is used as reference of the GPU peak performance.
- **CUBLAS** – CUBLAS with input and output transfers included using asynchronous execution, *i.e*, we used non-blocking calls and a synchronization at the end. The obtained results did not include memory allocations.
- **XKaapi (tile=s)** – our XKaapi implementation with the blocked matrix multiplication. Tasks used native calls to CUBLAS `DGEMM` on $s \times s$ tile sizes. Each measure includes all the costs of CUDA memory allocations and data transfers.

Figure 5.6 illustrates the results of `DGEMM` with the three versions and different tile sizes for XKaapi. *CUBLAS (no copy)* attained its peak performance (about 315 GFlop/s) for 4096×4096 square matrices. For larger matrices, the performance decreased to 293 GFlop/s.

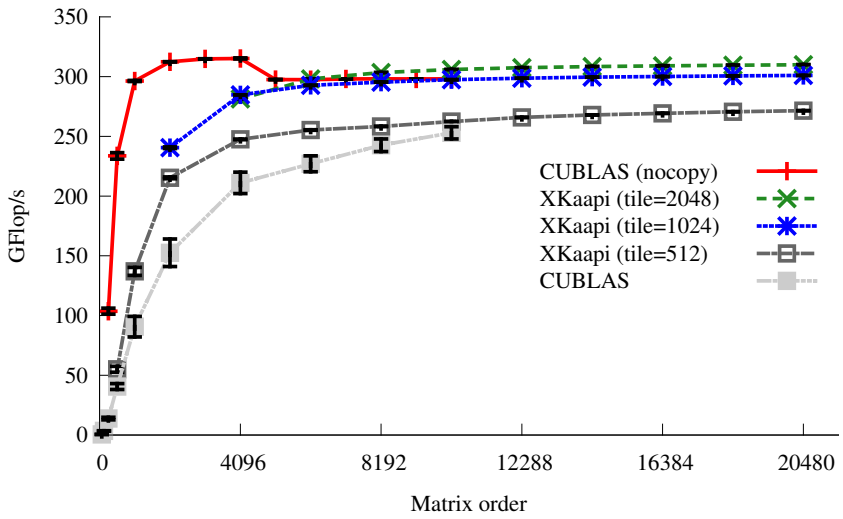


Figure 5.6 – Performance results from `DGEMM` on Idgraf for a single CPU and a single GPU, and different block sizes.

Our XKaapi version, that takes data transfers into account, with blocks of size 1024×1024 and 2048×2048, reached the GPU peak performance for matrices bigger than 6144×6144. For matrices bigger than 8192×8192 XKaapi's implementation sustained 309 GFlop/s for a 2048×2048 block size, outperforming CUBLAS (293 GFlop/s).

Thanks to the XKaapi software cache and to our design to exploit concurrent GPU operations, our blocked `DGEMM` algorithm sustained 309 GFlop/s performance peak even after the GPU runs out of memory with matrices larger than 10240×10240, which require ≈ 2.43 GB of device memory out of the 3 GB available on the NVIDIA Tesla C2050 GPUs. XKaapi version with block size of 1024×1024 generates tasks that can be exploited by our runtime to pipeline and overlap data transfers with computations. Our results suggest that we are able to overlap an important amount of the data transfers with the GPU kernel executions.

For small matrices, because the number of tasks remains low, data transfers were not entirely overlapped by computation. Even in this case, XKaapi attained significant results. For instance, the performance of *CUBLAS nocopy* with matrices of size 2048×2048 was about 312 GFlop/s. It dropped to 152 GFlop/s if we take into account the data transfers. Our XKaapi `DGEMM` for this matrix dimension and with block size of 1024×1024 generates 8 tasks for each sub-matrix product, and it reached 240 GFlop/s, which corresponds to 157% of improvement over *CUBLAS* version, which has data transfer costs.

5.6.4 Performance Results

Single-CPU and Single-GPU

We compared our work stealing based runtime XKaapi to StarPU (Augonnet et al., 2011) and single-GPU MAGMA (Tomov et al., 2010). StarPU schedules at runtime the entire task graph using the HEFT static scheduling algorithm. In XKaapi and StarPU, the Cholesky factorization of the diagonal block is sequential and executed on the CPU. MAGMA is a hand tuned library that can use up to one GPU. The work distribution is hand coded into the MAGMA library. The MAGMA version uses a more sophisticated implementation where part of the diagonal block factorization is exported on the GPU.

Figure 5.7 on the next page reports our results using one CPU and one GPU for Cholesky factorization. XKaapi and StarPU, with runtime scheduling decisions, outperformed MAGMA when the matrix is bigger than 10240×10240. The whole matrix size is about 800 MB, and can be stored into the 3 GB of device memory. Only the last matrix of dimension 20480×20480 can not be stored into the GPU memory. The main difference between XKaapi and StarPU *versus* MAGMA is that MAGMA is unable to exploit parallelism with dynamic scheduling, as is done by XKaapi and StarPU.

For small matrix dimensions (less than 2048×2048), the performance of MAGMA and XKaapi were similar, but StarPU seems to suffer from higher overhead. XKaapi had a little drop and then reached StarPU's performance.

Multi-GPU

For the multi-GPU evaluation, our experiments measured the performance of `DGEMM` and Cholesky using from 1 to 8 GPUs, with matrix dimension of 16384×16384 and block size of 1024×1024. Figure 5.8 on the facing page shows the performance results for `DGEMM` using XKaapi and StarPU. XKaapi outperformed StarPU in all cases and attained 2023.14 GFlop/s (or speed-up 6.74 on 8 GPUs with respect to single-GPU).

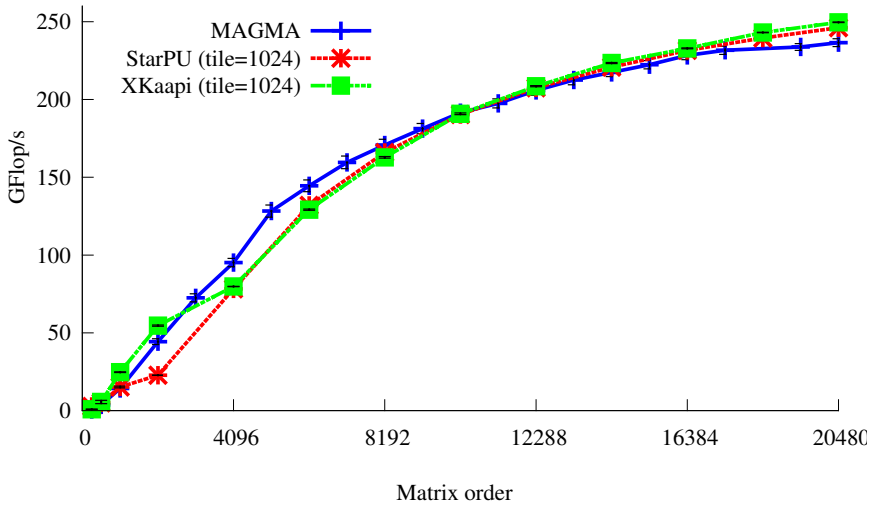


Figure 5.7 – Cholesky performance results on Idgraf for single-CPU and single-GPU with block size 1024×1024.

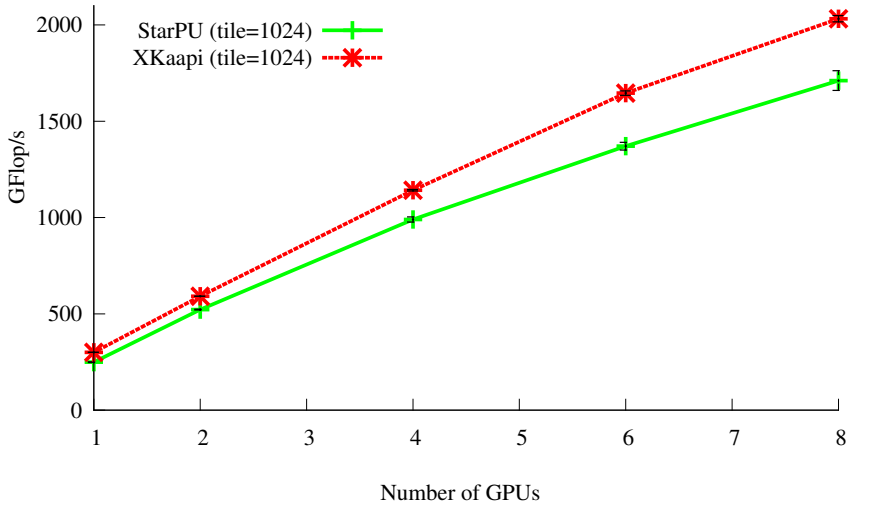


Figure 5.8 – `DGEMM` performance up to 8 GPUs. The matrix size was 16384×16384 with block size 1024×1024.

Figure 5.9 on the next page reports the results for the Cholesky factorization. In StarPU and XKaapi programs, all tasks, except the diagonal factorization `DPOTRF`, were performed by GPUs. Unlike the `DGEMM` case, the Cholesky factorization acceleration, up to 8 GPUs, was below the expected: neither XKaapi nor StarPU implementations did scale. StarPU reached 680.82 GFlop/s (or speed-up 2.94 with respect to single-GPU). Experiments with bigger matrices (up to 20480×20480) showed the same behavior. This means that, when using more that 4 GPUs, communications costs may not be neglected.

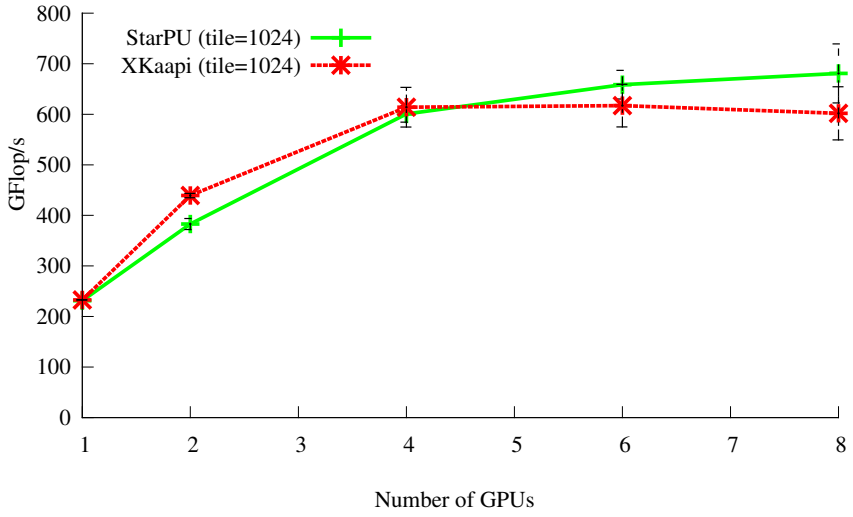


Figure 5.9 – Multi-GPU Cholesky factorization with 16384×16384 matrices and block size 1024×1024.

Memory Transfers

XKaapi and StarPU allow to monitor the execution by collecting *post-mortem* traces of performance counters. We have collected, for one instance of `DGEMM` and Cholesky, the total number of bytes exchanged between the main memory and the GPUs.

Table 5.1 shows in Gigabytes (GB) the total amount of memory transfers with StarPU and XKaapi on the `DGEMM` benchmark for matrix size 16384×16384. Surprisingly, StarPU generated bigger data exchanges than XKaapi up to 4 GPUs, although it uses an *a priori* HEFT algorithm. For a matrix of size S bytes, if the GPU memory could store the entire data, `DGEMM` implies $3 \times S$ bytes from host to device transfer and S bytes of transfer to get back the result. For a double precision matrix of dimension $16384 = 2^{14}$, the data transfer volume is 8 GB.

Table 5.1 – Memory transfers of `DGEMM` in GB with matrix order 16384×16384 and block size 1024×1024. The sum of input and output data transfers is 8 GB.

	`DGEMM` transfers (GB)				
GPUs	1	2	4	6	8
XKaapi	8.00	10.27	16.49	23.23	29.29
StarPU	22.54	14.97	16.98	20.09	24.35

On the Cholesky factorization, as illustrated in Table 5.2, the footprint generated by XKaapi was larger than StarPU. These results seem consistent with our hypothesis that work stealing is cache-unfriendly and may degrade performance. Indeed, HEFT scheduling of StarPU may minimize the makespan and reduce data footprint.

We previously showed that bad scaling of Cholesky factorization exhibited on Figure 5.9

Table 5.2 – Memory transfers of Cholesky in GB with matrix size 16384×16384 and block size 1024×1024. The sum of input and output data transfers is 4 GB.

	Cholesky transfers (GB)				
GPUs	1	2	4	6	8
XKaapi	3.71	7.38	12.28	12.89	14.62
StarPU	2.23	3.81	6.55	7.38	9.00

on the preceding page may be due to bad overlap of communication by computation. Table 5.2 evidences why XKaapi performed worst than StarPU. XKaapi had more data transfers. Moreover, when using more than 4 GPUs, the architecture share PCIe 16x links between GPUs (see Section 2.1.5 on page 16). Hence, bottlenecks on data transfers became key factor on performance.

Using work stealing directed by data affinity would allow XKaapi to reach StarPU performances. But the amount of data transfers would still be a bottleneck. To overcome this limitation, the parallel algorithm would use bigger blocks on GPU. Still, using bigger blocks means that `DPOTRF` would become a bottleneck as this task is currently run only on CPU. A parallel implementation, partially on GPU (such as MAGMA), for this task would then be required.

5.6.5 Comparison of Work Stealing Heuristics

Here we compared the performance of our work stealing heuristics (see Section 5.5) against the default work stealing algorithm (label *default*) for matrix product (`DGEMM`) and Cholesky factorization (`DPOTRF`) benchmarks. The matrix size was constant (40960×40960) while we vary the number of GPUs.

Parallel Matrix Product

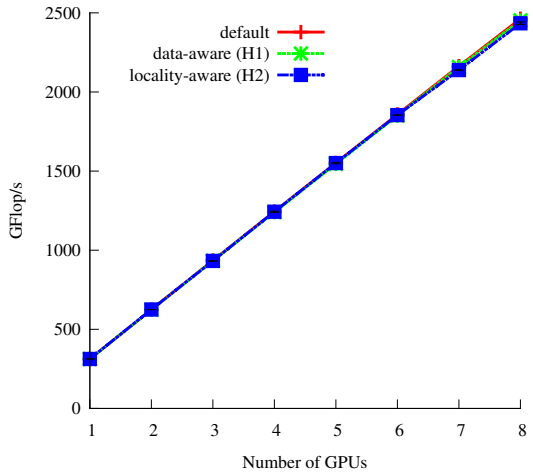
Figure 5.10 on the following page reports the performance of the parallel `DGEMM` using up to eight GPUs. For all heuristics, speedup linearly increased with the number of GPUs (Figure 5.10a). The peak performance was 2426.40 GFlop/s for 8 GPUs (2.43 TFlop/s). This corresponds to a sustained performance of 303 GFlop/s per GPU, which was very close to the peak (315 GFlop/s) on the `DGEMM` kernel. For matrices of size 16384×16384 XKaapi attained 2.0 TFlop/s, and 1.6 TFlop/s for matrices of size 8192×8192.

Our three heuristics showed similar GFlop/s performance. When looking at the total amount of data transfered (Figure 5.10b) the locality-aware heuristic (H2) outperformed the two other approaches with transfers reduced up to 24%. Clearly, the overlapping capability of XKaapi allowed to mask almost all the delays in data transfers. It appears that the different heuristics in this case do not impact performance.

Cholesky factorization

In addition to GPUs, we involved all remaining CPU cores in computations, out of the 12 available, after removing the ones each GPU monopolizes to run its GPU worker.

(a) Performance in GFlop/s.



(b) Total data transfers in (GB).

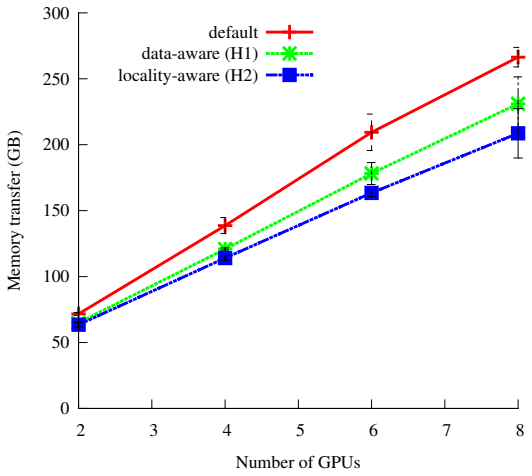


Figure 5.10 – Comparison of our three work stealing heuristics for `DGEMM` on 8 GPUs for a matrix size of 40960 × 40960.

Figure 5.11 on the next page illustrates the obtained results. We conclude that: (a) the default heuristic had a bigger communication footprint that may explain its poor scalability on more than four GPUs; (b) data-aware heuristic (H1), which may reduce the communication footprint, enabled a gain in scalability up to six GPUs; (c) locality-aware heuristic (H2) had the lowest volume of data transfers and scaled up to 8 GPUs. The peak performance with H2 was 1.79 TFlop/s in double precision and 3.92 TFlop/s in single precision.

We note that with more than 4 GPUs, at least 2 GPUs share the same PCIe-16x bus. Consequently, a scheduling algorithm that introduces a lot of memory transfer is more penalized on such hardware.

Scalability of the Cholesky Factorization

Figure 5.12 on the facing page gives an overview of the performance that have been achieved on the Cholesky factorization for different matrix sizes on 8 GPUs and 4 CPUs using our two heuristics and default work stealing. Except for matrices of size 4096 × 4096, which results were almost equal, locality-aware (H2) had the best performance for all matrix sizes and scaled as the matrix size grows.

Overlap Impact

We refine the analysis of performance impact when data transfers are overlapped with kernel executions on multi-GPU. Figure 5.13 on page 76 shows the performance results of the Cholesky factorization using the default work stealing and our locality-aware (H2) heuristic on 4 CPUs and 8 GPUs. On the default strategy, the overlap improved performance by 160.28 GFlop/s for the largest matrices (40960 × 40960). The gain was significantly

(a) Performance in GFlop/s. (b) Total data transfers in (GB).

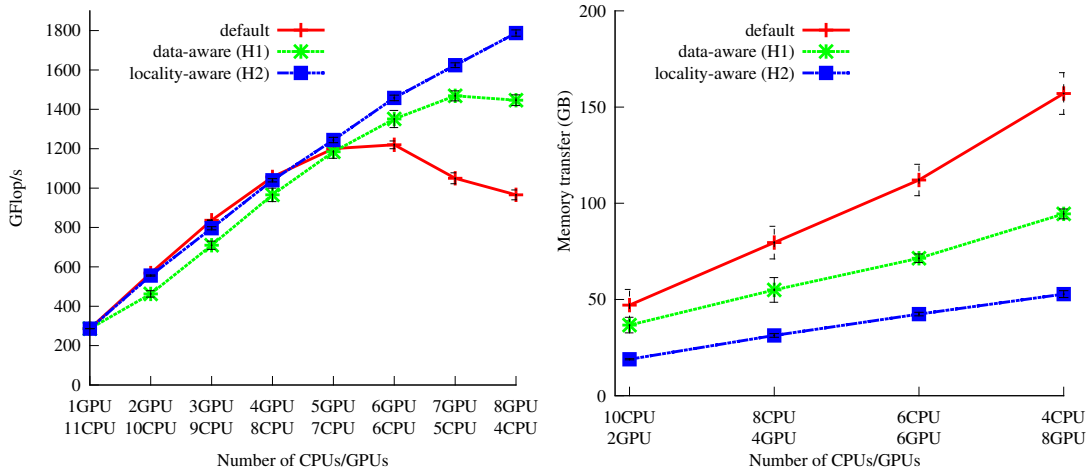


Figure 5.11 – Performance results of `DPOTRF` on eight GPUs and four CPUs for a matrix size of 40960 × 40960.

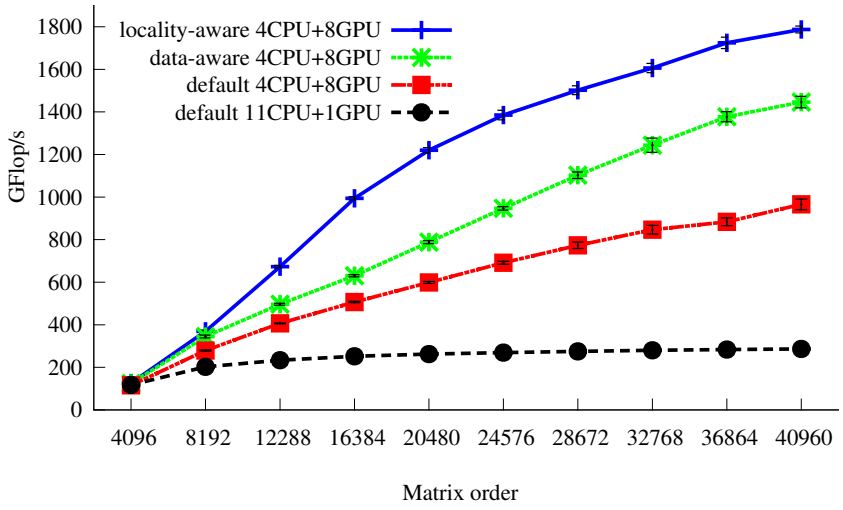


Figure 5.12 – Scalability of the work stealing heuristics for Cholesky on 8 GPUs and 4 CPUs compared to one GPU and one CPU execution.

higher for the locality-aware heuristic (H2), where the performance gain was about 550.48 GFlop/s for the largest matrices.

In addition, even in the cases without any overlapping, locality-aware heuristic improved performance over the default work stealing strategy by 431.45 GFlop/s for the largest matrices (40960 × 40960). These findings suggest that our locality-aware heuristic enables significant performance gains even without any concurrent operations.

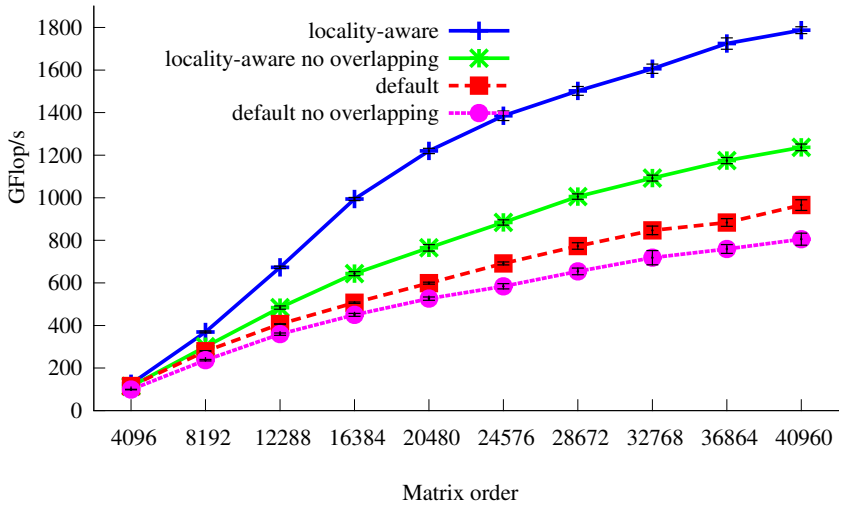


Figure 5.13 – Impact of overlapping in XKaapi default and locality-aware work stealing algorithm for Cholesky on 4 CPUs and 8 GPUs.

5.6.6 Multi-CPU Performance Impact

We analyze the gain of using several CPUs for the parallel-diagonal Cholesky factorization. Table 5.3 shows the performance results of the locality-aware heuristic (H2) when using up to 8 CPUs and 4 GPUs for different matrix sizes. Matrices up to a 16384×16384 size showed significant performance gains. For matrices larger than 32768×32768 the factorization did not benefit from additional CPUs. The reason for these different performance gains was the influence of the compute-bound tasks in the factorization. Level-3 BLAS operations such as `DGEMM` dominated the overall execution in a $O(N^3)$ growth order with respect to panel factorizations, which increase in order $O(N)$.

Table 5.3 – Performance results (in GFlop/s) using 4 GPUs and variable number of CPUs.

CPUs	Matrix order				
	4096	8192	16384	32768	40960
1	53.85	206.38	622.55	962.21	1052.58
	±0.98	±2.70	±7.90	±31.77	±20.53
4	115.16	391.05	755.91	1013.65	1022.45
	±1.02	±2.64	±6.89	±7.81	±37.55
8	138.34	439.70	782.21	999.46	1045.53
	±1.06	±3.38	±10.51	±6.90	±4.19

To illustrate executions, Figure 5.14 on the next page displays the Gantt diagrams for two configurations. On the top, one CPU and 4 GPUs compute the parallel-diagonal Cholesky factorization of a small matrix of size 6144×6144. This configuration reached a performance of 122.61 GFlop/s. We can see that GPUs were idle, because they waited for the panel factorization performed by the CPU (the factorization task is on the critical path

of the execution). By increasing the number of CPUs to 4 (bottom part of the figure), the performance increased to 243.80 GFlop/s.

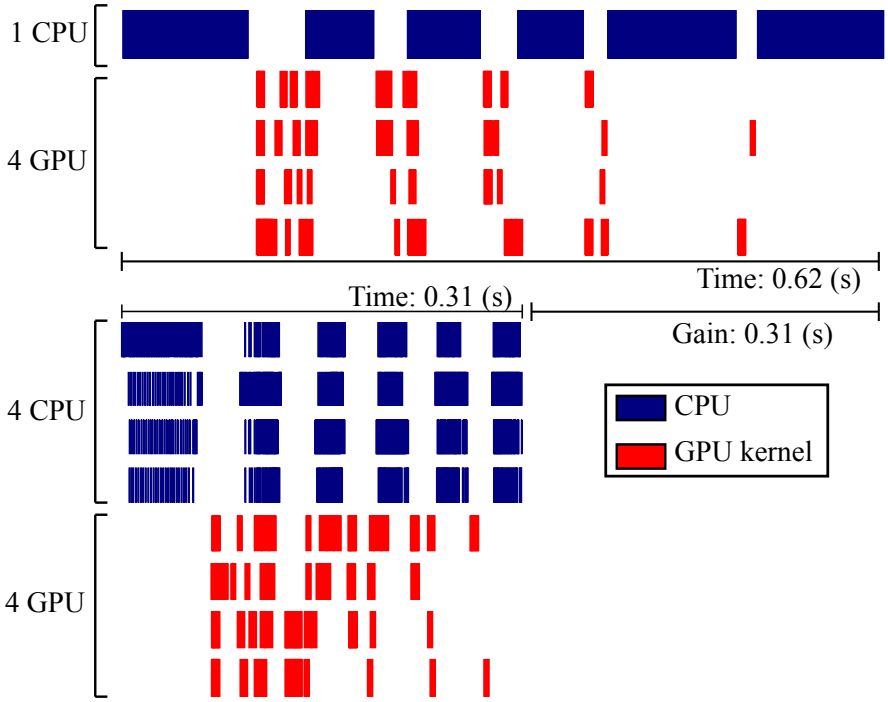


Figure 5.14 – Gantt chart from the parallel-diagonal Cholesky for a matrix size 6144×6144 on 4 GPUs (red) and up to 4 CPUs (blue).

These extra CPUs enabled to accelerate the panel factorization with a 8.4 GFlop/s gain per CPU, but more importantly they enabled to reduce the idle time of GPUs leading to a global gain of 121.18 GFlop/s. As reported by Kurzak et al. (2010), Buttari et al. (2009), and Song and Dongarra (2012) the acceleration of the tasks on the critical path (panel factorization) is important.

5.7 Summary

Most recent runtime systems incorporate support for heterogeneous architectures such as StarPU and OmpSs. It seems that a suitable programming model on these architectures may be based on task parallelism and data-flow dependencies for fine-grained parallelism. These aspects have been studied in a similar way for multicore runtime systems since fine-grained algorithms was essential to loose synchronization points and improve data locality (Buttari et al., 2009; Hermann, 2010).

In this Chapter, we presented the XKaapi runtime extensions for data-flow task programming on heterogeneous architectures. Algorithms on top of XKaapi express parallelism through tasks with dependencies without architecture details, and the runtime decides the target resource (CPU or GPU). In addition, it offers user annotations to pass scheduling hints.

We designed an asynchronous approach of concurrent GPU operations that achieved an almost ideal overlapping of data transfer and kernel execution with `DGEMM` algorithm for single-GPU. Thanks to this overlapping, the use of a dynamic work stealing algorithm permitted to reach high performance as the theory predicts for shared-memory machine without communication costs. Thus, overlapping almost enables to hide the heterogeneity of the memory accesses on a multi-GPU system.

Besides, the use of work stealing on multi-CPU and multi-GPU systems is one of the key contributions on scheduling for such architectures. Since classic work stealing is cache-unfriendly (Guo et al., 2010), tackling this problem is critical on such systems with disjoint memory spaces. We provided a work stealing scheduling with annotation at API level and a locality-aware work stealing based on local reduction of cache invalidations. This approach is similar to owner-computes rule (OCR) strategies since the runtime schedules tasks using access mode and metadata memory information of shared arguments.

Along with a work stealing strategy, a key contribution is the use of recursive CPU tasks to unfold parallelism at another level with fine-grained tasks. It is established that CPUs are efficient on fine-grained tasks and GPUs on coarse-grained tasks. Our experiments using a parallel-diagonal version of the Cholesky algorithm achieved significant performance results. These findings lead us to believe that different grain sizes may be a promising technique for heterogeneous systems. Furthermore, this concept can be expanded to exascale systems containing distributed and heterogeneous resources.

Nonetheless, our scheduling strategy based on work stealing lacks of more sophisticated decisions in order to consider processing power of available resources. We originally assumed that some tasks of a certain algorithm are more efficient on GPUs than CPUs. The obtained results with scheduling annotations seem consistent with our hypothesis. On the other hand, it is unlikely to achieve similar results for unknown tasks without empirical observations.

In the next Chapter, we overcome our scheduling limitation by a broader approach. We introduce a scheduling framework along with performance models for task and transfer prediction. Hence, scheduling algorithms such as work stealing and HEFT (Topcuoglu et al., 2002) are designed on top of our framework for XKaapi.

Chapter 6

Scheduling Strategies over Multi-CPU and Multi-GPU Systems

Contents

The works in this Chapter are part of an under reviewing paper.

With the recent evolution of processor design, future generations of processors will contain hundreds of cores. To increase the performance per watt ratio, the cores will be non-symmetric with few highly powerful cores and numerous, but simpler, cores. The success of these machines will rely on the ability to schedule the workload at runtime, even for small problem instances.

One of the main challenges is to define the scheduling strategy that may be able to exploit all potential parallelism on a heterogeneous architectures composed of multiple CPUs and multiple GPUs. Previous works demonstrate the efficiency of strategies such as static distribution (Dongarra et al., 2012; Horton et al., 2011; Song and Dongarra, 2012; Tomov et al., 2010), centralized list scheduling with data locality (Bueno et al., 2012), cost models (Agullo et al., 2010, 2011a,b; Augonnet et al., 2010b, 2011) based on Earliest-Finish-Time scheduling (Topcuoglu et al., 2002), and dynamic for a specific application

domain (Bosilca et al., 2012; Hermann et al., 2010). Nevertheless, few studies have reported on performance of different scheduling strategies for heterogeneous multi-CPU and multi-GPU platforms.

In our previous work, we state that the classic work stealing is cache-unfriendly and does not consider data locality (Gautier et al., 2013b; Lima et al., 2012). In Chapter 5, we describe a locality-aware work stealing that improves significantly the performance of compute-bound linear algebra problems such as matrix product and Cholesky factorization. However, it does consider the processing power of available resources.

In this Chapter, we study a broader approach to evaluate different scheduling strategies. We compare three different scheduling strategies for data-flow task programming on heterogeneous architectures: the locality-aware work stealing (Gautier et al., 2013b), the Heterogeneous Earliest-Finish-Time (HEFT) (Topcuoglu et al., 2002), and the distributed Dual Approximation algorithm (DDA) (Kedad-Sidhoum et al., 2013). The strategies are designed on top of the XKaapi scheduling framework with performance models for task and transfer prediction.

The remainder of this Chapter is organized as follows. Section 6.1 explains the XKaapi scheduling framework to design scheduling strategies. In Section 6.2 we introduce our performance models for task and data transfer prediction. Section 6.3 describes the algorithm details of the three scheduling strategies on top of XKaapi. Finally, Section 6.4 presents our experimental results on the Idgraf heterogeneous architecture.

6.1 Scheduling Framework

In most runtime systems, the scheduler is designed as a plug-in that interfaces with an API able to manage a list of tasks. Most of list algorithms consider a centralized management of the list. However, the cost of concurrent list access induces synchronization overhead that can not be ignored. A suitable approach is to distribute the list among workers and each manages its own list of tasks. For instance, work stealing and work pushing are popular distributed list scheduler algorithms.

We designed a framework in XKaapi in order to implement different scheduling strategies based on distributed list scheduling. Our interface is mainly inspired in work stealing and is composed of three operations: *pop*, *push* and *steal*.

Let us denote the operation's scope as *local*, whose manipulated list belongs to the current worker, and *remote* to a list not owned by the current worker. All three operations contain two parameters: a task list to manipulate (*local* or *remote*) and a task as input or output.

6.1.1 Overview of Task List and Task Descriptor

A XKaapi ready task list (*tasklist*) is a data structure for tasks with mutual exclusion to avoid race conditions. The list is double-ended (*deque*) and double-linked in order to remove and to insert an element at any position. By default XKaapi supports distributed list scheduling with one task list per worker. Still, a centralized strategy can create a global list of tasks. In this Chapter, all algorithms are based on distributed list scheduling and have one list per worker.

The task descriptor (*kaapi_taskdescr_t*) is an additional structure to encapsulate XKaapi tasks and contains task information and a list of its direct dependencies. It is used to optimize steal requests in XKaapi work stealing at runtime whenever a thief searches a stack for ready tasks for execution in a data-flow programming model (see Section 4.3.3 on page 53). In the context of our scheduling framework, based on the ready task list, we assume that scheduling strategies receive a task list with calculated true dependencies and created tasks.

6.1.2 A Distributed Scheduling Algorithm

Algorithm 5 illustrates a general scheduling loop of our scheduling strategies. At each iteration of the loop, either the own queue is not empty and the worker uses it or the worker emits a steal request to a randomly selected worker in order to get a task to execute. In Figure 6.1 we show a flowchart to represent the scheduling loop. Due to dependencies, once a worker executes a task, it calls the *activate* operation in order to activate its successors.

Algorithm 5: General scheduling loop of a worker w_j.

```
1  while Execution not terminated do
2  |  if Worker own queue is empty then
3  |  |  T ← steal from a random selected worker
4  |  |  if T ≠ ∅ then
5  |  |  |  local_push T into worker own queue
6  |  |  end
7  |  else
8  |  |  T ← pop from the worker own queue
9  |  |  Execute T
10 |  |  activate the task's successors of T
11 |  end
12 end
```

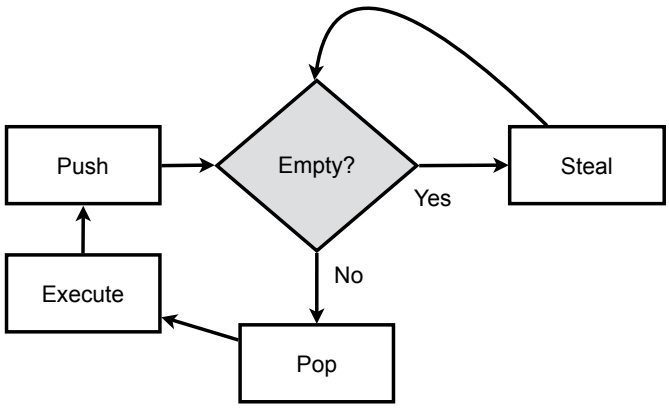


Figure 6.1 – General scheduling loop of the XKaapi scheduling framework.

Prologue and epilogue hooks give support to perform actions before and after task execution at line 9 of Algorithm 5, respectively. Augonnet et al. (2011) employ hooks to deal with inaccuracy or missing performance prediction in the context of the HEFT strategy. Our scheduling strategies also apply these hooks to calibrate performance models (Section 6.2) and correct erroneous predictions due to unpredictable or unknown behavior, such as operating system state or I/O disturbance. Let us note that those actions are optional and the scheduling strategy may decide if a performance model should be used.

All of our scheduling strategies follow this algorithm. The workers terminate their execution when all the tasks in the system are completed.

6.1.3 Scheduling by Pop, Push, and Steal

A framework interface for scheduling strategies is not a new concept in heterogeneous systems. Bueno et al. (2012) and Augonnet et al. (2011) described a minimal interface to design scheduling strategies and selection at runtime. But, there is little information available on the comparison of different strategies. Most of then reported performance on centralized list scheduling and performance model. We propose a framework based on work stealing to design scheduling algorithms derived from list scheduling. Our framework is composed of three basic operations (*pop*, *push*, and *steal*) along with additional methods. The C structure to describe a scheduling strategy is depicted in Figure 6.2.

A *pop* removes one task from the head of a task list for execution over the current worker. It is restricted to tasks capable of execution in the current worker, *i.e.*, tasks with an implementation to the current architecture type (CPU or GPU). Its scope is local to the current worker and may perform load balancing in centralized strategies. Thus, a pop can be issued to a remote task list with mutual exclusion.

The *push* method inserts one task at the head or the tail of a list. We designed two push versions depending on the list's scope: *local* and *remote*. A *local push* inserts a task at list's head and is commonly employed in the scope of a local task list to the current worker. Task insertion and removal from the same list position, in this case the list head, is a well-known technique to improve locality of tasks (Blumofe and Leiserson, 1998). An exception of the *local push* scope is centralized strategies in which there is one task list shared by a number of workers. Whereas, a *remote push* inserts tasks onto the tail of non-local list worker. This operation provides support for a number of additional list scheduling based strategies such as heuristics (Acar et al., 2000) and work pushing from cost models. In addition to both versions of *push*, a *push_ activated* inserts task successors into the list. It provides a way to apply cost model strategies over a set of ready tasks for execution.

A *steal* removes a task from the list's tail of a remote worker, or *victim*. Our *steal* operation is based on the classic work stealing algorithm detailed by Frigo et al. (1998). An idle thread, called a thief, initiates a steal request to a random selected victim. To find a ready task, a thief thread calls the *steal* from the framework passing as parameter the victim's task list. On reply, the *steal* returns a reference (or memory pointer) of one ready task. In our framework, XKaapi *steal* does not require the removal of the stolen task from the victim's list. The runtime only expects the selection of a task to be stolen and removed, which will be performed by XKaapi. A strategy can disable *steal* by not specifying a steal function.

```
1 typedef struct kaapi_sched_t {
2   /* Initialize the scheduling strategy. */
3   void (*init)(void);
4
5   /* Clean up this scheduling strategy. */
6   void (*finalize)(void);
7
8   /* Remove a task from this tasklist.
9    * Returns 0 if successful, EBUSY otherwise. */
10  int (*pop)( kaapi_tasklist_t*, kaapi_taskdescr_t** );
11
12  /* Insert a task into this tasklist. */
13  void (*push)(kaapi_tasklist_t* const,
14               kaapi_taskdescr_t* const);
15
16  /* Insert all activated tasks by a task into this tasklist.
17   * Note that the runtime has computed true depedencies and
18   * each task descriptor contains a list of tasks to be
19   * activated. Returns the number of activated tasks. */
20  uint32_t (*push_activated)(kaapi_tasklist_t* const,
21                             kaapi_taskdescr_t* const);
22
23  /* Remote a task from this processor victim. (optional) */
24  kaapi_taskdescr_t* (*steal)(kaapi_processor_t* const,
25                              kaapi_taskdescr_t* const);
26
27  /* Hook called before a task execution. (optional) */
28  void (*prologue)(kaapi_taskdescr_t* const);
29
30  /* Hook called after a task execution. (optional) */
31  void (*epilogue)(kaapi_taskdescr_t* const);
32 } kaapi_sched_t;
```

Figure 6.2 – Data structure in C of a scheduling strategy on XKaapi.

6.1.4 Prologue and Epilogue Hooks

Prologue and epilogue hooks give support to perform actions before and after task execution, respectively. It receives as parameter the actual task to execute. Augonnet (2011) studies the use of hooks to deal with inaccuracy or missing performance prediction in the context of the HEFT strategy. Our scheduling strategies also apply these hooks to calibrate performance models (Section 6.2 on the next page) and correct erroneous predictions due to unpredictable or unknown behavior, such as operating system state or I/O disturbance. Let us note that those actions are optional and the scheduling strategy may decide if a performance model should be used.

6.2 Performance Model

Cost models depend on a certain knowledge of both application algorithm and the underlying architecture to predict performance at runtime. In order to predict performance, we designed a performance model for task execution time and communication, similar to StarPU (Augonnet et al., 2010b). Our task prediction is based on history-based model, and transfer time estimation is based on asymptotic bandwidth. They are associated with scheduling strategies to predict task completion time such as HEFT.

6.2.1 Predicting Data Transfer

On a multi-GPU system it is important to estimate data transfer time in order to decide if its better to migrate a computation to another PU, sometimes in a distant memory node. XKaapi memory management keeps track of data replicas and is able to receive queries about data state on a certain resource and where a valid replica can be found.

Our performance model can predict communication transfer by asymptotic bandwidth, which is benchmarked through offline *sampling* of the PCIe latency and bandwidth. When the runtime initializes by the first time in the machine it begins the sampling procedure. This sampling detects all available resources (CPUs or GPUs) and performs a series of *ping-pong* benchmarks by measuring both the bandwidth and the latency between each pair of resource (from a CPU to a GPU for instance). In addition, we collect peer-to-peer transfers between GPUs when available, depending on the PCIe topology. All collected values are stored in a text file and loaded each time a scheduling strategy requests XKaapi performance modeling.

Each task prediction considers the state of meta-data information in the memory and calculates the transfer time by the number of bytes from one device to the host, or *vise versa*. At the transfer prediction, the model calculates the necessary transfers to execute a task on a CPU or GPU depending on its meta-data state on the target processor.

Let us assume that $t_{i \to j}$ is the predicted transfer time from worker i to worker j given n bytes of data to transfer, $B_{i \to j}$ the stored bandwidth, and $L_{i \to j}$ the latency. An estimation of transfer time between workers i and j can be derived from the bandwidth ($B_{i \to j}$) and latency ($L_{i \to j}$):

$$t_{i \to j} = \frac{n}{B_{i \to j}} + L_{i \to j} \tag{6.1}$$

Nevertheless, multi-GPU systems may have to consider bus contention due to multiple data transfers and other I/O activities not related to the parallel application. Modeling data transfer is a difficult problem since a lot of interactions may occur between I/O devices and the entire memory subsystem. We adopt the transfer modeling proposed by Augonnet (2011) with a slight modification. While Augonnet (2011) divides the bandwidth in Equation 6.1 by all accelerators in the system, we divide this bandwidth only by the actual accelerators in use at runtime. This division aims to consider the performance bottleneck of concurrent data transfers from or to distant workers. Equation 6.2 gives the estimated transfer time for a number of n bytes from worker i to j.

$$t_{i \to j} = \frac{n}{\frac{B_{i \to j}}{n_{accel.}}} + L_{i \to j} = \frac{n}{B_{i \to j}} \times n_{accel.} + L_{i \to j} \tag{6.2}$$

A more accurate transfer model may depend on a deep knowledge of the I/O and memory system. Besides, this strategy may also require a deeper understanding of the operating system and its subsystems related to I/O and memory. In the XKaapi performance model we do not take into account those aspects for the sake of simplicity. We do not consider, for instance, the number of hops to transfer data such as PCIe bridges in recent machines with multiple GPUs. On the other hand, we assume that the estimated bandwidth from offline sampling may implicitly reflect the penalty of hops to transfer over PCIe bridges. Moreover, our performance results on overlapping data transfers (see Section 5.6 on page 67) led us to infer that transfer costs would have fewer impact on the overall performance.

6.2.2 Performance Modeling of Tasks

The XKaapi performance model for task prediction relies on a history-based model for regular computations. We chose to develop an online performance model transparent to the application that does not depend on external tools and manual tuning.

The choice of a performance model for tasks is related to its work load characteristics. It may depend on the input size and layout, or the input contents as for irregular computations. For instance, the tiled algorithms for dense linear algebra divide the input in square blocks of equal size. Even it is possible to adapt granularity at runtime, executions with the same matrix size may use the same set of block sizes with slight variations. The runtime can take advantage of this regularity of data and work load by storing the execution time of each task based on input layout and size, plus worker type (CPU or GPU).

We use three task fields to identify a performance entry: *task name*, *input size* (footprint), and *processor type*. The key to identify an unique hash entry of tasks is then composed of $(task_{name}, task_{footprint}, p_{type})$. At each executed task the scheduling strategy can make use of *prologue* or *epilogue* hooks to update a task entry in order to be applied by the next execution. Still, we note that enabling task model sampling on GPUs may affect performance since the runtime has to disable asynchronous execution in order to know its actual completion time.

A limitation of a history-based model is to assume that performance is only dependent of data input size and layout, and independent of the actual data contents. An example of unpredictable task duration is the pivoting phase of the LU panel factorization.

6.3 Scheduling Strategies on Top of XKaapi

This Section introduces our three scheduling strategies designed on top of the XKaapi scheduling framework. We describe the locality-aware work stealing (Gautier et al., 2013b), the Heterogeneous Earliest-Finish-Time (HEFT) (Topcuoglu et al., 2002) and the distributed Dual Approximation (DDA) (Kedad-Sidhoum et al., 2013). Here we consider a multicore parallel architecture with m homogeneous CPUs and k homogeneous GPUs. Let us denote by p_i^{CPU} the processing time of task T_i on a CPU and p_i^{GPU} on a GPU.

6.3.1 Locality-Aware Work Stealing

The locality-aware work stealing (H2) is a heuristic for the classic work stealing in order to overcome its cache-unfriendly problem. The algorithm was introduced in our previous work on multi-GPU scheduling and detailed in Section 5.5.3 on page 66. The main difference relies on the *push* of task's successors to selected remote workers based on meta-data information attached to each user data (Gautier et al., 2013b). This heuristic resembles an "owner-computes rule" and is similar to the approach proposed by Acar et al. (2000), but without explicit annotation. H2 tries to reduce the invalidations of the data replicas through a *remote push* of a newly ready task on the worker that has a valid copy of its data parameters on *write* or *exclusive* access mode. If more than one worker is eligible, then it simply selects a worker at random among the set of possible workers. We note that this task pushed to a remote worker may be stolen by an idle worker.

6.3.2 Heterogeneous Earliest-Finish-Time

The Heterogeneous Earliest-Finish-Time (HEFT), proposed by Topcuoglu et al. (2002), is a scheduling algorithm for a bounded number of heterogeneous processors. It has two major phases: *task prioritizing* for computing the priorities of all tasks and a *worker selection* phase to select the "best" worker, which minimizes the task's finish time. The HEFT algorithm complexity is $O(v^2 \times p)$ for v tasks and p workers (CPUs plus GPUs). More details on this algorithm are given in Section 2.3.2 on page 25.

Our XKaapi HEFT scheduler is based on the algorithm presented in Section 2.3.2 on page 25 and implements both phases (task prioritizing and worker selection) at activation of the task's successors (*push_ activated* operation). The *task prioritizing* phase calculates for all ready tasks a speedup $S_i = \frac{p_i^{CPU}}{p_i^{GPU}}$ relative to a GPU execution. Next, it sorts the list of ready tasks by S_i in decreasing order. In the *worker selection* phase, the algorithm selects tasks in the order of their speedup S_i and schedules each task on its "best" worker, which minimizes the task's finish time. Algorithm 6 describe the basic steps of the HEFT strategy over XKaapi.

Algorithm 6: Heterogeneous Earliest-Finish-Time (`push_activated`).

Input : A task descriptor td
Output: A list of tasks tasklist

foreach *task i activated by task descriptor* td **do**
  $S_i \leftarrow \frac{p_i^{CPU}}{p_i^{GPU}}$
end
Sort activated tasks by decreasing speedup S_i
foreach *task i activated by task descriptor* td **do**
  Schedule task i to minimize finish time on a worker p_i
  Remote push of task i into tasklist of worker p_i
  Update dates of worker p_i
end

Although the original HEFT considers that the appropriate time slot on a worker p_i starts when all input data of a task T_i is available at p_i, we define the appropriate time slot when the worker p_i completes the execution of its last assigned task. Our HEFT algorithm incorporates data communications onto the search of an appropriate idle time slot by time prediction of data transfer for each worker (CPU or GPU) according to the task data state.

We added to HEFT performance prediction two constants called "factors" for the execution (α) and communication (β) time. The predicted time for a task is defined by $T_i = \alpha p_i + \beta T_i$ where α modifies the execution time p_i and β the total predicted transfer time.

6.3.3 Distributed Dual Approximation

The principle of the distributed Dual Approximation (DDA) algorithm is based on a dual approximation (Hochbaum and Shmoys, 1987). Let us recall that a k-dual approximation scheduling algorithm considers a *guess* λ (which is an estimation of the optimal makespan) and either delivers a schedule of makespan at most $k\lambda$ or answers correctly that there is no schedule of length at most λ. The process is repeated by a classical binary search on λ up to a targeted precision of ϵ (Kedad-Sidhoum et al., 2013).

Our DDA implements the scheduling algorithm at activation of the task's successors (*push_ activated* operation). Algorithm 7 on the next page illustrates the designed algorithm, which consists on the following steps:

- Choice of the initial guess $\lambda = \sum_i \frac{max(p_i^{CPU}, p_i^{GPU})}{2}$ (lines 2 and 4);
- Extract the tasks which fit only into GPUs ($p_i^{CPU} > \lambda$), and symmetrically those which are dedicated to CPU (line 5);
- Keep this schedule if the tasks fit into λ (line 8). Otherwise, reject this schedule if there is a task larger than λ on both CPUs and GPUs (line 11);
- Add to the tasks allocated to the GPU those which have the largest speedup relative to GPU time (defined by $\frac{p_i^{CPU}}{p_i^{GPU}}$) up to reaching the threshold λ (line 14);
- Put all the remaining tasks in the m CPUs and execute them using a earliest-finish-time scheduling policy (line 14).

We also added to DDA performance prediction two constants called "factors" for the execution (α) and communication (β) time. The predicted time for a task is defined by $T_i = \alpha p_i + \beta T_i$ where α modifies the execution time p_i and β the total predicted transfer time.

6.4 Experiments

This Section presents the experimental results of our scheduling strategies designed through the XKaapi scheduling framework on a heterogeneous multi-CPU and multi-GPU architecture (6.4.1). We describe the benchmarks from the PLASMA library (6.4.3) and the

Algorithm 7: Distributed Dual Approximation (`push_activated`).

Input : A list of ready tasks tasklist

$lower \leftarrow 0$
$upper \leftarrow \sum_j max(p_i^{CPU}, p_i^{GPU})$
while $(upper - lower) > \epsilon$ **do**
    $\lambda \leftarrow \frac{upper+lower}{2}$
    Schedule tasklist into λ to minimize finish time
    if *tasks do fit into* λ **then**
        $upper \leftarrow \lambda$
        Keep current schedule
    else
        $lower \leftarrow \lambda$
        Reject current schedule
    end
end
Push tasklist based on the last schedule that fits λ

methodology of our experiments (6.4.2). At last, we show the performance results with the scheduling strategies described in Section 6.3 (6.4.4) and their analysis (6.4.5).

Appendix B on page 149 reports the performance model results for the experimental results of this Chapter. The Appendix contains the task execution time from the history-based model, and the asymptotic bandwidth values from XKaapi offline sampling.

6.4.1 Platform and Environment

All experiments have been conducted on the heterogeneous, multi-GPU system, named "Idgraf" (see Section 2.1.5 on page 16). Figure 6.3 on the facing page illustrates the hardware topology of Idgraf with two hexa-core CPUs and eight Tesla C2050 GPUs. We used as software environment GNU/Linux Debian *squeeze* x86/64, the compiler GCC 4.4, CUDA 5.0, and the library ATLAS 3.9.39 (BLAS and LAPACK).

6.4.2 Methodology

All factorizations were in double precision floating-point operations. We repeated the number of executions (up to 30) and we report the mean, as well as the 95% confidence interval.

In each experiment, we show in the x-axis the number of resources as the number of GPUs or the number of CPUs and GPUs for each execution. We employ this notation to clearly distinguish the number of computing CPUs and GPUs at runtime. Since XKaapi dedicates a CPU to manage a GPU, the number of computing CPUs is the number of total CPUs minus the number of GPUs (see Section 5.2 on page 61 for details).

In the case of the locality-aware work stealing (H2), we denote as "*WS locality*" and suffix "*SetArch*" for annotated tasks to execute only on GPUs if a GPU version is available.

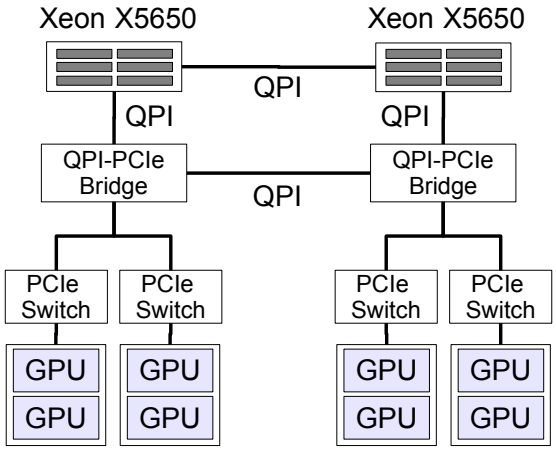


Figure 6.3 – Idgraf hardware topology for experimental results on scheduling strategies.

6.4.3 Benchmarks

Our experiments use the tiled algorithms of PLASMA (Buttari et al., 2009) for matrix product (`DGEMM`), Cholesky (`DPOTRF`), LU (`DGETRF`), and QR (`DGEQRF`). We implemented and extended the QUARK API (YarKhan et al., 2011) in XKaapi to support task multi-versioning. Each benchmark calls a registration function responsible to associate one PLASMA task to a GPU version. At the task execution, our QUARK version runs the appropriate task implementation if the target worker is a CPU or a GPU. The GPU kernels of QR and LU are based on previous works for heterogeneous architectures from Agullo et al. (2010, 2011a,b) and adapted from PLASMA CPU algorithm and MAGMA from Tomov et al. (2010).

We report the highest performance obtained from different block sizes (NB) on each combination benchmark plus scheduling strategy. Each figure has the strategy name and the block size used for the experiment. The internal block value (IB) for PLASMA tasks was 128×128.

Cholesky

The Cholesky factorization (`DPOTRF`) decomposes an $n \times n$ real symmetric positive definite matrix A into the form $A = LL^T$ where L is an $n \times n$ real lower triangular matrix with positive diagonal elements (Agullo et al., 2010). The tile Cholesky algorithm (`PLASMA_dpotrf_Tile`) has four kernels with a GPU version: `DPOTRF` (CPU/GPU), `DTRSM` (CPU/GPU), `DSYRK` (CPU/GPU), and `DGEMM` (CPU/GPU). We used the *SetArch* for all tasks with GPU version.

LU

The LU factorization (`DGETRF`) of a matrix A has the form $A = LU$ where L is lower triangular and U is upper triangular. Similar to Agullo et al. (2011a), our tile LU factorization from PLASMA (`PLASMA_dgetrf_incpiv_Tile`) has four kernels with a GPU version: `DGETRF` (CPU/GPU), `DGESSM` (CPU/GPU), `DTSTRF` (CPU/GPU), and `DSSSSM`

(CPU/GPU). We used the *SetArch* for tasks with GPU version and speedup greater than one (see Figure B.1 on page 149) over the CPU version: `DGESSM` and `DSSSSM`.

QR

The QR factorization (`DGEQRF`) of an $m \times n$ real matrix A has the form $A = QR$ where Q is an $m \times m$ real orthogonal matrix and R is an $m \times n$ real upper triangular matrix (Agullo et al., 2011b). Our tile QR factorization from PLASMA (`PLASMA_dgeqrf_Tile`) has four kernels in which two have a GPU version: `DGEQRF` (CPU), `DORMQR` (CPU/GPU), `DTSQRT` (CPU), and `DTSMQR` (CPU/GPU). We used the *SetArch* for all tasks with GPU version.

6.4.4 Performance Results

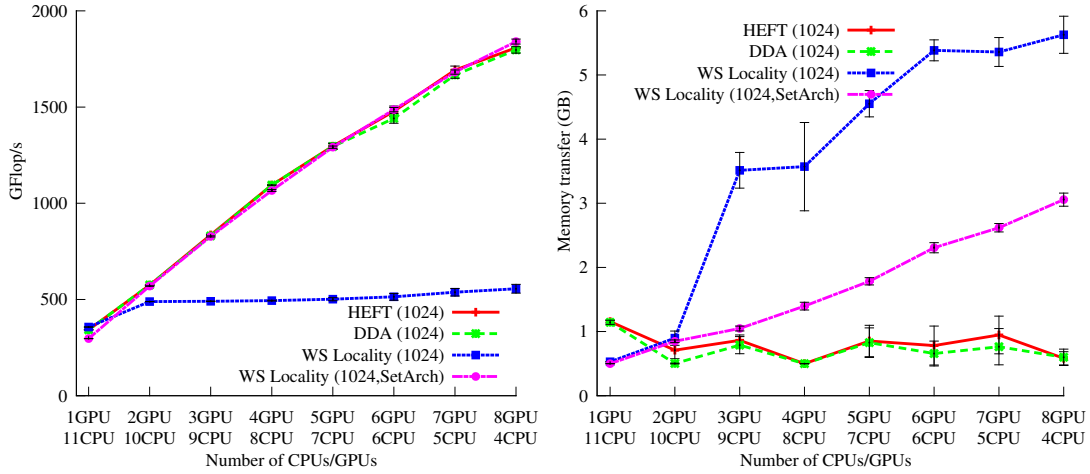
We present in this Section the results with matrix sizes 8192×8192 and 16384×16384 in order to evaluate the behavior of each scheduling strategy with different work loads.

Figure 6.4 on the next page shows the experimental results with matrix multiplication (`DGEMM`) in respect with performance (Figures 6.4a and 6.4c) and total memory transfer (Figures 6.4b and 6.4d). The cost models and the locality-aware with *SetArch* had a similar performance behavior and displayed a gap of up to 32.13 GFlop/s. HEFT and DDA strategies attained the peak performance of 1808.24 GFlop/s and 1797.35 GFlop/s with 8192×8192 matrix, respectively, while the locality-aware work stealing with *SetArch* annotation was slightly better with 1840.37 GFlop/s. This same pattern was observed in 16384×16384 matrix with peak performance of 2160.17 GFlop/s for work stealing with *SetArch*. In addition, work stealing had the highest communication footprint compared to other strategies in both inputs (5.62 GB and 42.22 GB) but attained significant performance results for matrix 16384×16384 (Figure 6.4c). These results suggest that the performance on highly compute-bound problems have direct relation with the communication footprint. Nevertheless, the work stealing strategy with locality and *SetArch* slightly outperformed cost models.

Figure 6.5 on page 92 shows the performance results with the Cholesky factorization (`DPOTRF`) in respect with performance (Figures 6.5a and 6.5c) and total memory transfer (Figures 6.5b and 6.5d). The cost models and the locality-aware work stealing with *SetArch* had a similar performance behavior. The HEFT strategy outperformed the work stealing with *SetArch* with matrix 8192×8192 (difference of $\approx$ 81.41 GFlop/s), but it it was surpassed by work stealing *SetArch* with matrix 16384×16384 ($\approx$ 30.81 GFlop/s). In most cases the HEFT strategy outperformed the others, except for matrix 16384×16384 from 4 GPUs (Figure 6.5c). Besides, the obtained performance results have direct relation with communication footprint. The lowest footprint with 8 GPUs was attained by HEFT for matrix 8192×8192 (2.31 GB) and by work stealing *SetArch* for matrix 16384×16384 (11.31 GB). As expected, work stealing showed the highest footprint in both matrix sizes. DDA strategy, even including data transfers in prediction, had high footprint that affected the performance in both inputs.

Figure 6.6 on page 93 reports the experimental results with the LU factorization (`DGETRF`) in respect with performance (Figures 6.6a and 6.6c) and total memory transfer (Figures 6.6b and 6.6d). With the exception of work stealing, all strategies showed

(a) `DGEMM` (8192 × 8192). (b) Transfer in GB (8192 × 8192).



(c) `DGEMM` (16384 × 16384). (d) Transfer in GB (16384 × 16384).

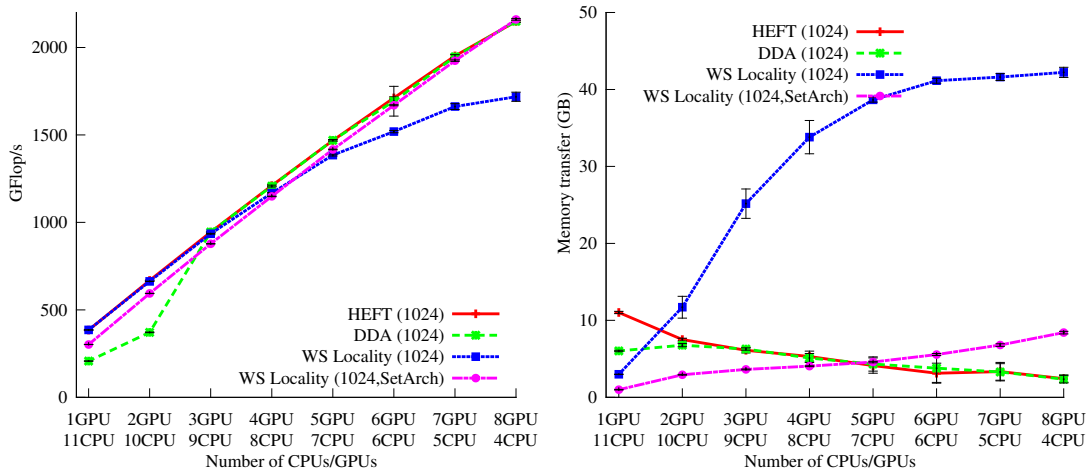
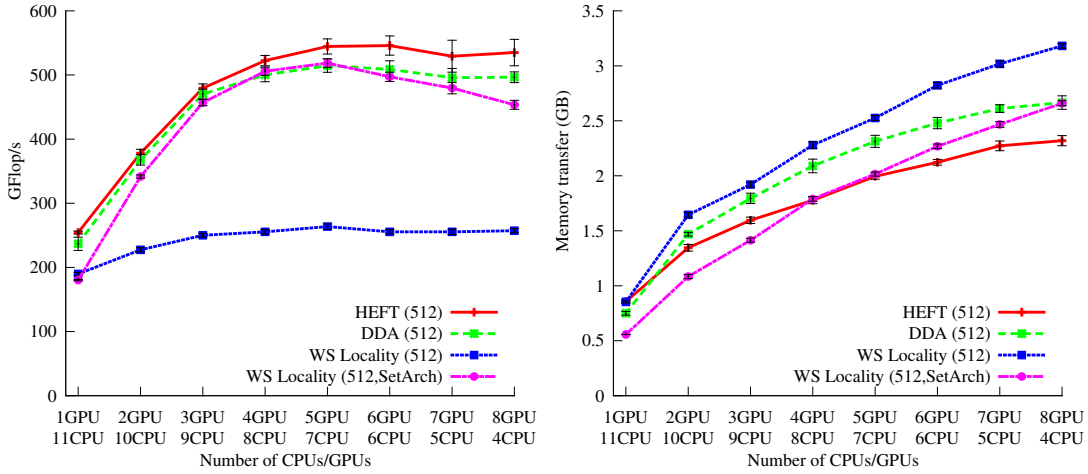


Figure 6.4 – Performance results of matrix product (`DGEMM`).

similar performance results, whereas HEFT slightly outperformed DDA with peak performance of 136.04 GFlop/s (matrix 8192×8192) and 301.68 GFlop/s (matrix 16384×16384). Work stealing with *SetArch* outperformed cost models for some cases, for instance from 4 GPUs & 8 CPUs to 6 GPUs & 6 CPUs (matrix 8192 × 8192) and 3 GPUs & 9 CPUs (16384 × 16384). Nevertheless, LU did not scale for matrix 16384 × 16384 from 5 GPUs & 7 CPUs. Differently of `DGEMM` and Cholesky, the locality-aware work stealing with *SetArch* did not attain the expected performance results for matrix 16384 × 16384 since its implementation has two GPU tasks with *SetArch* attribute from a total of four tasks. Hence, its communication footprint was higher and the performance was partially affected since it is comparable to DDA and HEFT. The lowest footprint of LU with 8 GPUs was 5.31 GB (8192 × 8192) and 32.35 GB (16384 × 16384), both with HEFT strategy.

(a) `DPOTRF` (8192 × 8192). (b) Transfer in GB (8192 × 8192).



(c) `DPOTRF` (16384 × 16384). (d) Transfer in GB (16384 × 16384).

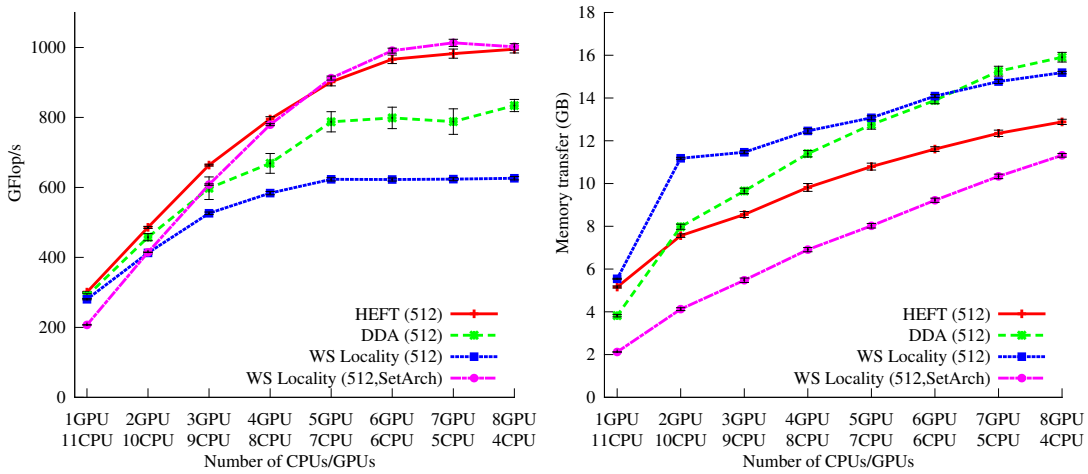
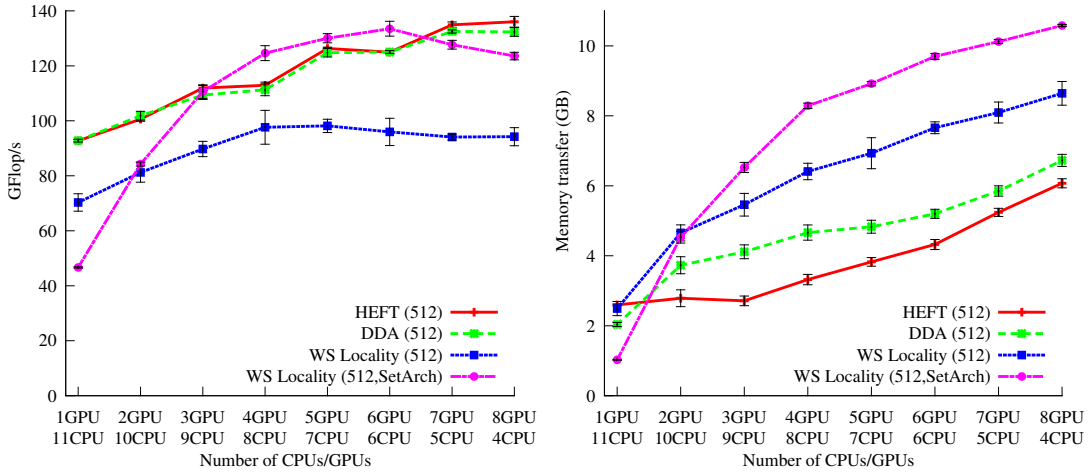


Figure 6.5 – Performance results of Cholesky (`DPOTRF`).

Finally, Figure 6.7 on page 94 reports the experimental results with the QR factorization (`DGEQRF`) in respect with performance (Figures 6.7a and 6.7c) and total memory transfer (Figures 6.7b and 6.7d). The performance results of HEFT outperformed the work stealing strategies for almost all cases, with peak performance of 119.10 GFlop/s (8192 × 8192) and 293.11 GFlop/s (16384 × 16384). Still, QR did not scale for matrix 16384 × 16384 from 5 GPUs & 7 CPUs. QR was the only test where work stealing showed performance results near the obtained peak, even with the highest footprint compared to other strategies. The cost model strategies had a minimal communication footprint compared to work stealing strategies, regardless of DDA that attained the lowest performance results. The lowest footprint of QR with 8 GPUs was 10.78 GB (8192 × 8192) and 72.86 GB (16384 × 16384), both with HEFT strategy.

(a) `DGETRF` (8192 × 8192). (b) Transfer in GB (8192 × 8192).



(c) `DGETRF` (16384 × 16384). (d) Transfer in GB (16384 × 16384).

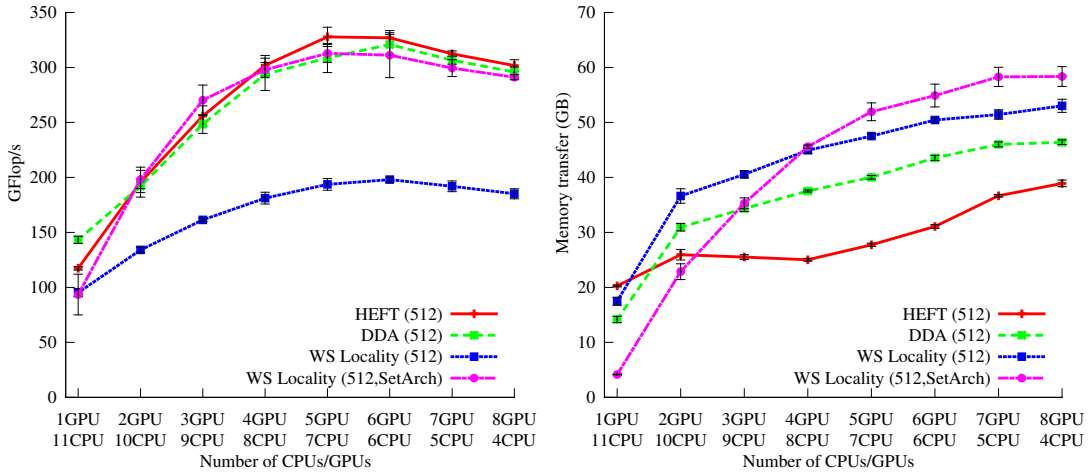
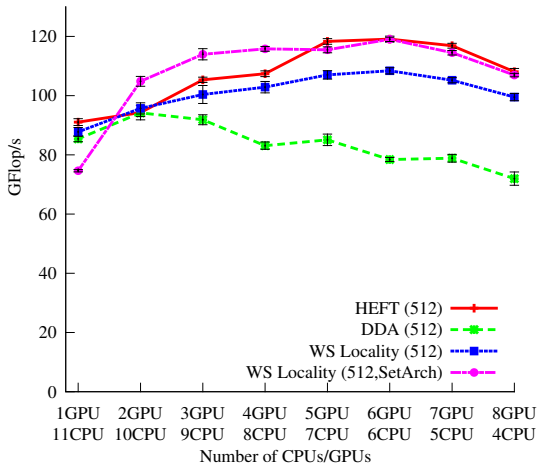


Figure 6.6 – Performance results of LU (`DGETRF`).

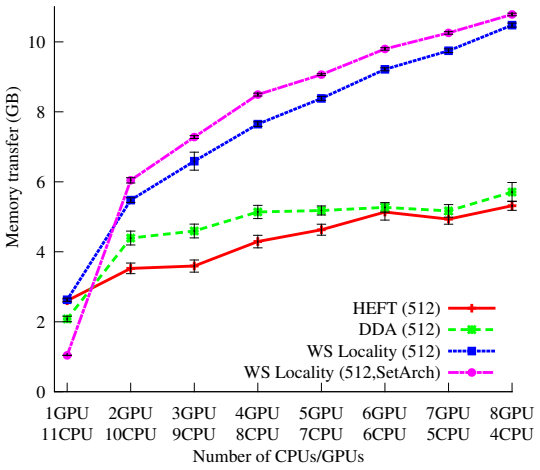
6.4.5 Discussion

The experimental results of the four benchmarks had similar behaviors with the addition of more GPUs depending on the problem nature. LU and QR performance did not scale like matrix multiplication or Cholesky for the matrix size 16384 × 16384. These results can be explained by assuming that LU and QR have fewer BLAS-3 tasks than the other two. Cholesky and `DGEMM` are compute-bound and its workload is dominated by BLAS-3 operations (*matrix-to-matrix*). Whereas, LU and QR present more BLAS-2 operations (*matrix-vector*) and did not profit of all GPU parallelism, specially in our algorithm version. Other works on LU and QR factorizations attained higher performance results on heterogeneous architectures (Agullo et al., 2011a,b), most of them are reported in single-

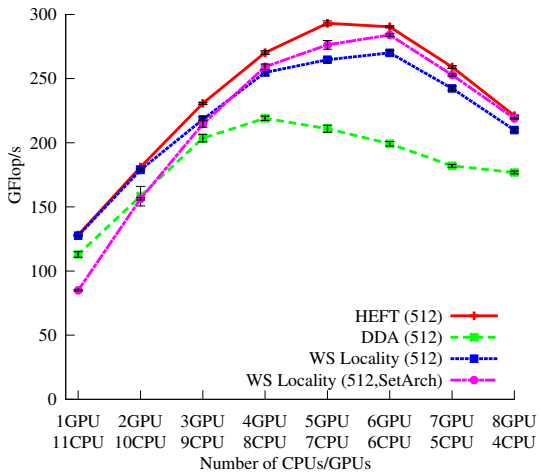
(a) `DGEQRF` 8192 × 8192.



(b) Transfer in GB (8192 × 8192).



(c) `DGEQRF` 16384 × 16384.



(d) Transfer in GB (16384 × 16384).

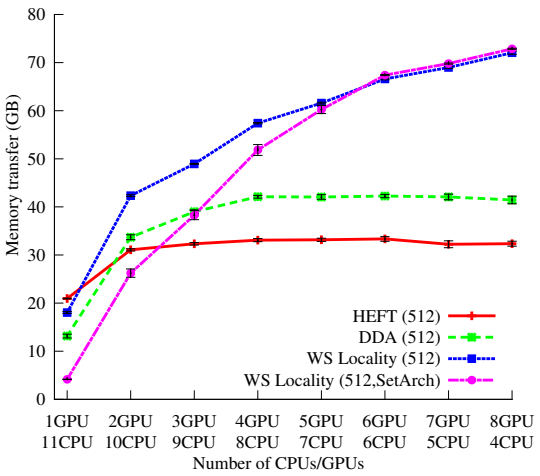


Figure 6.7 – Performance results of QR (`DGEQRF`).

precision floating point operations.

In addition, the HEFT strategy outperformed other strategies in most cases, and had the lowest communication footprint for `DGEMM`, LU and QR. One possible explanation is that the performance prediction allows to reduce footprint and exploit efficiently the target architecture. HEFT selects the "best" processor that minimizes the task's finish time, and reduce communication footprint given the knowledge of PCIe bandwidth capacity between each pair of CPU and GPU. Nonetheless, this knowledge of the underlying architecture comes at the cost of tuning and building a performance model at runtime or offline. We note that such a model can not be applied to irregular applications since their workload is unknown until execution.

In a similar way, DDA strategy attained similar performance results compared to HEFT

for DGEMM, Cholesky and LU. Its main advantage is theoretical guarantee in worst-case scenario in which task predictions differ significantly (Kedad-Sidhoum et al., 2013). Nevertheless, it showed a higher footprint than HEFT for nearly all cases. These results suggest that DDA would be improved if it considers data affinity in its prediction besides data transfers for task execution.

On the other hand, the designed locality-aware work stealing (H2) with *SetArch* annotation for GPUs showed significant performance despite its higher communication footprint for DGEMM, QR and LU. This strategy without any annotation only reported competitive results for experiments in which CPUs dominate execution, *i.e.*, using 11 CPUs. These results led us to infer two conclusions. First, a dynamic scheduling strategy can exploit efficiently heterogeneous architectures if the strategy makes use of data locality or uses a scheduling hint such as *SetArch*. In our experiments, there was no need to tune our executions but annotate certain tasks highly efficient on GPUs, which attain a speedup greater than one. Second, without using any annotations, work stealing is efficient on multi-CPU and it may guarantee performance gains only when CPU predominates performance. For instance, QR results with work stealing without annotations were near other strategies.

In some cases, our experimental results with work stealing *SetArch* outperformed cost models and also had a higher communication footprint. This phenomena occurred for most of points in Figure 6.4 (DGEMM), Figure 6.6 (LU), and Figure 6.7 (QR). Exception to this was Cholesky results in Figure 6.5 whose footprint was lower than obtained for cost models. These findings imply that the XKaapi runtime efficiently overlapped GPU transfer with kernel execution (Lima et al., 2012). Despite this advantage, it also stress one of the strategy limitations. The limitation of our locality-aware heuristic relies on the lack of knowledge about the application algorithm. In this sense, the strategy without any annotation is not aware of the efficiency in a certain architecture type. Since CPUs and GPUs are heterogeneous in computing power, a bad decision will affect tasks outside the critical path, which are candidates to be offload to accelerators, and may impact performance.

6.5 Summary

Scheduling is one of the essential building blocks for high performance on parallel systems, especially heterogeneous architectures. The heterogeneous nature of such architectures involves assumptions that include processing power, data transfer rate, bus contention, work load balancing, etc. Previous works studied strategies such as static distribution (Dongarra et al., 2012; Horton et al., 2011; Song and Dongarra, 2012; Tomov et al., 2010), data locality (Bueno et al., 2012), and cost models (Agullo et al., 2010, 2011a,b; Augonnet et al., 2010b, 2011) based on Earliest-Finish-Time scheduling (Topcuoglu et al., 2002).

In this Chapter, we presented a comparison of different scheduling strategies based on cost models and dynamic scheduling by work stealing for heterogeneous multi-CPU and multi-GPU architectures. We designed and evaluated three scheduling strategies on top of XKaapi runtime: locality-aware work stealing, Heterogeneous Earliest-Finish-Time (HEFT) and distributed Dual Approximation (DDA). We conducted experimental results with four tile algorithms from PLASMA on a heterogeneous architecture composed of 8 GPUs and 12 CPUs.

Our designed strategy HEFT outperformed the work stealing strategies in almost all cases. Clearly, the main reason for its efficiency is the knowledge of both the application and the underlying architecture to predict performance. Therefore, it reduced communication footprint and processor's idle time by selecting the "best" processor at runtime. The cost models such as HEFT and DDA have the disadvantages of requiring performance tuning for prediction on a specific architecture, and they are not applicable to irregular applications.

Our work stealing strategy with annotations, in this case the use of *SetArch* for GPU specific tasks, showed better results than cost models for some cases. The four tile algorithms for PLASMA attained significant results with this strategy, mainly `DGEMM`, Cholesky and QR. Our work stealing strategy had a higher communication footprint, but it is possible to obtain performance results with the overlap of GPU transfer and kernel execution. These findings led us to believe that work stealing can be suitable for heterogeneous architectures. Its does not require tuning for specific architecture, and is applicable to irregular problems. However, it is complex to attain the peak performance without any knowledge of architecture details on a heterogeneous architectures. Except for the QR experiments, the other problems did not attain performance as good as cost models.

We conclude that scheduling strategies based on cost models can be applied considering a specific type of parallel algorithms. These algorithms have tasks whose workload are very regular, and it varies according to its input size. A task entry on current performances models, including the model presented, has as identifier the input size and assumes a certain workload associated. Thus, no other parameters is considered at execution time prediction. Although simple techniques to correct prediction errors were applied, such as time correction before task execution (see Section 6.2), a scheduling decision of cost models such as HEFT was not questioned. Cost models may be insensitive to imbalances from the underlying system or tasks. Besides, tuning of the designed algorithm is essential and restricted to the target architecture.

On the other hand, the use of work stealing may show efficient performance provided that scheduler hints are given along with an efficient runtime system. This strategy may be efficient on irregular algorithms, and reacts to imbalances dynamically at runtime based on the idle resources. We originally assumed that most of tasks with a GPU version provide a speedup greater than one over its CPU version. A dynamic strategy with data locality can overcome the cache-unfriendly nature of pure work stealing. Furthermore, the use of task annotation (*SetArch*) in order to hint the scheduler can deal with the processing power differences of resources. We acknowledge the limitations of this approach that include previous knowledge of task nature (compute-bound or memory-bound).

These two types of scheduling strategies, grouped by cost model and dynamic scheduling, can efficiently distribute workload on heterogeneous architectures. While cost models were conceived for this architectures, dynamic scheduling is efficient in conjunction with heuristics and scheduler hints, as suggested by our results.

Finally, on the context of runtime systems, we conclude that our scheduling framework can provide a basic support for different scheduling strategies. Our framework, conceived from work stealing, is also capable of expressing cost model strategies along with performance models. We were able to design two different scheduling strategies, cost models and dynamic scheduling, and to attain notable performance results. Besides, our framework to express scheduling strategies on top of XKaapi opens the design of other schedulers

without major runtime modifications.

Chapter 7

Runtime Support for Native Mode on Intel Xeon Phi Coprocessor

Contents

Part of the works in this Chapter were published in Lima et al. (2013).

With the introduction of the Intel Xeon Phi coprocessor, Intel proposed an evolution in the way to develop applications for accelerators. Several researchers have recently moved their focus on this architecture (Cramer et al., 2012; Eisenlohr et al., 2012; Heinecke et al., 2013; Labarta and Beltran, 2013; Newburn et al., 2013; Pennycook et al., 2013; Tomov et al., 2010), trying to position the Intel Xeon Phi as a good candidate for executing efficient high-performance parallel applications. For instance, the European DEEP project aims at studying the contribution of the Intel Xeon Phi technology in the design of a novel architecture for *paving the way* to Exascale.

High performance on multicore architectures requires several threads of control running mostly independent code, with limited synchronizations to ensure a smooth progress of the computation. Programming directly with threads is considered as highly unproductive and error prone (Lee, 2006). Many parallel programming environments have been proposed to exploit such architectures, and two of them, Cilk and OpenMP, behave especially well for executing fine-grained parallelism (see Sections 3.1.1 on page 31 and 3.1.2 on page 32). Both Cilk and OpenMP have basic constructs to create independent tasks and to parallelize independent loops. Intel provides a rich set of parallel programming environments like Pthreads (of Electrical and Electronic Engineers, 1995), OpenMP, Intel Cilk Plus and TBB running on the Intel Xeon Phi, letting the programmer to choose the one that best suits his needs. The availability of all these environments clearly positions the Intel Xeon Phi coprocessor as a reliable target for accelerated applications.

Even if these programming environments improves opacity, they may not be suited for large-scale shared-memory architectures like a 240-threads Intel Xeon Phi coprocessor. In particular, several studies (Duran et al., 2009b; Kurzak et al., 2010) show that the strong synchronizations imposed by both OpenMP and Cilk execution models artificially limit the available parallelism: in these programming models, the synchronization construct blocks the running thread until previously spawned tasks have completed their execution. These studies emphasize data-flow approaches that are able to expose fine-grain one-to-one synchronizations between tasks: the runtime system can detect concurrent tasks as soon as their inputs are produced.

Data-flow programming model is a promising approach to take into account data transfers between disjoint memory address spaces. It was successfully validated on multi-CPU / multi-GPU architectures, as described in Chapter 5 on page 59 and Chapter 6 on page 79, and on large-scale distributed platforms (Bueno et al., 2012; Gautier et al., 2007). However, few studies have reported on data-flow task programming on the Intel Xeon Phi.

In this Chapter, we present performance evaluations of the XKaapi data-flow runtime on native Intel Xeon Phi applications: our goal is to study the strengths and the weaknesses of XKaapi to program native applications. We report a novel experimental evaluation on porting a high-performance data-flow programming environment into the Intel Xeon Phi coprocessor.

The Intel Xeon Phi can be seen as a set of hyperthreaded cores that share a global memory, which is not far from a multicore architecture. Porting XKaapi source code to the Xeon Phi was not difficult, requiring to specialize memory barriers and atomic operations to take into account the Xeon Phi specificities. Besides, we designed two main runtime modifications to optimize XKaapi:

- A new thread placement over physical cores and hardware threads (Section 7.1);
- A modified version of work stealing where one thread is allowed to steal (Section 7.2) in a physical core at the same time.

Most of this Chapter is dedicated to experimental results of the XKaapi runtime on the Intel Xeon Phi using three sets of benchmarks (Section 7.3).

7.1 Thread Placement

The Intel MIC architecture has a hardware topology similar to a SMP processor with four hardware threads (HT) per physical core, which is not far from a multicore architecture. As detailed in Section 2.1.4 on page 15, the MIC system seems a general-purpose SMP at user mode level.

We designed a new thread placement in order to evenly attribute threads to the Intel Xeon Phi physical cores. XKaapi default placement fills all processing units sequentially, *i.e.*, in an Intel Xeon Phi it fills each physical core with threads in all four hardware threads to step on the next physical core. In this placement strategy an execution with one thread per physical core is not possible. For instance, considering an Intel Xeon Phi card with 60 physical cores, 60 threads would fill only 30 physical cores. In our new strategy , threads

are placed equally among the available cores. A 60 threads placement will result in one thread per physical core.

Figure 7.1 illustrates a comparison between the default XKaapi placement over the new strategy for the Intel Xeon Phi coprocessor. In our example, we note that there are 2 physical cores and 8 hardware threads to execute 5 threads. XKaapi default placement (7.1a) distributes 4 threads on the first physical core and 1 thread on the second. Our new placement (7.1b) puts 3 threads on the first physical core and 2 threads on the second.

(a) Default placement. (b) New placement.

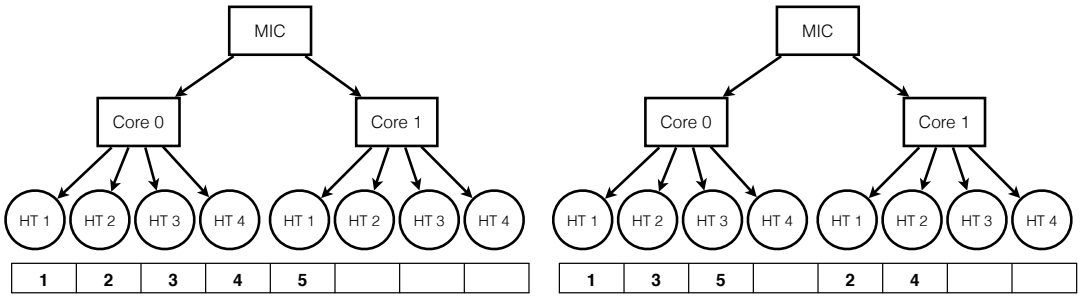


Figure 7.1 – XKaapi thread placement of five threads on the Intel Xeon Phi.

7.2 Work Stealing Scheduler

XKaapi work stealing scheduler have demonstrated the good scalability of XKaapi even at fine grain (Broquedis et al., 2012; Gautier et al., 2013b). The Intel Xeon Phi version has the same features detailed in Section 4.3 on page 51. Nevertheless, XKaapi does not have optimizations for physical cores with hardware threads. Each core has support to 4 hardware threads that may execute two instructions per clock cycle (Jeffers and Reinders, 2013, p. 249). Its multithreading may be useful for HPC workloads although the optimal number of threads may range from 2 to 4 threads per physical core.

XKaapi work stealing algorithm was modified on the Intel Xeon Phi version in order to avoid bottlenecks on the physical core. Algorithm 8 on the following page shows the steps performed before and after a thread executes a steal request. A XKaapi thread is allowed to enter in steal state only when no other thread in the same physical core is attempting to steal. Line 5 does the steal request and was unmodified for this version. If a thread is already in steal state on the same physical core, the current thread yields the CPU (hardware thread) by `pause` instruction. The operations to test and enter in stealing state (lines 1 and 4) were implemented by a Compare-and-Swap (CAS) operation.

Algorithm 8: XKaapi algorithm to steal tasks on the Intel Xeon Phi.

```
1 while There is a thread in Stealing State on this physical core do
2 |   Yield the hardware thread
3 end
4 Enter in Stealing State on this physical core
5 Try to steal a ready task from a random worker
6 Leave Stealing State from this physical core
```

7.3 Experiments

This Section presents experimental results of the XKaapi runtime system on a single Intel Xeon Phi coprocessor in native execution. Our objectives are:

1. Compare XKaapi over OpenMP and Intel Cilk Plus for multicore processors (Intel Xeon Sandy Bridge) and manycore coprocessors (Intel Xeon Phi);

2. Evaluate XKaapi on an Intel Xeon Phi using task parallel benchmarks such as PLASMA and BOTS.

First we describe our target platform (Section 7.3.1) composed of four processors and one coprocessor. Next we report performance results of an Intel Xeon Phi coprocessor compared to an Intel Xeon Sandy Bridge processor (Section 7.3.2). Then we show our preliminary experiments with PLASMA (Section 7.3.3) and BOTS (Section 7.3.4).

All times reported in this Section are average of more than 30 executions with a warm-up phase of 2 runs.

7.3.1 Platform and Environment

All the applications were executed natively on the Intel Xeon Phi environment. The Intel Xeon Phi used is a 5110P with 60 cores running at 1.053 Ghz and sharing 8 GB of memory. Each core has support to 4 hardware threads, for a total of 240 threads.

The host platform, hereafter called *Intel Xeon Sandy Bridge*, contains 4 Intel Xeon E5-4620 multicore processors for a total of 32 cores running at 2.20 GHz and sharing 384 GB of main memory. On this machine, hyperthreading was activated and enabled 2 hardware threads per core resulting in a total of 64 hardware threads.

The software environment used on Intel Xeon Sandy Bridge was the following: the operating system was a Debian distribution with a 3.8.11 Linux kernel; the OpenMP and Intel Cilk Plus applications were compiled with the Intel C/C++ compiler 13.1.3 and executed using the corresponding runtime system from Intel. Intel Xeon Phi's firmware version was 1.14.4616 and comes with version 13.0.1 of the Intel C/C++ compiler, MPSS 2.1.6720-13 and compiler_xe_2013.1.117. The benchmarks were compiled with Intel `icpc` and the `-O3` option. On Intel Xeon Sandy Bridge, threads were explicitly bound to physical cores for OpenMP and XKaapi.

We evaluated XKaapi version 2.0 with the modifications described in this Chapter. XKaapi applications were compiled with the same Intel compilers used to compile OpenMP and Intel Cilk Plus applications.

7.3.2 Comparison Xeon vs Xeon Phi

In this Section we report experimental evaluations of XKaapi compared to Intel OpenMP and Intel Cilk Plus on an Intel Xeon Sandy Bridge machine and an Intel Xeon Phi co-processor. We designed three parallel benchmarks in order to evaluate performance and runtime overhead:

- Fibonacci computation that allows to study overhead and scalability;
- NQueens search to generate irregular and dynamic tasks;
- Cholesky factorization based on tile blocked algorithm from PLASMA in order to measure raw performance on each architecture.

Fibonacci

The Fibonacci benchmark computes the n-th Fibonacci number using a naive recursive computation. The purpose of this benchmark is to compare the overheads and scalability of the runtime systems that come with the OpenMP, the Intel Cilk Plus and the XKaapi programming environments on both the Intel Xeon Phi coprocessor and the Intel Xeon Sandy Bridge machine.

The code executed by each of the three environments generates the same number of tasks: each recursive call creates two child tasks (with `#pragma omp task`, `cilk_spawn` or `ka::Spawn`) to compute the $n-1$ and $n-2$ Fibonacci numbers in parallel, and then synchronizes the created tasks (with `#pragma omp taskwait`, `cilk_sync` or `ka::Sync`) before returning the sum of the two subresults. The recursion stops when the computation of the Fibonacci number is less than 2 (see Section 3.1 on page 31 for code examples in OpenMP and Cilk).

Figure 7.2 on the following page reports experimental results on Intel Xeon Sandy Bridge (7.2a) and Intel Xeon Phi (7.2b) respectively. The obtained results show that XKaapi had the lowest overhead among the three tested environments. XKaapi was 1.83 times faster than Intel Cilk Plus with 64 threads on Intel Xeon Sandy Bridge and 2.01 times faster than Intel Cilk Plus with 240 threads on Intel Xeon Phi. XKaapi speedup was 3.28 with 64 threads on Intel Xeon Sandy Bridge and 21.78 with 160 threads on the Intel Xeon Phi. Intel Cilk Plus had speedup over the sequential version for almost all cases except on Intel Xeon Sandy Bridge below 24 threads. Its speedup was 1.79 with 64 threads on Intel Xeon Sandy Bridge and about 10.11 with 240 threads on Intel Xeon Phi.

On the contrary, Intel OpenMP exhibited poor performance for this benchmark with fine grain recursive tasks. The speedup results on Intel Xeon Sandy Bridge were below 1 for all thread configurations. The maximum speedup obtained on Intel Xeon Phi was 1.18.

On this benchmark, the Intel Xeon Phi was about 13 times slower than Intel Xeon Sandy Bridge on the sequential execution (T_s), but on 1-core execution (T_1) 6.4 times slower with Intel Cilk Plus and 5.4 slower with XKaapi. If we look at the maximum performance on both architectures, the Intel Xeon Phi was about 2.1 times slower than Intel Xeon Sandy Bridge on the tested benchmark.

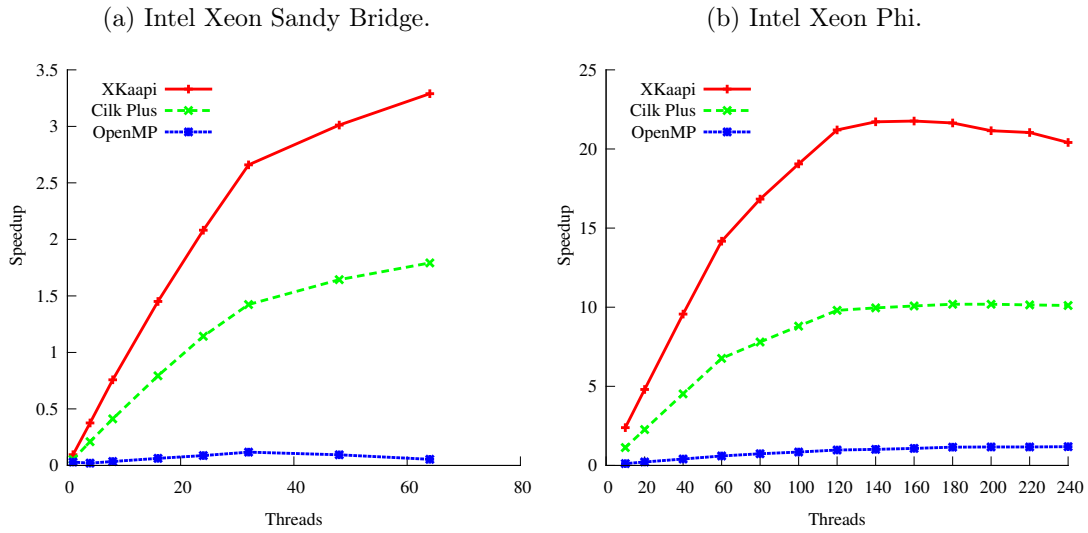


Figure 7.2 – Fibonacci speedup over sequential execution for XKaapi, OpenMP, and Intel Cilk Plus (input of $N = 38$).

NQueens

The NQueens benchmark is based on the Takaken (Takaken) optimized sequential code to compute the number of solutions for the NQueens problem. It has been parallelized using XKaapi since 2007 (Gautier et al., 2007) and we adapted it to OpenMP and Intel Cilk Plus. We have decided not to consider the OpenMP BOTS NQueens program as baseline as it runs slower than Takaken's code, mainly because it does not take symmetries of the configuration into account. Sequential execution of our code is about 1200 times faster than BOTS NQueens for $N = 16$ using the same `icpc` compiler with the `-O3` option.

The principle of the parallelization is a recursive exploration of the different configurations of the chessboard: a set of possible configurations is generated at each recursive call, taking symmetries into account. Each configuration is explored by an independent task. On final recursion, possible solutions are accumulated in a global variable. The parallelism is generated until a threshold, then the code performs sequential exploration.

The OpenMP, Intel Cilk Plus and XKaapi codes generate the same independent tasks. The main difference between the three environments resides in the way solutions are accumulated. As the original code relies on a 3D vector of solutions holding each of the 3 considered symmetries, the OpenMP version uses a critical region to accumulate the solutions. The Intel Cilk Plus version behaves similarly, using a mutex to implement the same kind of critical region. So, for each accumulation, these runtime systems perform an *a priori* synchronization before accessing the global variable.

Nevertheless, the XKaapi version creates tasks with access to the global variable declared as "cumulative write access" (Galilee et al., 1998; Gautier et al., 2007), which allows to accumulate arbitrary data with an user-defined associative operator. When a thief thread steals a task, the runtime creates a new *per thief thread* data that the stolen task and its descendants use for the accumulation. When the stolen task completes, the new

data is accumulated to the victim thread's data. At the end, the global variable contains the final accumulated result. This mechanism enables the XKaapi runtime to reduce the required synchronizations compared to OpenMP and Intel Cilk Plus.

Our first experiments with NQueens aim to evaluate the impact of the grain size on the parallel algorithm. Figure 7.3 illustrates execution times for different grain sizes on Intel Xeon Sandy Bridge and Intel Xeon Phi. First we note that setting the NQueens threshold to bigger values will generate more parallelism, creating more fine-grain tasks. Both Intel Cilk Plus and OpenMP results can be explained by the overhead when executing fine-grain tasks (greatest threshold in the Figure). On the other hand, XKaapi seems to be able to efficiently execute applications at finer task grains while limiting the negative impact of runtime-related overheads on the overall execution time.

(a) Intel Xeon Sandy Bridge. (b) Intel Xeon Phi.

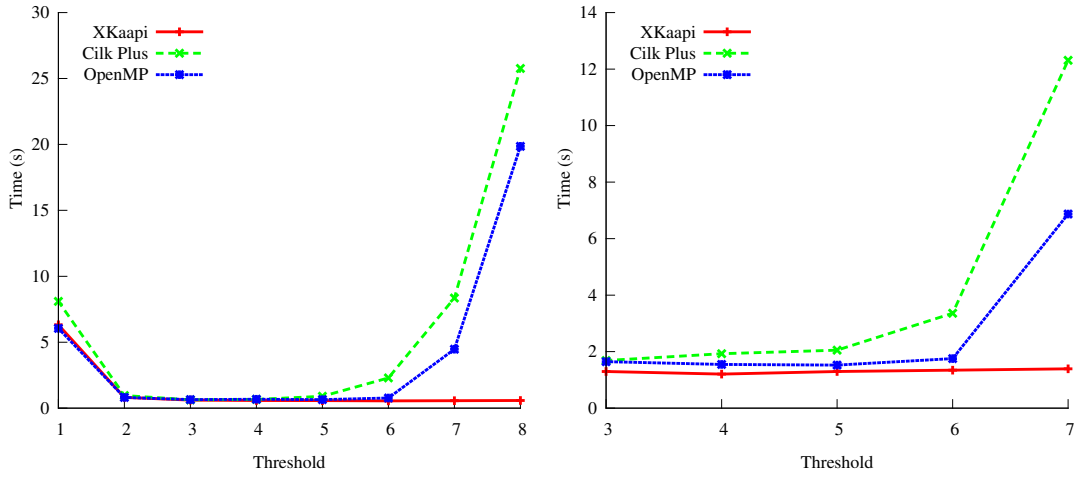


Figure 7.3 – Time *versus* threshold for NQueens benchmark (N=17) for XKaapi, Intel Cilk Plus and OpenMP.

On 64 hardware threads (32 cores) of Intel Xeon Sandy Bridge, the minimum time for XKaapi was obtained with a threshold of $t = 6$, which is different from the one used by Intel Cilk Plus ($t = 3$) and OpenMP ($t = 5$). On the 240 threads of Intel Xeon Phi, we set the threshold to $t = 6$ for all the environments.

Figure 7.4 on the following page reports the speedup $S = T_s/T_P$ for NQueens ($N = 17$) on Intel Xeon Sandy Bridge and Intel Xeon Phi. For each environment, we report the performance obtained using the best threshold. The speedup for all three (OpenMP, Intel Cilk Plus, and XKaapi) environments were similar up to 16 threads on Intel Xeon Sandy Bridge and up to 60 threads on the Intel Xeon Phi. For all other cases, XKaapi outpeformed OpenMP and Intel Cilk Plus. On Intel Xeon Sandy Bridge, XKaapi reached a speedup of 30.21 on 32 threads and 39.52 on 64 threads while OpenMP reached 29.39 and 33.43 of speedup respectively. On Intel Xeon Phi XKaapi reached a speedup of 58.76 on 60 threads and 85.51 on 240 threads while OpenMP reached 55.10 and 65.51 of speedup respectively. XKaapi had the maximum speedup on both architectures: 39.52 on Intel Xeon Sandy Bridge (64 threads) and 90.18 on Intel Xeon Phi (120 threads). XKaapi was 1.18 times

faster than OpenMP on 64 threads on Intel Xeon Sandy Bridge and 1.3 times faster than OpenMP on 240 threads on Intel Xeon Phi.

(a) Intel Xeon Sandy Bridge. (b) Intel Xeon Phi.

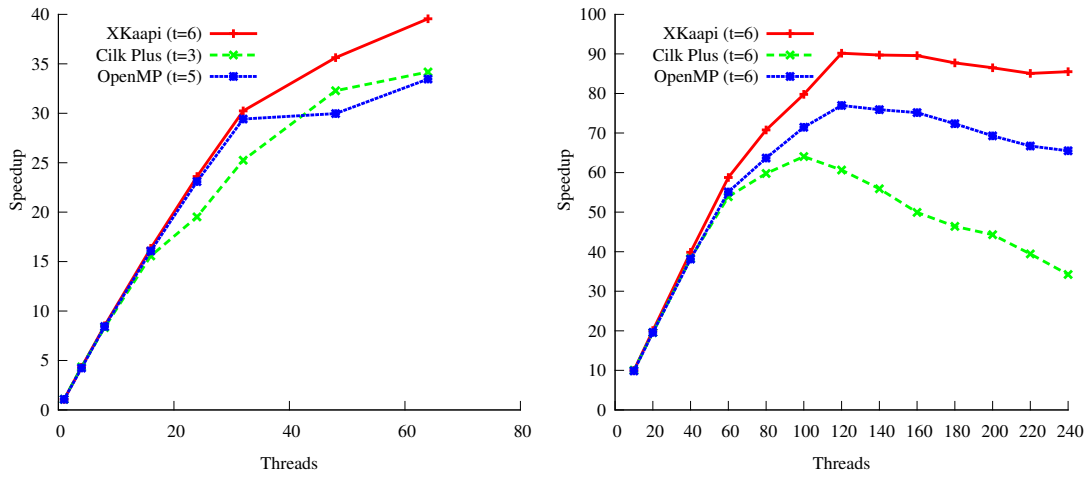


Figure 7.4 – Scalability of the NQueens benchmark (N=17) for XKaapi, Intel Cilk Plus and OpenMP.

For NQueens ($N = 17$), the best execution time on Intel Xeon Phi obtained by XKaapi was 1.27 seconds on 120 threads. On Intel Xeon Sandy Bridge, the best execution time on 16 threads over 8 physical cores was 1.32 seconds. For this benchmark, we can confirm that the Intel Xeon Phi coprocessor may be able to reach better performance than a single Intel Xeon Sandy Bridge socket.

Cholesky

The Cholesky factorization (POTRF) decomposes an $n \times n$ real symmetric positive definite matrix A into the form $A = LL^T$ where L is an $n \times n$ real lower triangular matrix with positive diagonal elements (Agullo et al., 2010).

Figure 7.5 on the next page shows the pseudo-code of both the XKaapi and the Intel Cilk Plus versions. The main difference between the two versions is the absence of synchronization in the XKaapi code thanks to data-flow dependencies between tasks. The user is responsible for indicating the mode each task uses to access memory and the runtime system build dependencies between tasks that access to same memory region. The XKaapi code illustrates the C++ interface that does not require a specific compiler: in the notation `A(ri,rj)`, `ri` and `rj` denote a range of indexes so that `A(ri,rj)` is a sub-matrix of matrix `A`. Thanks to this finer knowledge of tasks dependencies, the runtime system can schedule ready tasks between two main iterations in k (Duran et al., 2009b; Kurzak et al., 2010). The OpenMP version is similar to the Intel Cilk Plus version by replacing `cilk_spawn` by `#pragma omp task` and `cilk_sync` by `#pragma omp taskwait`.

Our experiments use the same parallel version of the Cholesky factorization, as found in PLASMA (Buttari et al., 2009). The algorithm has been re-implemented in two versions: block version (same as in PLASMA) and parallel-diagonal version (Gautier et al., 2013b).

```
1 /* XKaapi */
2 for( k=0; k < NB; k++ ) {
3   ka::Spawn<POTRF>()( A(k,k) );
4   for( m=k+1; m < NB; m++ )
5     ka::Spawn<TRSM>()( A(k,k),
6       A(m,k) );
7
8   for( m=k+1; m < NB; m++ ) {
9     ka::Spawn<SYRK>()( A(m,k),
10      A(m,m) );
11    for( n=k+1; n < m; n++ )
12      ka::Spawn<GEMM>()( A(m,k),
13        A(n,k), A(m,n) );
14
15  }
16 }
```

```
1 /* \cilk */
2 for( k=0; k < NB; k++ ) {
3   POTRF( A(k,k) );
4   for( m=k+1; m < NB; m++ )
5     cilk_spawn TRSM( A(k,k),
6       A(m,k) );
7   cilk_sync;
8   for( m=k+1; m < NB; m++ ) {
9     cilk_spawn SYRK( A(m,k),
10      A(m,m) );
11    for( n=k+1; n < m; n++ )
12      cilk_spawn GEMM( A(m,k),
13        A(n,k), A(m,n) );
14  cilk_sync;
15  }
16 }
```

Figure 7.5 – Example of a left-looking Cholesky factorization with XKaapi and Intel Cilk Plus.

The parallel-diagonal Cholesky is a two level parallel algorithm: at the upper level, we use the PLASMA algorithm; at the lower level the panel factorization task (POTRF) is parallelized using the same parallel algorithm as at upper level by decomposing one tile in sub-tiles of size 32×32. We have not used auto-tuning to select the sizes of the tile and sub-tile, but an empirical approach: after a few experiments showing their average good performances, we have decided to use theses values. The two algorithms on XKaapi are similar to the multi-GPU version depicted in Appendix A on page 139.

Figure 7.6 on the next page reports the GFlop/s rate obtained for a matrix of size 8192×8192 with tiles of size 256×256. The overall best performance was obtained by XKaapi for both architectures. On Intel Xeon Sandy Bridge, OpenMP and Intel Cilk Plus had a similar level of performance. On Intel Xeon Phi, the behavior is the same than on Intel Xeon Sandy Bridge, except the XKaapi parallel-diagonal version showed an important performance improvement. Results on bigger matrices follow the same behavior (Figure 7.7 on the following page) on the two architectures.

On the Intel Xeon Phi architecture, the cores are efficient on regular matrix operations, which is the case for tasks TRSM, GEMM and SYRK of Figure 7.5, but not for the POTRF task. On the Cholesky factorization, the POTRF tasks on diagonal block `A(ri,ri)` are on the critical path of the execution. Any reduction in the completion of POTRF tasks allows other cores to resume their execution, thus reducing idle time. The same phenomenum was observed on multi-CPU/multi-GPU factorization (Agullo et al., 2010; Gautier et al., 2013b; Kurzak et al., 2010) where POTRF tasks are inefficient on GPU, thus to decrease execution time, tasks belonging to the critical path have to be parallelized. On multi-CPU/multi-GPU this was done by executing PORTF tasks on CPUs. On Intel Xeon Phi, the same performance improvement can be obtained by parallelizing POTRF tasks as performed by the XKaapi parallel-diagonal version described in our previous multi-GPU

(a) Intel Xeon Sandy Bridge. (b) Intel Xeon Phi.

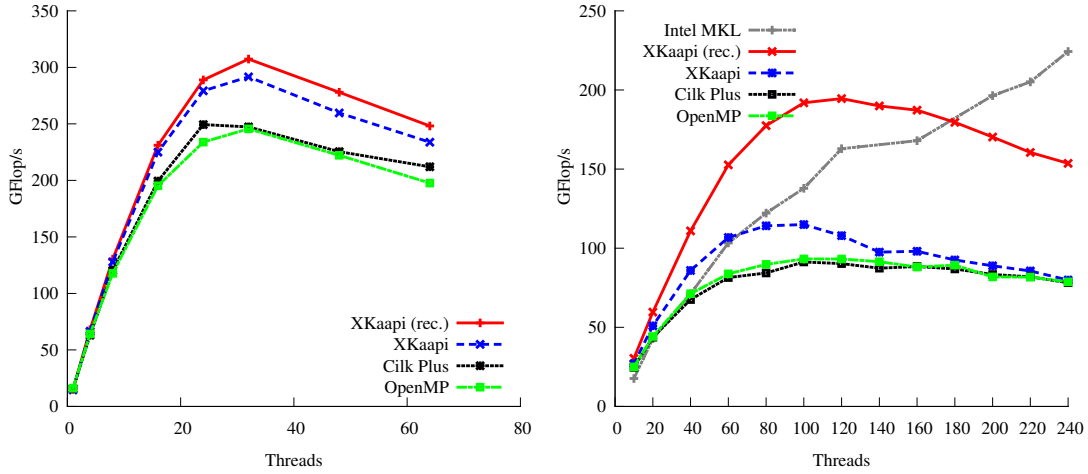


Figure 7.6 – Results of Cholesky benchmarks for XKaapi, OpenMP, and Intel Cilk Plus for matrix size 8192×8192 and tile size 256×256. MKL was only measured on the Intel Xeon Phi and omitted on the Intel Xeon Sandy Bridge.

(a) Intel Xeon Sandy Bridge. (b) Intel Xeon Phi.

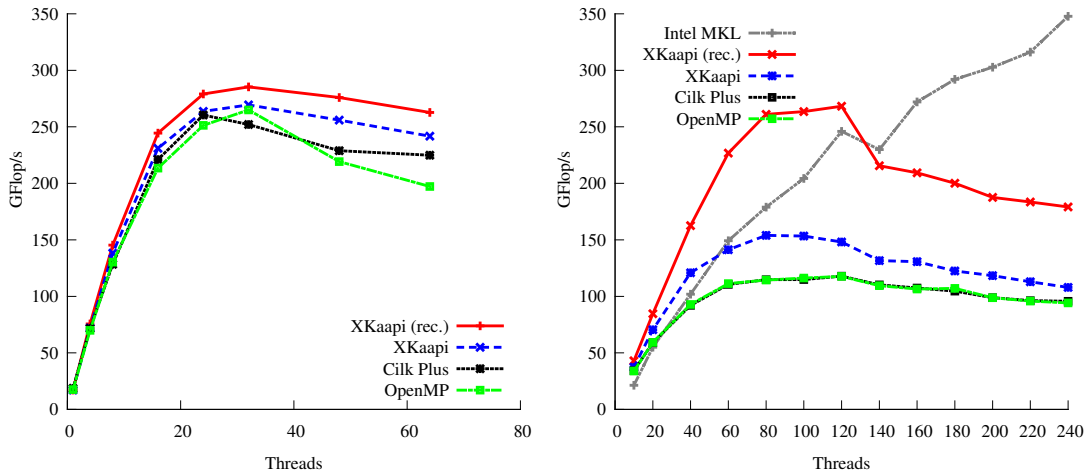


Figure 7.7 – Results of Cholesky benchmarks for XKaapi, OpenMP, and Intel Cilk Plus for a matrix of size 16384×16384 and tile size 512×512. MKL was only measured on the Intel Xeon Phi and omitted on the Intel Xeon Sandy Bridge.

implementation (Gautier et al., 2013b).

Figures 7.6b and 7.7b show an interesting behavior on the Intel Xeon Phi. If we compare MKL performance with the two XKaapi codes (parallel-diagonal and block versions), XKaapi versions were above the performance of MKL in most cases. The parallel-diagonal

version was up to 47% faster than MKL (for 60 threads) and faster than MKL up to 180 threads (matrix 8192×8192). Moreover, for a matrix size of 8192×8192, the XKaapi version reached 152.67 GFlop/s with 60 threads, 191.9 GFlop/s with 100 threads and only 194.57 GFlop/s with 120 threads. The MKL reached 103.35 GFlop/s with 60 threads, 137.8624 with 100 threads and 224.28 GFlop/s with all the 240 threads.

Without any description of how the MKL POTRF routine is implemented, the XKaapi version was able to provide a higher ratio GFLops/thread than MKL up to 180 threads. These findings led us to believe that our initial port on Intel Xeon Phi may not take care of affinity between tasks and data, and our randomized work stealing may have an important memory footprint. Techniques we have developed for multi-GPUs (Gautier et al., 2013b) would be tested to improve the scalability of our runtime system.

The XKaapi parallel-diagonal Cholesky factorization obtained at most 81.15 GFlop/s on one Intel Xeon Sandy Bridge socket for a matrix size of 8192×8192. The same code on Intel Xeon Phi performed at 194.57 GFlop/s, and the MKL was at most 224.28 GFlop/s. Therefore, one Intel Xeon Phi was 2.4× more powerful than one Intel Xeon Sandy Bridge socket on this benchmark.

Runtime Overhead

Table **??** shows the runtime overhead measured from Intel OpenMP, Intel Cilk Plus, and XKaapi on Intel Xeon Sandy Bridge processor and Intel Xeon Phi coprocessor. Our overhead metric is T_1/T_s where T_1 is the execution time of the parallel algorithm with 1 thread and T_s the sequential algorithm. These measures suggest that XKaapi had a lower overhead for both architectures on the Fibonacci benchmarks. Intel OpenMP had the highest overhead of almost 33× on Intel Xeon Sandy Bridge and 17× on Intel Xeon Phi over the sequential time. On the other hand, all runtime systems had similar results for the NQueens search: 0.93× on Intel Xeon Sandy Bridge and 0.98× on Intel Xeon Phi.

Table 7.1 – Runtime overhead for Fibonacci and NQueens on Intel Xeon Sandy Bridge and Intel Xeon Phi.

	Intel Xeon Sandy Bridge				Intel Xeon Phi			
		T_1/T_s				T_1/T_s		
	T_s	OpenMP	Cilk	XKaapi	T_s	OpenMP	Cilk	XKaapi
Fibonacci	0.27	33.48	19.13	10.59	3.77	17.39	8.80	4.11
NQueens	21.67	0.93	0.92	0.93	114.95	0.98	0.98	0.98

7.3.3 PLASMA: Cholesky, LU, and QR

Our experiments use the tiled algorithms of PLASMA[1] version 2.5.2 for Cholesky, LU, and QR. The purpose of this benchmark is to compare performance results of static strategy

[1] `http://icl.cs.utk.edu/plasma/`

and dynamic load balancing (XKaapi and QUARK). We performed experiments over four parallel versions of each benchmark:

1. MKL parallel version;
2. PLASMA static scheduling;
3. Dynamic scheduling with XKaapi;
4. Dynamic scheduling with QUARK from PLASMA.

We implemented the QUARK API (YarKhan et al., 2011) in order to execute PLASMA algorithms over the XKaapi runtime on Intel Xeon Phi.

We used MKL kernels for BLAS and LAPACK to implement algorithm tasks. We report all results from PLASMA in floating-point operations per seconds (GFlop/s). Table **??** lists PLASMA methods and parameters for our experiments. We note that the peak performance obtained on the Intel Xeon Phi of our experiments was 603.63 GFlop/s with MKL matrix multiplication in double precision (`DGEMM`).

Table 7.2 – PLASMA method and parameters for experiments on the Intel Xeon Phi. The parameters of PLASMA are matrix size, block size (NB), and internal block size (IB).

Method	LAPACK method	PLASMA method	Matrix size	NB	IB
Cholesky	`DPOTRF`	`PLASMA_dpotrf_Tile`	8192	128	32
LU	`DGETRF`	`PLASMA_dgetrf_incpiv_Tile`	8192	128	32
QR	`DGEQRF`	`PLASMA_dgeqrf_Tile`	8192	128	32

Figure 7.8 on the facing page shows performance results of Cholesky. In most cases MKL outperformed other strategies. The peak performance with MKL was 215.60 GFlop/s, which is 35.71% of the coprocessor peak. XKaapi attained performance results near PLASMA static scheduling, whose peak was 109.45 GFlop/s at 120 threads. PLASMA QUARK exhibited poor performance beyond 40 threads.

Figure 7.9 on the next page illustrates performance results of LU. In most cases MKL outperformed other strategies and attained 159.73 GFlop/s of peak performance, which is 26.46% of the coprocessor peak. XKaapi version showed similar performance compared to PLASMA static version. PLASMA QUARK exhibited poor performance beyond 40 threads.

Figure 7.10 on page 112 shows performance results of QR. In a similar way MKL outperformed other strategies with peak of 149.86 GFlop/s, which is 24.82% of the coprocessor peak. XKaapi outperformed PLASMA static version from 80 threads and its peak was 83.31 GFlop/s. PLASMA QUARK outperformed PLASMA static scheduling and MKL with 60 threads (60.57 GFlop/s), but dropped using more than 80 threads.

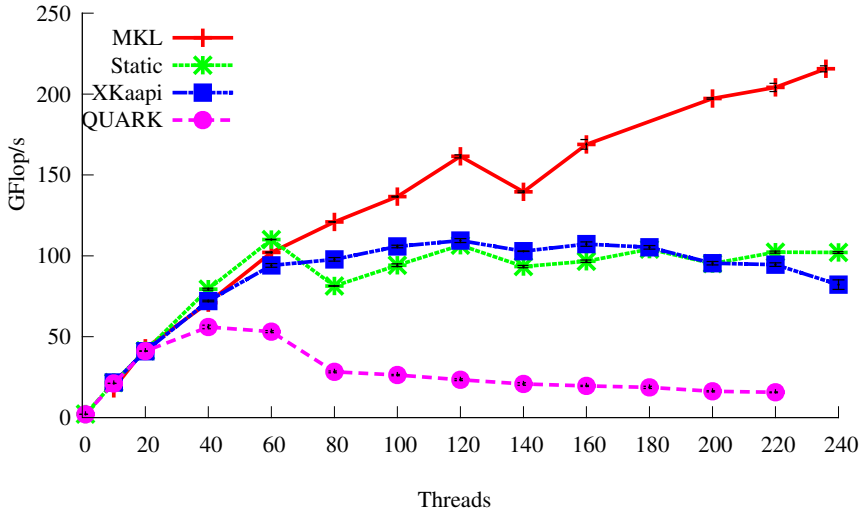


Figure 7.8 – Cholesky factorization results from PLASMA with matrix size 8192×8192 on Intel Xeon Phi.

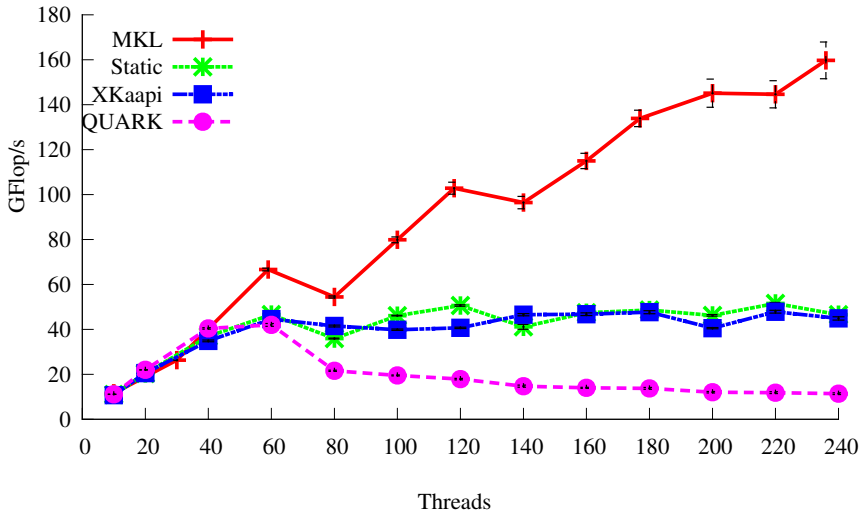


Figure 7.9 – LU factorization results from PLASMA with matrix size 8192×8192 on Intel Xeon Phi.

7.3.4 BOTS: FFT, Health, SparseLU, and Strassen

The purpose of our preliminary experiments in this Section is to evaluate the XKaapi runtime with task parallel benchmarks and compare to native Intel OpenMP on an Intel Xeon Phi. The Barcelona OpenMP Tasks Suite[2] (BOTS) is a set of applications that allow to evaluate OpenMP implementations (Duran et al., 2009a). Most of its applications use the OpenMP task directives. In our experiments we selected four benchmarks from BOTS:

[2] `https://pm.bsc.es/projects/bots`

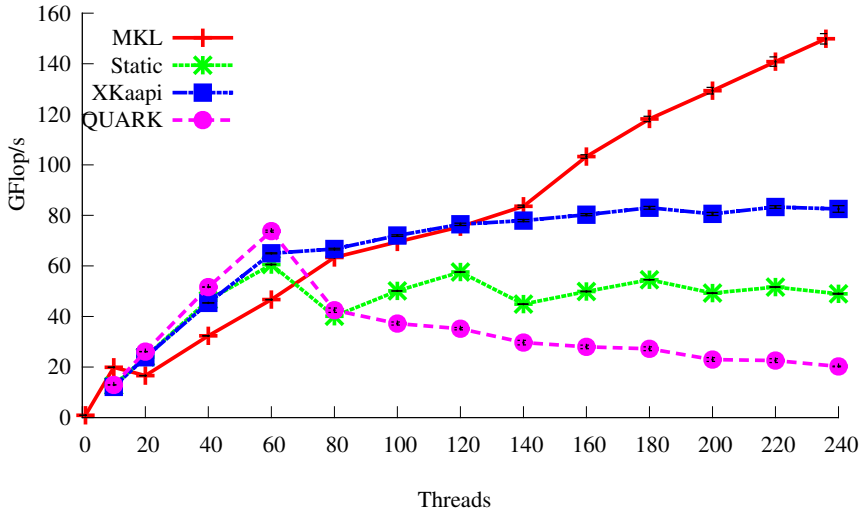


Figure 7.10 – QR factorization results from PLASMA with matrix size 8192 × 8192 on Intel Xeon Phi.

1. **FFT** computes the one-dimensional Fast Fourier Transform of a vector with n complex values. Its parallel version is based on Cilk's recursive implementation.
2. **Health** simulates the Columbian Health Care System.
3. **SparseLU** computes a LU matrix factorization over sparse matrices.
4. **Strassen** algorithm uses hierarchical decomposition of a matrix for multiplication of large dense matrices. Its parallel version is based on Cilk's recursive implementation.

We designed a BOTS version using XKaapi compiler annotations with data dependencies (Lementec et al., 2011b).

Table **??** gives the input parameters, T_1, and overhead T_1/T_s of BOTS on the Intel Xeon Phi. Both OpenMP and XKaapi had similar results for T_1 and overhead of about 1 for all benchmarks.

Table 7.3 – Input parameters and runtime overhead of BOTS on the Intel Xeon Phi.

			OpenMP		XKaapi	
Benchmark	Input	T_s	T_1	T_1/T_s	T_1	T_1/T_s
FFT	32M floats	158.82 s	160.59 s	1.01	160.79 s	1.01
Health	medium.input	61.14 s	72.57 s	1.18	71.73 s	1.17
SparseLU	$n = 25$, $m = 25$	1.03 s	1.03 s	0.99	1.04 s	1.00
Strassen	$n = 1024$, $y = 64$	5.94 s	5.68 s	0.95	6.03 s	1.01

Figure 7.11 on the facing page shows speedup T_s/T_P on Intel Xeon Phi coprocessor. XKaapi outperformed OpenMP with SparseLU (7.11c) in all cases and Health up to 120

threads (7.11b). Nonetheless, OpenMP reached better results on FFT (7.11a) and Strassen (7.11d). On FFT, OpenMP reached the maximum speedup of 85.24 with 220 threads, 2.92 times better than XKaapi best result (29.14 of speedup with 100 threads). For Health, OpenMP had the maximum speedup of 33.98 with 180 threads, only 1.03 times better than XKaapi best result (32.79 of speedup with 120 threads). SparseLU maximum speedup of 16.55 on 60 threads was obtained with XKaapi, 1.63 times better than OpenMP best result (10.15 of speedup with 20 threads). In Strassen OpenMP attained the best speedup of 11.87 with 40 threads, 1.33 times better than XKaapi best speedup (8.86 with 20 threads).

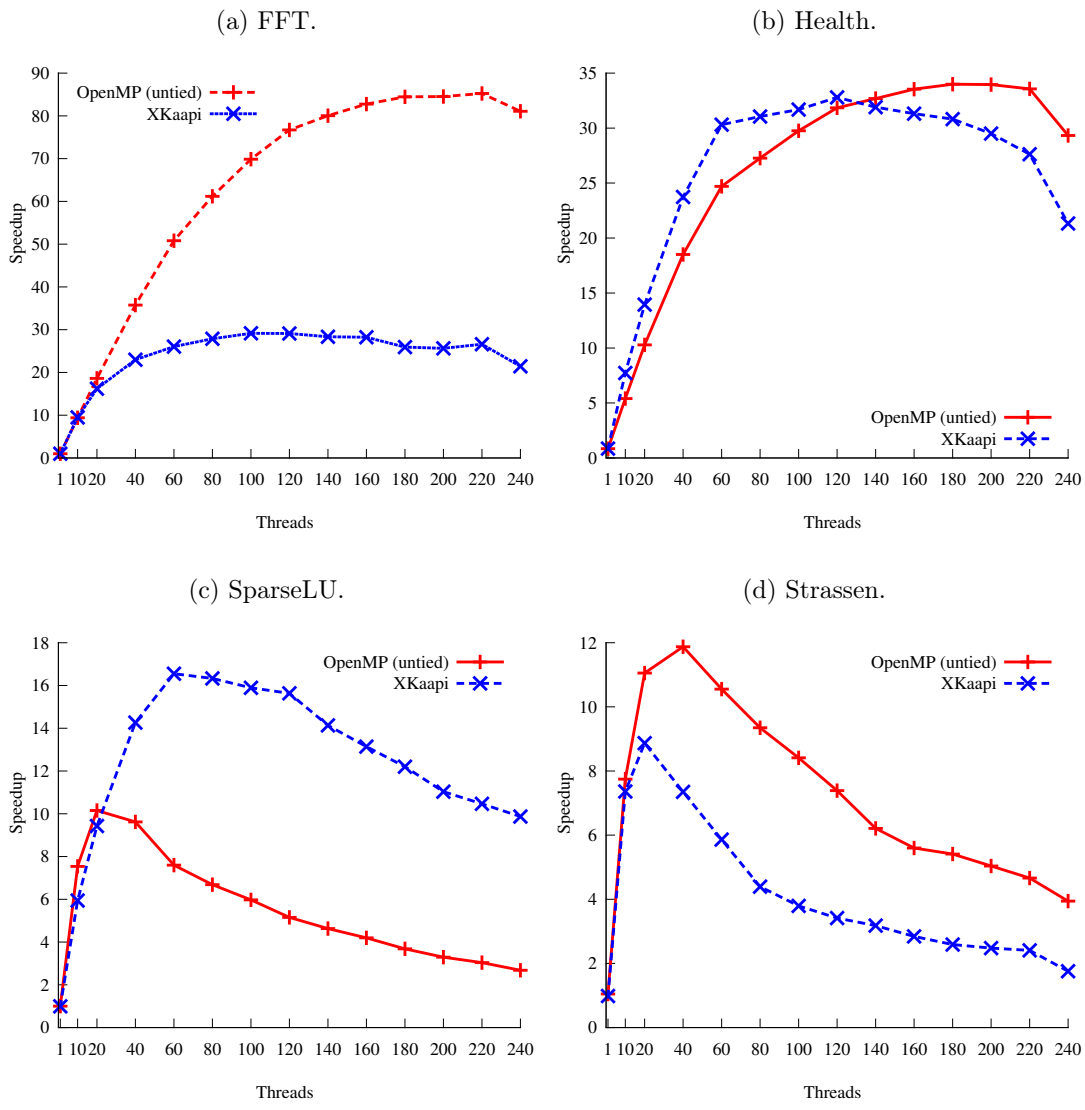


Figure 7.11 – Preliminary results of BOTS benchmarks FFT, Health, SparseLU, and Strassen on the Intel Xeon Phi.

7.4 Discussion

In our benchmarks, we evaluated the XKaapi runtime overhead and performance. It appears that XKaapi had the lowest overhead compared to Intel OpenMP and Intel Cilk Plus. XKaapi intrinsic overheads due to the computation of the data-flow dependencies between tasks are incurred by steal operations. If the number of steal operations is very small compared to the number of created tasks, as in Fibonacci (Frigo et al., 1998), data-flow related overheads do not impact XKaapi's performance obtained.

One reason for Intel OpenMP overhead could be the fact that the 1-core execution is optimized to avoid task creation, performing simple function calls as for the sequential code. The reference time T_1 does not include overheads that only appears when several cores are used. We already noticed that the overheads of GNU/GCC libGOMP runtime system (Broquedis et al., 2012) on fine grain task-based programs were large, and smaller for Intel's OpenMP implementation. On this task-based program, the Intel OpenMP runtime could be improved to achieve better performance, for instance by using the approach described by Broquedis et al. (2012).

Results of our benchmarks on an Intel Xeon Sandy Bridge machine and Intel Xeon Phi coprocessor showed that one Intel Xeon Phi chip with 60 cores can be a competitive architecture almost outperforming one socket of the complex superscalar and out-of-order 8 cores Intel Xeon E5-4620 processor, if, and only if, (a) the application exhibits enough parallelism, even irregular and dynamic, for the 240 available threads; (b) the runtime is able to schedule fine grain tasks with low overhead.

In addition, XKaapi outperformed other strategies on our performance experiments on Cholesky in C++. Although, its performance was below MKL after 120 threads. These results can be explained by the lack of autotunning and use of vector instructions. It seems that the grain size of our experiments did not attain best performance and may exhibit cache problems. We assume that MKL would have a fine tuned task size and would explore Intel Xeon Phi full potential of its VPUs.

Our PLASMA experiments attained results compared to static scheduling of PLASMA, but below MKL. It seems that PLASMA algorithms are not efficient on accelerators such as the Intel Xeon Phi. For instance, our parallel-diagonal strategy was able to improve performance significantly. Besides, the poor performance of QUARK dynamic scheduling suggests that QUARK is not able to scale and have contentions such as global locks and memory barriers.

Our preliminary experiments with BOTS benchmarks exposed the main weaknesses of the XKaapi runtime compared to Intel OpenMP. It seems that XKaapi exhibited high overhead on recursive tasks with workload such as FFT and Strassen, unlike Fibonacci benchmark from our previous experiments. Besides, Intel OpenMP may optimize recursive calls at compiler time.

7.5 Summary

In this Chapter, we presented performance results of the XKaapi data-flow programming model on the Intel Xeon Phi coprocessor in native execution. We evaluated three sets of benchmarks in order to mainly evaluate XKaapi runtime overhead and performance. In

our first set, we compared XKaapi to OpenMP and Intel Cilk Plus, native Intel Xeon Phi parallel programming environments provided by Intel. We conducted experiments with a 60-core Intel Xeon Phi and four Intel Xeon Sandy Bridge with 32 physical cores and 64 hardware threads. Our second set was composed linear algebra algorithms of PLASMA over dynamic scheduling with XKaapi and QUARK, and static scheduling from PLASMA. Finally, the third set consisted in evaluate XKaapi with four task parallel benchmarks from BOTS.

Our performance benchmarks on linear algebra applications showed that using finer synchronizations between tasks (data-flow dependencies) is more efficient than only relying on the fork-join model as OpenMP and Intel Cilk Plus. Fine grain parallelism may be increasingly essential as the number of cores grow faster than memory capabilities. Although the 60 cores of our Intel Xeon Phi shared only 8 GB of memory, the design of parallel applications under these constraints require finer tasks that may hopefully take advantage of finer data-flow dependencies for better performance.

This Chapter presented preliminary and promising performance results of XKaapi on the Intel Xeon Phi coprocessor. These results led us to infer that a data-flow task programming with efficient scheduling would be essential on accelerators. Although, on some cases, it seems that to use the full potential of the Intel Xeon Phi coprocessor would depend on the use of vector instructions (Jeffers and Reinders, 2013, p. 249).

Chapter 8

Conclusion

Contents

With the reported findings in this thesis we hope to contribute with the state of art of parallel programming for parallel systems, specially multicore architectures with accelerators. The main objective of this thesis is to study the issues of data-flow task programming on multi-CPU architectures enhanced with accelerators. We target those architectures with the XKaapi runtime system.

The structure of this thesis reflects our approach. We first studied the issues on multi-GPU architectures for asynchronous execution and scheduling. Work stealing with heuristics showed significant performance results, but did not consider the computing power of different resources. Next, we designed a scheduling framework and a performance model to support scheduling strategies over XKaapi runtime. Finally, we performed experimental evaluations over the Intel Xeon Phi coprocessor in native execution.

Our conclusion is twofold. First we concluded that data-flow task programming can be efficient on accelerators, which may be GPUs or Xeon Phi coprocessors. The main advantages of this programming model are the implicit synchronizations and abstraction of data transfers. Second, the runtime support of different scheduling strategies is essential. Cost models provide significant performance results over very regular computations, while work stealing can react to imbalances at runtime. Although its provably efficiency, work stealing performance depends on scheduling heuristics to consider data locality on heterogeneous systems.

8.1 Contributions

Throughout this thesis, we have shown that data-flow task programming provides a flexible way to exploit parallelism and loose synchronization. Parallelism is explicit and favors fine granularity that is essential on modern multicore and manycore architectures. In addition, data-flow dependencies provide an explicit memory view of the underlying architecture and abstract data transfers on disjoint address spaces. Unlike shared-memory models, detection of synchronizations is implicit to the application and expressed by access modes, *i.e*, a task executes if and only if its input parameters are produced.

We designed an execution mechanism to improve asynchronism on multi-GPU systems by concurrent GPU operations. This approach enabled the overlap of data transfer along with execution of GPU code avoiding explicit GPU synchronizations. Thanks to this overlapping, we were able to hide most of communication costs and reduce heterogeneity of memory accesses. Therefore, our concurrent GPU operations reduced synchronization and enabled efficient data-flow task programming on accelerators.

In addition, the use work stealing heuristics on multi-GPU is one of our key contributions on scheduling. Classic work stealing does not consider data locality in scheduling decisions and results on high data footprint. A locality heuristic based on owner-computes rule showed data transfer reduction and, consequently, significant performance speedup. Nonetheless, the support of multiple scheduling strategies is important on heterogeneous architectures. Although regular computations benefit from performance prediction and scheduling by cost models, imbalances can be addressed by low-overhead, receiver-initiated, scheduling such as work stealing.

Along with programming model and scheduling, we shown that the use of recursive tasks allow to unfold parallelism at runtime and to improve performance substantially. It is established that multicore processors are efficient on fine-grained task, while accelerators benefit from coarse-grained decomposition. A parallel algorithm with a hybrid decomposition, mixing fine-grained and coarse-grained tasks, can take full advantage of the underlying architecture.

We also studied the impact of data-flow task programming and work stealing on a manycore accelerator. Our experiments on the Intel Xeon Phi coprocessor showed XKaapi efficiency over native tools based on fork-join programming model. However, we concluded that compute-bound benchmarks lacked of vector operators for optimal performance.

This thesis also contributed to the XKaapi runtime development from the french team MOAIS (INRIA Rhônes-Alpes). In the last XKaapi source version, source control statistics reported that the multi-GPU version resulted in 440 commits and about $7,500$ additional or modified code lines.

8.2 Perspectives

The contributions of this thesis have raised several open questions. In this Section, we detail some of the possible research opportunities to better exploit parallel systems, mainly composed of accelerators.

8.2.1 Compiler Directives

The current compiler directives of XKaapi runtime offer task parallelism, data-flow dependencies, and adaptive loops (Lementec et al., 2011b). However, it does not offer task multi-versioning for heterogeneous systems.

Another possible approach is the design of a established standard API. A standard interface based on pragmas may avoid the re-write of applications on top of XKaapi. The OpenMP version 4.0 standard predicts the use of data dependencies and accelerators.

A future work includes an optimized runtime system for OpenMP 4.0 that may target different architectures such as multi-GPU and Intel Xeon Phi coprocessors. Since 4.0 it incorporates an accelerator model and data-flow dependencies by `depend` clause. Figure 8.1 shows an example of OpenMP 4.0 data dependencies for a matrix multiplication program.

```
1 void matmul(int NB, float A[NB][NB], float B[NB][NB], float C[NB][NB])
2 {
3 #pragma omp parallel
4 #pragma omp single
5   {
6     for(int i = 0; i < NB; i++)
7       for(j = 0; j < NB; j++)
8         for(k = 0; k < NB; k++)
9         #pragma omp task depend(in:A[i][k],B[k][j]) depend(inout:C[i][j])
10          matmul_tile( A[i][k], B[k][j], C[i][j] );
11  }
12 }
```

Figure 8.1 – Example of matrix multiplication on OpenMP 4.0 standard.

8.2.2 XKaapi Benchmarks

Throughout this work, we basically performed experimental results using linear algebra algorithms because they are studied by other related works in the context of task parallelism and heterogeneous systems. Nevertheless, a runtime system with different programming interfaces such as XKaapi may ideally have a set of benchmarks to evaluate its performance under new features, or architectures.

A suggested future work is the support of task parallel benchmarks in order to evaluate the XKaapi runtime. There are a number of benchmarks for OpenMP, Cilk Plus, and CUDA, that cover a range of applications such as PBBS (Shun et al., 2012), BOTS (Duran et al., 2009a), Parboil (Stratton et al., 2012), and Rodinia (Che et al., 2009, 2010).

8.2.3 Intel Xeon Phi Extensions

A perspective from this thesis is future extensions of XKaapi runtime for the Intel Xeon Phi coprocessor. The first research question is if work stealing heuristics can improve scalability on native execution. This line would follow our previous research on multi-GPU scheduling. Besides, a promising topic is to study the performance of PCIe interconnected multi-Intel Xeon Phi architectures. In this scenario, a system may run several instances of the XKaapi runtime, one for each coprocessor and one for the host system, that communicate through PCIe and message passing. The programming model remains unmodified, while scheduling may use distributed-memory techniques for load balancing.

8.2.4 Parallel Adaptive Algorithms

Most of parallel algorithms in this work are expressed by task parallelism and data dependencies, whose execution results in a DAG of tasks or a DFG considering their data dependencies. Still, a parallel algorithm implies in an overhead calculated by

$$T_{overhead} = T_1/T_{serial} \tag{8.1}$$

where T_1 is the execution of the parallel algorithm with one processor. Such overhead may prevent applications to scale in manycore architectures.

The concept of parallel adaptive algorithms is simple: to unfold parallelism only when necessary. Hence, in the context of work stealing scheduling, it creates *parallel work* only with idle resources at stealing requests. The adaptive algorithm can be expressed by two operations: `extract_seq` and `extract_par`. A worker will extract sequential work by `extract_seq` with no parallel overhead until it becomes idle. Then, it will try to extract parallel work from the computation in progress by `extract_par`. Consequently, a worker will execute the sequential version most of time and will minimize the work T_1/P. Traoré et al. (2008) show promising results using STL algorithms designed over parallel adaptive algorithms.

A future work in the context of this thesis is the design of adaptive algorithms on heterogeneous systems. Such approach can allow the use of variable grain sizes of sequential work since accelerators are able to execute bigger computations than CPUs in general.

8.2.5 Exascale Systems

An exascale system refers to a computing system capable to attain performance over one exaflop (10^{18} operations). Dongarra et al. (2011) list several assumptions about exascale systems on hardware and software level, as well as programming models. For instance, it is essential that a programming model offers support for multiple levels of parallelism. Those levels include distributed and shared architectures composed of accelerators such as GPUs and Xeon Phi coprocessors. In addition, the runtime system should be able to exploit parallelism over millions of cores

A research question based on this thesis is if data-flow task programming model may produce efficient parallel algorithms for multi-level systems. Dongarra et al. (2011) suggests that a programming model for exascale may provide interoperability between stablished models such as MPI and OpenMP. We believe that recent extensions to OpenMP indicate the acceptance of data-flow task programming as a programming model for exascale. In addition, parallel overhead may be addressed by adaptive parallel algorithms.

Another future perspective is the research on distributed support using XKaapi. In this thesis we addressed issues on a single system composed of multicore CPUs and many-core coprocessors such as GPU and Xeon Phi. However, we did not develop solutions on distributed-memory systems. Previous works on Athapascan/KAAPI describe strategies to data-flow programming on distributed systems (Galilee et al., 1998; Gautier et al., 2007). The contribution of this thesis combined with previous works on Athapascan may give interesting results.

Bibliography

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proc. of ACM SPAA*, SPAA '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-185-2. doi: 10.1145/341800.341801. (Cited on pages 2, 23, 28, 66, 82 and 86.)

Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen-mei W Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, 2010. URL `http://hal.inria.fr/inria-00547847/en/`. (Cited on pages 79, 89, 95, 106, 107, 139 and 140.)

Emmanuel Agullo, Cedric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. Lu factorization for accelerator-based systems. In *Proceedings of the 2011 9th IEEE/ACS International Conference on Computer Systems and Applications*, AICCSA '11, pages 217–224, Washington, DC, USA, 2011a. IEEE Computer Society. ISBN 978-1-4577-0475-8. doi: 10.1109/AICCSA.2011.6126599. URL `http://dx.doi.org/10.1109/AICCSA.2011.6126599`. (Cited on pages 79, 89, 93 and 95.)

Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *Proc. of the 25th IEEE IPDPS*, USA, 2011b. (Cited on pages 16, 79, 89, 90, 93 and 95.)

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006. (Cited on pages 19 and 22.)

Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1562764.1562783. (Cited on page 1.)

C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *Proc. of the 16th ICPADS*, pages 291–298, 2010a. doi: 10.1109/ICPADS.2010.129. (Cited on pages 2, 27, 28 and 151.)

Cédric Augonnet. *Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective*. PhD thesis, Université Bordeaux 1, Talence, France, December 2011. URL `http://tel.archives-ouvertes.fr/tel-00635651/fr/`. (Cited on pages 40, 83 and 84.)

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Maik Nijhuis. Exploiting the cell/be architecture with the starpu unified runtime system. In *Proc. of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pages 329–339, Berlin, Heidelberg, 2009a. Springer-Verlag. ISBN 978-3-642-03137-3. doi: 10.1007/978-3-642-03138-0_36. (Cited on pages 13, 40 and 46.)

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 863–874. Springer Berlin / Heidelberg, 2009b. (Cited on pages 2, 40, 51 and 52.)

Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In *Proc. of Euro-Par*, pages 56–65. Springer-Verlag, 2010b. ISBN 3-642-14121-8, 978-3-642-14121-8. (Cited on pages 79, 84 and 95.)

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011. ISSN 1532-0634. doi: 10.1002/cpe.1631. (Cited on pages 40, 64, 65, 70, 79, 82, 95 and 151.)

Eduard Ayguadé, Rosa Badia, Francisco Igual, Jesús Labarta, Rafael Mayo, and Enrique Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 851–862. Springer Berlin / Heidelberg, 2009a. (Cited on pages 2, 27, 28, 40, 41, 48, 51 and 52.)

Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The Design of OpenMP Tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009b. ISSN 1045-9219. doi: 10.1109/TPDS.2008.105. (Cited on pages 22 and 31.)

R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurr. Comput.: Pract. Exper.*, 21:2438–2456, 2009. ISSN 1532-0626. doi: 10.1002/cpe.v21:18. (Cited on page 41.)

K.J. Barker, K. Davis, A. Hoisie, D.K. Kerbyson, M. Lang, Scott Pakin, and J.C. Sancho. Entering the petaflop era: The architecture and performance of roadrunner. In *Proc. of the 2008 ACM/IEEE Supercomputing*, pages 1–11, 2008. doi: 10.1109/SC.2008.5217926. (Cited on page 12.)

Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a programming model for the cell BE architecture. In *Proc. of the ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: http://doi.acm.org/10.1145/1188455.1188546. (Cited on page 13.)

Michael A. Bender and Cynthia A. Phillips. Scheduling dags on asynchronous processors. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 35–45, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-667-7. doi: 10.1145/1248377.1248384. URL `http://doi.acm.org/10.1145/1248377.1248384`. (Cited on page 23.)

R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27:202–229, 1998. ISSN 0097-5397. doi: 10.1137/S0097539793259471. (Cited on pages 23, 28 and 82.)

Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '97, pages 10–10, Berkeley, CA, USA, 1997. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1268680.1268690`. (Cited on page 28.)

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995. ISSN 0362-1340. doi: 10.1145/209937.209958. URL `http://doi.acm.org/10.1145/209937.209958`. (Cited on pages 23, 28, 32, 45 and 51.)

George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 38(1–2):37–51, 2012. ISSN 0167-8191. doi: 10.1016/j.parco.2011.10.003. (Cited on page 80.)

Francois Broquedis, Thierry Gautier, and Vincent Danjean. libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms. In *Proc. of the 12th IWOMP*, pages 102–115, Rome, Italy, 2012. (Cited on pages 47, 51, 52, 101 and 114.)

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *Proc. of the ACM SIGGRAPH*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM. doi: http://doi.acm.org/10.1145/1186562.1015800. (Cited on page 45.)

Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesús Labarta. Productive cluster programming with OmpSs. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 555–566, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23399-9. URL `http://dl.acm.org/citation.cfm?id=2033345.2033405`. (Cited on pages 2, 27, 28, 41 and 46.)

Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Productive Programming of GPU Clusters with OmpSs. In *Proc. of the IEEE IPDPS*, 2012. (Cited on pages 2, 22, 27, 28, 41, 48, 51, 52, 65, 66, 79, 82, 95, 100 and 151.)

Javier Bueno, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Implementing ompss support for regions of data in architectures with multiple address spaces. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 359–368, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2130-3. doi: 10.1145/2464996.2465017. (Cited on pages 41, 42 and 61.)

Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1): 38–53, 2009. ISSN 0167-8191. doi: DOI:10.1016/j.parco.2008.10.002. (Cited on pages 1, 22, 28, 53, 68, 77, 89, 106, 139 and 151.)

Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 57–64, Aire-la-Ville, Switzerland, 2008. Eurographics Association. ISBN 978-3-905674-09-5. URL `http://dl.acm.org/citation.cfm?id=1413957.1413967`. (Cited on page 26.)

Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, Cambridge, USA, 2007. (Cited on pages 20, 22, 31, 38 and 48.)

Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IISWC*, pages 44–54, 2009. doi: 10.1109/IISWC.2009.5306797. (Cited on page 120.)

Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proc. of the IISWC*, pages 1–11, 2010. doi: 10.1109/IISWC.2010.5650274. (Cited on page 120.)

UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005. (Cited on page 21.)

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848. (Cited on page 33.)

Ludovic Coutér. C Language Extensions for Hybrid CPU/GPU Programming with StarPU. Research Report RR-8278, INRIA, April 2013. URL `http://hal.inria.fr/hal-00807033`. (Cited on page 40.)

Guilherme Cox, Andre Maximo, Cristiana Bentes, and Ricardo Farias. Irregular grid raycasting implementation on the cell broadband engine. In *Proc. of the 21st SBAC-PAD*, pages 93–100, October 2009. doi: 10.1109/SBAC-PAD.2009.15. (Cited on page 13.)

T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. Openmp programming on intel xeon phi coprocessors: An early performance comparison. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 38–44, November 2012. ISBN 978-3-00-039545-1. (Cited on pages 44 and 99.)

Gregory F Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 197–200, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-997-5. doi: http://doi.acm.org/10.1145/1383422.1383447. (Cited on page 45.)

James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proc. of the 2009 ACM/IEEE Supercomputing*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. doi: 10.1145/1654059.1654113. URL `http://doi.acm.org/10.1145/1654059.1654113`. (Cited on page 28.)

R Dolbeau, S Bihan, and F Bodin. HMPP: A hybrid multi-core parallel programming environment. In *Proc. of the Workshop on GPGPU*, 2007. (Cited on page 45.)

J. Dongarra, M. Gates, A. Haidar, Y. Jia, K. Kabir, P. Luszczek, and S. Tomov. MAGMA MIC 1.0: Linear Algebra Library for Intel Xeon Phi Coprocessors, 2013a. URL `http://icl.eecs.utk.edu/projectsfiles/magma/pubs/MAGMA_MIC_1.pdf`. (Cited on page 45.)

Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, editors. *Sourcebook of parallel computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 1-55860-871-0. (Cited on pages 11 and 18.)

Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011. ISSN 1094-3420. doi: 10.1177/1094342010391989. URL `http://dx.doi.org/10.1177/1094342010391989`. (Cited on page 121.)

Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimire Tomov. Dense linear algebra on accelerated multicore hardware. In Michael W. Berry, Kyle A. Gallivan, Efstratios Gallopoulos, Ananth Grama, Bernard Philippe, Yousef Saad, and Faisal Saied, editors, *High-Performance Scientific Computing*, pages 123–146. Springer London, 2012. ISBN 978-1-4471-2436-8. doi: 10.1007/978-1-4471-2437-5_5. URL `http://dx.doi.org/10.1007/978-1-4471-2437-5_5`. (Cited on pages 79 and 95.)

Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. Achieving numerical accuracy and high performance using recursive tile lu factorization with partial pivoting. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2013b. ISSN 1532-0634. doi: 10.1002/cpe.3110. URL `http://dx.doi.org/10.1002/cpe.3110`. (Cited on page 142.)

A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proc. of the ICPP'09*, pages 124–131, 2009a. doi: 10.1109/ICPP.2009.64. (Cited on pages 111 and 120.)

Alejandro Duran, Roger Ferrer, Eduard Ayguadé, Rosa M. Badia, and Jesus Labarta. A proposal to extend the openmp tasking model with dependent tasks. *Int. J. Parallel Program.*, 37:292–305, June 2009b. ISSN 0885-7458. doi: http://dx.doi.org/10.1007/s10766-009-0101-1. (Cited on pages 100 and 106.)

John Eisenlohr, David E. Hudak, Karen Tomko, and Timothy C. Prince. Dense linear algebra factorization in openmp and cilk plus on intel mic architecture: Development experiences and performance analysis, April 2012. (Cited on pages 45 and 99.)

Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. Identifying the Key Features of Intel Xeon Phi: A Comparative Approach. Technical Report Identifying the Key Features of Intel Xeon Phi: A Comparative Approach, Delft University of Technology, 2013a. (Cited on page 18.)

Jianbin Fang, Ana Lucia Varbanescu, Henk Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu. Benchmarking Intel Xeon Phi to Guide Kernel Design. Technical Report PDS-2013-005, Delft University of Technology, 2013b. (Cited on page 18.)

Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: http://doi.acm.org/10.1145/1188455.1188543. (Cited on page 45.)

M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071. (Cited on page 10.)

Ian Foster. *Designing and Building Paralllel Programs*. Addison-Wesley, 1995. URL `http://www.mcs.anl.gov/dbpp`. (Cited on page 21.)

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998. ISSN 0362-1340. doi: 10.1145/277652.277725. URL `http://doi.acm.org/10.1145/277652.277725`. (Cited on pages 20, 23, 24, 28, 33, 48, 51, 52, 82 and 114.)

Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proc. of the 21st SPAA*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-606-9. doi: 10.1145/1583991.1584017. (Cited on pages 20, 32 and 52.)

F. Galilee, G.G.H. Cavalheiro, J.-L. Roch, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *Proc. of the 1998 PACT*, pages 88–95, 1998. doi: 10.1109/PACT.1998.727176. (Cited on pages 22, 31, 34, 45, 47, 51, 104 and 121.)

Benedict R. Gaster, Lee Howes, David Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 2012. (Cited on pages 38 and 45.)

Thierry Gautier, Xavier Besseron, and Laurent Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proc. of PASCO'07*, London, Canada, 2007. ACM. ISBN 978-1-59593-741-4. (Cited on pages 20, 28, 34, 45, 47, 48, 51, 100, 104 and 121.)

Thierry Gautier, Fabien Lementec, Vincent Faucher, and Bruno Raffin. X-KAAPI: a Multi Paradigm Runtime for Multicore Architectures. In *Proc. of the 42st ICPP Workshops*, Lyon, France, October 2013a. (Cited on page 28.)

Thierry Gautier, Joao V.F. Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *Proc. of the IEEE 27th IPDPS*, pages 1299–1308, 2013b. doi: 10.1109/IPDPS.2013.66. (Cited on pages 3, 60, 80, 85, 86, 101, 106, 107, 108, 109 and 142.)

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley, USA, 2th edition, 2003. ISBN 0-201-64865-2. (Cited on pages 20 and 21.)

William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, USA, 1999. (Cited on page 20.)

M. Gschwind, D. Erb, S. Manning, and M. Nutter. An open source environment for cell broadband engine system software. *Computer*, 40(6):37–47, June 2007. ISSN 0018-9162. doi: 10.1109/MC.2007.192. (Cited on page 13.)

Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2006.41. (Cited on page 12.)

Yi Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proc. of the IEEE IPDPS*, pages 1–12, 2009. doi: 10.1109/IPDPS.2009.5161079. (Cited on pages 24, 25 and 28.)

Yi Guo, Jisheng Zhao, V. Cave, and V. Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *Proc. of IEEE IPDPS*, pages 1 –12, 2010. doi: 10.1109/IPDPS.2010.5470425. (Cited on pages 23, 67 and 78.)

Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, George Chrysos, Aniruddha G Shet, and Pradeep Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on intel(r) xeon phi(tm) coprocessor. In *Proc. of the 27th IEEE IPDPS*, Boston, USA, May 2013. IEEE. (Cited on pages 45 and 99.)

D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. of the ACM SPAA*, pages 355–364, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0079-7. doi: 10.1145/1810479.1810540. (Cited on page 53.)

Everton Hermann. *Interactive Physical Simulation on Multi-core and Multi-GPU Architectures*. PhD thesis, Institut National Polytechnique de Grenoble - INPG, 2010. (Cited on pages 4, 27, 28, 42 and 77.)

Everton Hermann, Bruno Raffin, Franĉois Faure, Thierry Gautier, and Jérémie Allard. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In *Proc. of Euro-Par*, volume 6272, pages 235–246. Springer, 2010. doi: 10.1007/978-3-642-15291-7_23. (Cited on pages 27, 46, 47, 50, 56 and 80.)

P. N. Hilfinger, Dan Bonachea, Kaushik Datta, David Gay, Susan Graham, Amir Kamil, Ben Liblit, Geoff Pike, Jimmy Su, and Katherine Yelick. Titanium language reference manual. Technical Report UCB/EECS-2005-15, Computer Science Division, University of California, Berkeley, 2005. (Cited on page 21.)

M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008. ISSN 0018-9162. doi: 10.1109/MC.2008.209. (Cited on page 9.)

Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, January 1987. ISSN 0004-5411. doi: 10.1145/7531.7535. URL `http://doi.acm.org/10.1145/7531.7535`. (Cited on page 87.)

Mitch Horton, Stanimire Tomov, and Jack Dongarra. A class of hybrid lapack algorithms for multicore and gpu architectures. In *Proc. of the 2011 SAAHPC*, pages 150–158, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4448-9. doi: 10.1109/SAAHPC.2011.18. URL `http://dx.doi.org/10.1109/SAAHPC.2011.18`. (Cited on pages 79 and 95.)

Jim Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*, volume 1. Morgan Kaufmann, 2013. (Cited on pages 16, 18, 44, 101 and 115.)

Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. Scaling Hierarchical N-body Simulations on GPU Clusters. In *Proc. of the 2010 ACM/IEEE Supercomputing*, pages 1–11, 2010. doi: 10.1109/SC.2010.49. (Cited on page 39.)

J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5): 589–604, 2005. ISSN 0018-8646. doi: 10.1147/rd.494.0589. (Cited on page 12.)

Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM. ISBN 0-89791-587-9. doi: 10.1145/165854.165874. (Cited on page 20.)

Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, and Denis Trystram. Scheduling independent tasks on multi-cores with gpu accelerators. In *Proc. of the 11th HeteroPar Workshop*, 2013. (Cited on pages 80, 85, 87 and 95.)

David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 2nd edition, 2012. (Cited on pages 1, 11, 13, 22, 38, 39 and 45.)

P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, March 2005. ISSN 0272-1732. doi: 10.1109/MM.2005.35. (Cited on page 11.)

J. Kurzak, A. Buttari, P. Luszczek, and J. Dongarra. The playstation 3 for high-performance scientific computing. *Computing in Science & Engineering*, 10(3):84–87, May 2008. ISSN 1521-9615. doi: 10.1109/MCSE.2008.85. (Cited on page 13.)

J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurr. Comput. : Pract. Exper.*, 22:15–44, 2010. ISSN 1532-0626. doi: 10.1002/cpe.v22:1. (Cited on pages 77, 100, 106 and 107.)

J. Labarta and V. Beltran. Prototype programming environment in booster node, delivrable d5.1, eu deep project dynamical exascale entry platform. Technical Report FP7-ICT-2011-7, February 2013. (Cited on pages 44 and 99.)

Edward A. Lee. The problem with threads. *Computer*, 39:33–42, 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.180. (Cited on page 99.)

Charles E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-497-3. doi: 10.1145/1629911.1630048. URL `http://doi.acm.org/10.1145/1629911.1630048`. (Cited on pages 32 and 45.)

Fabien Lementec, Vincent Danjean, and Thierry Gautier. X-Kaapi C programming interface. Rapport Technique RT-0417, INRIA, December 2011a. (Cited on page 47.)

Fabien Lementec, Thierry Gautier, and Vincent Danjean. The X-Kaapi's Application Programming Interface. Part I: Data Flow Programming. Rapport Technique RT-0418, INRIA, December 2011b. (Cited on pages 47, 48, 112 and 119.)

J. V. F. Lima, Thierry Gautier, Nicolas Maillard, and Vincent Danjean. Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs. In *Proc. of the 24th SBAC-PAD*, pages 75–82, New York, USA, 2012. IEEE. (Cited on pages 3, 60, 80, 95 and 153.)

J. V. F. Lima, F. Broquedis, T. Gautier, and B. Raffin. Preliminary Experiments with XKaapi on Intel Xeon Phi Coprocessor. In *Proc. of the 25th SBAC-PAD*, Porto de Galinhas, Brazil, October 2013. doi: 10.1109/SBAC-PAD.2013.28. (Cited on pages 3 and 99.)

Michael D Linderman, Jamison D Collins, Hong Wang, and Teresa H Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGOPS Oper. Syst. Rev.*, 42 (2):287–296, 2008. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1353535.1346318. (Cited on page 45.)

E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008. ISSN 0272-1732. doi: 10.1109/MM.2008.31. (Cited on page 13.)

Michael D McCool, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin. Performance evaluation of GPUs using the RapidMind development platform. In *Proc. of the 2006 ACM/IEEE Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: http://doi.acm.org/10.1145/1188455.1188642. (Cited on page 45.)

Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *Proc. of the 5th PGAS*, 2011. (Cited on page 28.)

Chris J. Newburn, Serguei Dmitriev, Ravi Narayanaswamy, John Wiegert, Ravi Murty, Francisco Chinchilla, Rajiv Deodhar, and Russ McGuire. Offload compiler runtime for the intel xeon phi coprocessor. In *Proc. of the 27th IEEE IPDPS Workshops and PhD Forum*, 2013. (Cited on pages 22, 44 and 99.)

Rob van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Satin: Efficient parallel divide-and-conquer in java. In *Proc. of the 6th Euro-Par*, Euro-Par '00, pages 690–699, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67956-1. (Cited on page 28.)

Rob van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proc. of the 8th ACM SIGPLAN PPoPP*, PPoPP '01, pages 34–43, New York, NY, USA, 2001. ACM. ISBN 1-58113-346-4. doi: 10.1145/379539.379563. (Cited on page 28.)

Robert W. Numrich and John Reid. Co-arrays in the next fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, August 2005. ISSN 1061-7264. doi: 10.1145/1080399.1080400. URL `http://doi.acm.org/10.1145/1080399.1080400`. (Cited on page 21.)

Institute of Electrical and Inc. Electronic Engineers. Information Technology – Portable Operating Systems Interface (POSIX) – Part: System Application Program Interface (API) – Amendment 2: Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE, New York, NY, 1995. (Cited on pages 20 and 99.)

M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband engine processor. *IBM Systems Journal*, 45(1):85–102, 2006. ISSN 0018-8670. (Cited on page 13.)

OpenMP Architecture Review Board, 1997-2013. Available from Internet: `http://www.openmp.org`. (Cited on page 22.)

J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. ISSN 0018-9219. doi: 10.1109/JPROC.2008.917757. (Cited on page 13.)

J. Panetta, P. Souza, C. Cunha, A. Romanelli, F. Roxo, I. Pedrosa, S. Sinedino, L. Monnerat, L. Carneiro, and C. Albrecht. Seismic imaging on novel computer architectures. In *Proc. of the 11th International Congress of the Brazilian Geophysical Society*, August 2009. (Cited on page 13.)

Matthew Papakipos. The PeakStream Platform for Many-Core Computing, 2007. (Cited on page 45.)

S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis. Exploring simd for molecular dynamics, using intel xeon processors and intel xeon phi coprocessors. In *Proc. of the 27th IEEE IPDPS*, 2013. (Cited on pages 18, 44 and 99.)

D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. In *Proc. of the IEEE ISSCC*, pages 184–592 Vol. 1, 2005. doi: 10.1109/ISSCC.2005.1493930. (Cited on page 12.)

Laurent Pigeon. *Environnement interopérable distribué pour les simulations numériques avec composants CAPE-OPEN*. PhD thesis, Institut National Polytechnique de Grenoble (INPG), September 2007. (Cited on page 27.)

J. Planas, R.M. Badia, E. Ayguade, and J. Labarta. Self-adaptive ompss tasks in heterogeneous environments. In *Proc. of the IEEE 27th IPDPS*, pages 138–149, 2013. doi: 10.1109/IPDPS.2013.53. (Cited on pages 27, 41 and 42.)

Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. *SIGPLAN Not.*, 44(4):121–130, 2009. ISSN 0362-1340. doi: 10.1145/1594835.1504196. (Cited on pages 42 and 65.)

Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *Proc. of the 16th Euro-Par: Part I*, EuroPar'10, pages 217–229, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15276-7, 978-3-642-15276-4. URL `http://dl.acm.org/citation.cfm?id=1887695.1887719`. (Cited on page 28.)

Kaushik Ravichandran, Sangho Lee, and Santosh Pande. Work stealing for multi-core hpc clusters. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 205–217, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23399-9. (Cited on page 28.)

James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly & Associates, Inc., Sebastopol, USA, 2007. ISBN 9780596514808. (Cited on pages 20, 33, 45 and 48.)

Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, June 1993. ISSN 0018-9162. doi: 10.1109/2.214440. URL `http://dx.doi.org/10.1109/2.214440`. (Cited on page 35.)

Jean-Louis Roch, Rémi Revire, and Thierry Gautier. Athapascan : an API for Asynchronous Parallel Programming User's Guide. Rapport de recherche RT-0276, INRIA, February 2003. (Cited on pages 34, 49 and 140.)

Jean-Louis Roch, Daouda Traoré, and Julien Bernard. On-line adaptive parallel prefix computation. In WolfgangE. Nagel, WolfgangV. Walter, and Wolfgang Lehner, editors, *Euro-Par 2006 Parallel Processing*, volume 4128 of *Lecture Notes in Computer Science*, pages 841–850. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-37783-2. doi: 10.1007/11823285_88. URL `http://dx.doi.org/10.1007/11823285_88`. (Cited on page 28.)

Claudio Schepke, Nicolas Maillard, Joerg Schneider, and Hans-Ulrich Heiss. Online Mesh Refinement for Parallel Atmospheric Models. *International Journal of Parallel Programming*, 41(4):552–569, 2013. ISSN 0885-7458. doi: 10.1007/s10766-012-0235-4. URL `http://dx.doi.org/10.1007/s10766-012-0235-4`. (Cited on page 4.)

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, August 2008. ISSN 0730-0301. doi: 10.1145/1360612.1360617. (Cited on page 15.)

Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proc. of the 24th ACM SPAA*, SPAA '12, pages 68–70, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1213-4. doi: 10.1145/2312005.2312018. URL `http://doi.acm.org/10.1145/2312005.2312018`. (Cited on page 120.)

Fengguang Song and Jack Dongarra. A scalable framework for heterogeneous GPU-based clusters. In *Proc. of ACM SPAA*, pages 91–100, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1213-4. doi: 10.1145/2312005.2312025. (Cited on pages 77, 79, 95 and 140.)

J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. IMPACT Technical Report IMPACT-12-01, University of Illinois, at Urbana-Champaign, March 2012. (Cited on page 120.)

Takaken. Source code for n queens problem. URL `http://www.ic-net.or.jp/home/takaken/e/queen`. Available from Internet: `http://www.ic-net.or.jp/home/takaken/e/queen`, Cited Jan. 2014. (Cited on page 104.)

Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Le Mentec, and Bruno Raffin. A Work Stealing Algorithm for Parallel Loops on Shared Cache Multicores. In *Proc. of the 4th HPPC Workshop*, August 2010a. (Cited on page 28.)

Marc Tchiboukdjian, Vincent Danjean, and Bruno Raffin. Binary mesh partitioning for cache-efficient visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(5):815–828, 2010b. ISSN 1077-2626. doi: http://doi.ieeecomputersociety.org/10.1109/TVCG.2010.19. (Cited on page 28.)

Marc Tchiboukdjian, Nicolas Gast, and Denis Trystram. Decentralized list scheduling. *Annals of Operations Research*, pages 1–23, 2012. ISSN 0254-5330. doi: 10.1007/s10479-012-1149-7. (Cited on pages 53 and 56.)

William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proc. of the 19th PACT*, PACT '10, pages 365–376, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi: http://doi.acm.org/10.1145/1854273.1854319. (Cited on page 45.)

Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010. ISSN 0167-8191. doi: 10.1016/j.parco.2009.12.005. (Cited on pages 45, 70, 79, 89, 95 and 99.)

H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, March 2002. ISSN 1045-9219. doi: 10.1109/71.993206. (Cited on pages 25, 78, 79, 80, 85, 86 and 95.)

Julio Toss and Thierry Gautier. A new programming paradigm for gpgpu. In Christos Kaklamanis, Theodore Papatheodorou, and PaulG. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 895–907. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32819-0. doi: 10.1007/978-3-642-32820-6_88. URL `http://dx.doi.org/10.1007/978-3-642-32820-6_88`. (Cited on page 26.)

Daouda Traoré, Jean-Louis Roch, Nicolas Maillard, Thierry Gautier, and Julien Bernard. Deque-free work-optimal parallel stl algorithms. In Emilio Luque, Tomàs Margalef, and Domingo Benítez, editors, *Euro-Par 2008 – Parallel Processing*, volume 5168 of *Lecture Notes in Computer Science*, pages 887–897. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-85450-0. doi: 10.1007/978-3-540-85451-7_95. (Cited on pages 28 and 120.)

R. Vasudevan, Sathish S. Vadhiyar, and Laxmikant V. Kalé. G-charm: an adaptive runtime system for message-driven parallel applications on hybrid systems. In *Proc. of the 27th IEEE/ACM Supercomputing*, ICS '13, pages 349–358, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2130-3. doi: 10.1145/2464996.2465444. (Cited on page 39.)

A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide: Queueing and runtime for kernels. Technical Report ICL-UT-11-02, University of Tennessee, 2011. URL `http://icl.cs.utk.edu/quark`. (Cited on pages 47, 51, 52, 89, 110 and 151.)

Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. Renderants: interactive reyes rendering on gpus. In *Proc. of the ACM SIGGRAPH Asia*, SIGGRAPH Asia '09, pages 155:1–155:11, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-858-2. doi: 10.1145/1661412.1618501. URL `http://doi.acm.org/10.1145/1661412.1618501`. (Cited on page 26.)

Part III

Appendixes

Appendix A

Cholesky over XKaapi

This Appendix gives a complete example of the Cholesky factorization designed using XKaapi. This factorization is an ideal example for heterogeneous architectures since its compute-bound kernels are efficient and attain substantial performance gains. We describe the tiled algorithm from the literature and the code for a heterogeneous architecture composed of multi-CPU and multi-GPU. In addition, we show our parallel Cholesky algorithm with two level parallelism in which the factorization of a diagonal panel `POTRF` is split in sub-tiles.

A.1 Tiled Cholesky Algorithm

The Cholesky factorization (or decomposition) is mainly used for the numerical solution of linear equations $Ax = b$, where A is symmetric and positive definite. It has the form $A = LL^T$ where L is an $m \times n$ real lower triangular matrix with positive diagonal elements. Assuming a matrix $m \times n$ and $m = n$ divided in $n_{tiles} \times n_{tiles}$ tiles, Algorithm 9 illustrates the tiled Cholesky factorization as described by Agullo et al. (2010) and similar to the tiled version from Buttari et al. (2009). The cost to factor a matrix A is $\sim \frac{1}{3}n^3$ requering $O(n^3)$ operations.

Algorithm 9: The tiled algorithm for Cholesky factorization.

Input : An $m \times n$ lower triangular matrix A with positive diagonal elements with $n_{tiles} \times n_{tiles}$ tiles

```
1  for k ← 0 to n_tiles − 1 do
2  |  POTRF(A_kk)
3  |  for m ← k + 1 to n_tiles − 1 do
4  |  |  TRSM( A_kk, A_mk )
5  |  end
6  |  for m ← k + 1 to n_tiles − 1 do
7  |  |  SYRK( A_mk, A_mm )
8  |  |  for n ← k + 1 to m do
9  |  |  |  GEMM( A_mk, A_nk, A_mn )
10 |  |  end
11 |  end
12 end
```

A.2 XKaapi Data-Flow Version

In this section, we show the implementation of the Cholesky algorithm using the Kaapi++ API with multi-versioning for heterogeneous architectures. We split the code in three parts: main and initialization (Figure A.1 on the next page), the parallel Cholesky algorithm (Figure A.2 on page 142), and a task implementation with CPU and GPU version (Figure A.3 on page 144).

XKaapi initialization in Figure A.1 on the facing page (from line 20 to 23) creates a community group, as though Athapascan (Roch et al., 2003), and spawns the main task, which is the tree root of the generated DAG. Next, the `doit` struct encapsulates the equivalent "main" program of a XKaapi application and perform all data initializations. XKaapi requires data registering (line 8) and unregistering (line 11) in order to perform asynchronous operations on multi-GPU. The main task of the Cholesky algorithm is spawned through the `TaskCholesky` structure (line 9).

We used the task attribute `ka::SetStaticSched()` that is a scheduling optimization. It computes true dependencies of the tasks from the stack and creates a list of successors' tasks, also named *ready task list*. Subsequent steal operations in a thread with a ready task list have lower cost because the runtime does not need to compute true dependencies at each steal operation. Therefore, it moves the overhead of computing ready tasks from each steal operation to the task with `ka::SetStaticSched()` attribute. In the case of the Cholesky algorithm, this attribute can be used since tasks have regular computations and do not create recursive tasks. Besides, we note that tasks using `ka::SetStaticSched()` can not spawn more tasks recursively except for tasks using this attribute.

The left-looking Cholesky implementation with XKaapi is illustrated in Figure A.2 on page 142. Matrix tiles are expressed in ranges by `ka::rangeindex` structure in conjunction with a two dimensional range `ka::range2d`. A `ka::range2d` contains the number of rows (`A->dim(0)`) and columns (`A->dim(1)`) of a matrix, plus the leading dimension (`A->lda()`).

In addition, the XKaapi version makes use of the task attribute `ka::SetArch` to specify the target architecture. The `ka::ArchHost` restricts task execution over CPUs workers and `ka::ArchCUDA` restricts execution over CUDA GPUs workers. This task attribute is useful in cases when the programmer knowns beforehand the task performance for a specific architecture. In the Cholesky algorithm, it is well established and reported by other works that the panel decomposition `TaskPOTRF` is efficient on CPUs and the BLAS-3 matrix-matrix operations are efficient on GPUs (`TaskTRSM`, `TaskSYRK`, and `TaskGEMM`) (Agullo et al., 2010; Song and Dongarra, 2012).

Each task of the algorithm in Figure A.2 on page 142 has a *task signature* to use XKaapi task multi-versioning. The implementation of `TaskTRSM` is shown in Figure A.3 on page 144. The `TaskTRSM` signature is written at line 1 with the number of parameters and their access modes, in this case read for matrix `A` and read-write for matrix `B`. The CPU version of `TaskTRSM` at line 6 is a specialization of the `TaskBodyCPU` structure that will be executed by CPU workers. On the other hand, the GPU version at line 24 is a specialization of the `TaskBodyGPU` structure and will be executed by GPU workers. We note that the GPU version receives an additional parameter (`ka::gpuStream`) that gives the CUDA stream of the current GPU for asynchronous execution.

```
1 struct doit {
2   void operator()(int argc, char** argv )
3   {
4     /* n is the matrix order, nbsize the block size */
5     double* dA = (double*) calloc(n * n, sizeof(double));
6     ka::array<2,double> A(dA, n, n, n);
7     /* register memory for asynchronous transfers */
8     ka::Memory::Register( A );
9     ka::Spawn<TaskCholesky>(ka::SetStaticSched())( A, nbsize );
10    ka::Sync();
11    ka::Memory::Unregister( A ); /* unregister memory */
12    free(dA);
13  }
14 };
15
16 int main(int argc, char** argv)
17 {
18   try {
19     ka::Community com = ka::System::join_community(argc,argv);
20     ka::SpawnMain<doit>()(argc, argv);
21     com.leave();
22     ka::System::terminate();
23   }
24   catch (const std::exception& E) {
25     ka::logfile() << "Catch : " << E.what() << std::endl;
26   }
27   catch (...) {
28     ka::logfile() << "Catch unknown exception: " << std::endl;
29   }
30
31   return 0;
32 }
```

Figure A.1 – Initialization and data handling for the Cholesky program. Data is registered before execution and unregistered at the end.

From the algorithm introduced with data dependencies, the runtime unfolds the parallelism and creates the data-flow graph (DFG) illustrated in Figure A.4 on page 145. We note that the green tasks (`TaskPOTRF`), in the critical path, are executed by CPUs.

A.3 Parallel Diagonal Decomposition

The XKaapi runtime has support for recursive task creation and fine-grain parallelism. We designed a two-level parallel version of the Cholesky algorithm, similar to the tiled version of Figure A.2 on the next page, but with a parallel diagonal panel decomposition. The

```
1 struct TaskCholesky: public ka::Task<2>::Signature<
2   /* A and block size nbsize (int) */
3   ka::RPWP<ka::range2d<double> >, int
4 >{};
5
6 template<> struct TaskBodyCPU<TaskCholesky> {
7   void operator()( ka::range2d_rpwp<double> A , int nbsize )
8   {
9     const int N = A->dim(0);
10
11    for(int k=0; k < N; k+= nbsize ){
12      ka::rangeindex rk(k, k+nbsize);
13      ka::Spawn<TaskPOTRF>( ka::SetArch(ka::ArchHost) )
14          ( A(rk,rk) );
15      for(int m=k+nbsize; m < N; m+= nbsize){
16        ka::rangeindex rm(m, m+nbsize);
17        ka::Spawn<TaskTRSM>( ka::SetArch(ka::ArchCUDA) )
18            ( A(rk,rk), A(rm,rk) );
19      }
20      for(int m=k+nbsize; m < N; m+= nbsize){
21        ka::rangeindex rm(m, m+nbsize);
22        ka::Spawn<TaskSYRK>( ka::SetArch(ka::ArchCUDA) )
23            ( A(rm,rk), A(rm,rm) );
24        for(int n=k+nbsize; n < m; n+= nbsize ){
25          ka::rangeindex rn(n, n+nbsize);
26          ka::Spawn<TaskGEMM>( ka::SetArch(ka::ArchCUDA) )
27              ( A(rm,rk), A(rn,rk), A(rm,rn) );
28        }
29      }
30    }
31  }
32 };
```

Figure A.2 – Left-looking Cholesky implementation with XKaapi C++ API (kaapi++). It shows the task *Signature* with its parameters and access modes, as well as the parallel CPU version.

`TaskPOTRF` task of Figure A.2 (line 13) was substituted by the `TaskParallelPOTRF` parallel task. It splits the diagonal panel in sub-tiles of size 128×128 and applies the same algorithm for the tiled Cholesky through host-only tasks (attribute `ka::SetArch(ka::ArchHost)`).

Since the panel decomposition `TaskPOTRF` is on the critical path of the tiled Cholesky algorithm, we are able to achieve significant performance improvements and reduce the idle time of GPUs (Gautier et al., 2013b). Dongarra et al. (2013b) uses a recursive approach to improve the overall performance and sustain numerical quality for LU factorization with partial pivoting.

A.4 Version with Compiler Annotations

In the previous section, our C++ version of Cholesky is shown using task multi-versioning. XKaapi also supports source-to-source compilation with annotations through the `KaCC` compiler. An incremental version of Cholesky with annotations is illustrated in Figure A.6 on page 147.

```
1 struct TaskTRSM: public ka::Task<2>::Signature<
2      ka::R<ka::range2d<double> >, /* A */
3      ka::RW<ka::range2d<double> > /* B */
4 >{};
5
6 template<> struct TaskBodyCPU<TaskTRSM> {
7   void operator()(
8     ka::range2d_r<double> A, ka::range2d_rw<double> B
9   )
10  {
11    const double* const a = A->ptr();
12    const int lda = A->lda();
13    double* const b = B->ptr();
14    const int ldb = B->lda();
15    const int n = B->dim(0);
16    const int k = (transA == CblasNoTrans ? A->dim(1) :
17      A->dim(0) );
18
19    cblas_dtrsm( CblasRowMajor, CblasRight, CblasLower,
20      CblasTrans, CblasNonUnit, n, k, 1.0, a, lda, b, ldb );
21  }
22 };
23
24 template<> struct TaskBodyGPU<TaskTRSM> {
25   void operator()( ka::gpuStream stream,
26      ka::range2d_r<double> A, ka::range2d_rw<double> B )
27   {
28     const double* const a = A->ptr();
29     const int lda = A->lda();
30     double* const b = B->ptr();
31     const int ldb = B->lda();
32     const int n = B->dim(0);
33     const int k = (transA == CblasNoTrans ? A->dim(1) :
34       A->dim(0) );
35
36     cublasDtrsm(
37       kaapi_cuda_cublas_handle(), /* CUBLAS handle */
38       CUBLAS_SIDE_RIGHT, CUBLAS_FILL_MODE_LOWER,
39       CUBLAS_OP_T, CUBLAS_DIAG_NON_UNIT,
40        m, n, alpha, A, lda, B, ldb
41      );
42   }
43 };
```

Figure A.3 – Implementation of the `TRSM` task for the Cholesky algorithm. This task uses multi-versioning with task signature and CPU and GPU versions.

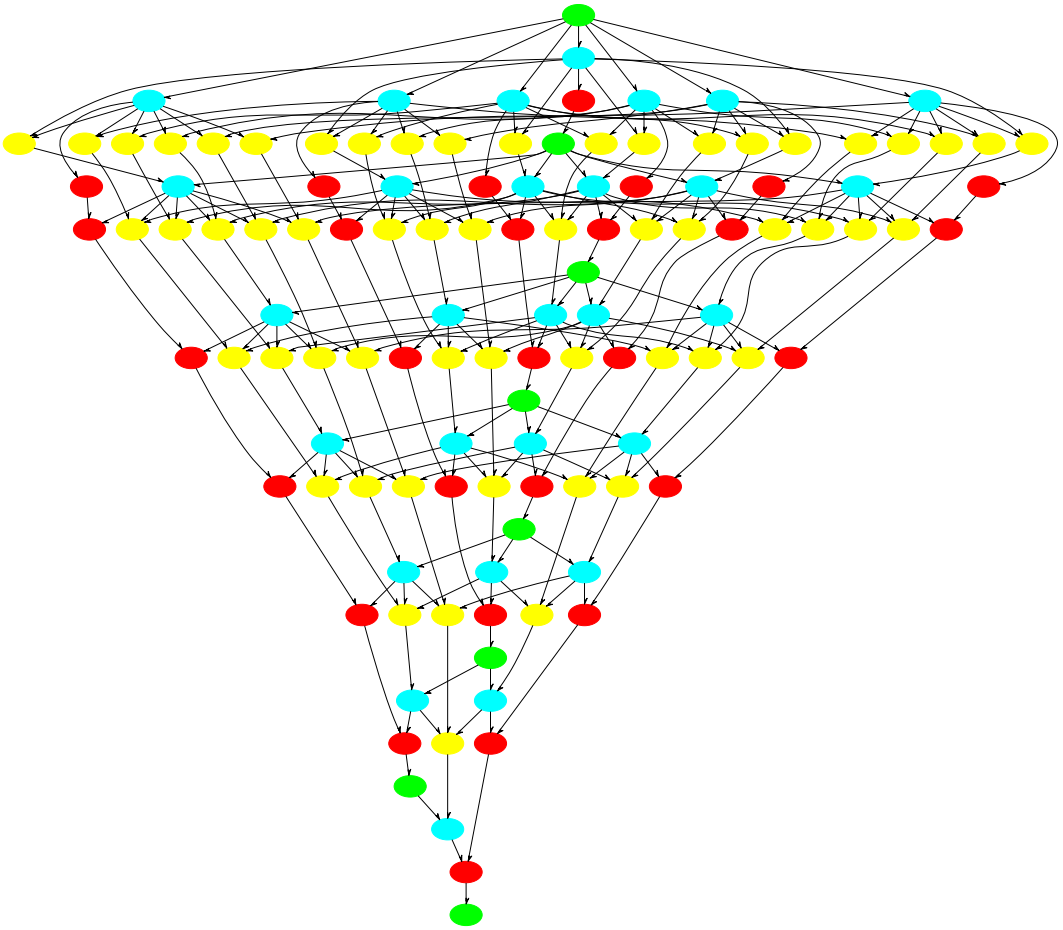


Figure A.4 – The resulting data-flow graph (DFG) of a Cholesky factorization.

```
1 struct TaskParallelPOTRF: public ka::Task<1>::Signature
2 <
3   ka::RPWP<ka::range2d<double> > /* A */
4 >{};
5
6 template<> struct TaskBodyCPU<TaskParallelPOTRF> {
7   void operator()( ka::range2d_rpwp<double> A )
8   {
9     const int N    = A->dim(0);
10    const int lda  = A->lda();
11    double* const a = A->ptr();
12    const int nbsize = 128; /* low-level block size */
13
14    if( N > nbsize ){
15      for (int k=0; k < N; k += nbsize) {
16        ka::rangeindex rk(k, k+nbsize);
17        ka::Spawn<TaskPOTRF>( ka::SetArch(ka::ArchHost) )
18          ( A(rk,rk) );
19        for (int m=k+nbsize; m < N; m += nbsize) {
20          ka::rangeindex rm(m, m+nbsize);
21          ka::Spawn<TaskTRSM>( ka::SetArch(ka::ArchHost) )
22            ( A(rk,rk), A(rm,rk) );
23        }
24        for (int m=k+nbsize; m < N; m += nbsize) {
25          ka::rangeindex rm(m, m+nbsize);
26          ka::Spawn<TaskSYRK>( ka::SetArch(ka::ArchHost) )
27            ( A(rm,rk), A(rm,rm));
28          for (int n=k+nbsize; n < m; n += nbsize) {
29            ka::rangeindex rn(n, n+nbsize);
30            ka::Spawn<TaskGEMM>( ka::SetArch(ka::ArchHost) )
31              ( A(rm,rk), A(rn,rk), A(rm,rn));
32          }
33        }
34      }
35    }
36    else
37      clapack_dpotrf(CblasRowMajor, CblasLower, N, a, lda);
38  }
39 };
```

Figure A.5 – Two-level Cholesky version with XKaapi.

```
1 void cholesky( double** A, int NB, int BS )
2 {
3   /* NB is the number of blocks, and BS the block size */
4   for (int k=0; k < NB; k++){
5 #pragma kaapi task readwrite(A[k*NB+k]{ld=BS; [BS][BS]})
6     clapack_dpotrf( CblasRowMajor, CblasLower,
7       BS, A[k*NB+k], BS );
8
9     for (int m=k+1; m < NB; m++){
10 #pragma kaapi task read(A[k*NB+k]{ld=BS; [BS][BS]}) \
11                  readwrite(A[m*NB+k]{ld=BS; [BS][BS]})
12       cblas_dtrsm( CblasRowMajor, CblasLeft, CblasLower,
13                    CblasNoTrans, CblasUnit,
14                    BS, BS, 1., A[k*NB+k], BS, A[m*N+k], BS );
15     }
16
17     for (int m=k+1; m < NB; m++){
18 #pragma kaapi task read(A[m*NB+k]{ld=BS; [BS][BS]}) \
19                  readwrite(A[m*NB+m]{ld=BS; [BS][BS]})
20       cblas_dsyrk( CblasRowMajor, CblasLower, CblasNoTrans,
21         BS, BS, -1.0, A[m*NB+k], BS, 1.0, A[m*NB+m], BS );
22
23       for (int n=k+1; n < m; n++){
24 #pragma kaapi task read(A[m*NB+k]{ld=BS; [BS][BS]}, \
25                  A[n*NB+k]{ld=BS; [BS][BS]}) \
26                  readwrite(A[m*NB+n]{ld=BS; [BS][BS]})
27         cblas_dgemm( CblasRowMajor, CblasNoTrans, CblasTrans,
28           BS, BS, BS, -1.0, A[m*NB+k], BS, A[n*NB+k], BS, 1.0,
29           A[m*NB+n], BS );
30       }
31     }
32   }
33 }
```

Figure A.6 – A Cholesky version with XKaapi compiler.

Appendix B

XKaapi Performance Model Results

This Appendix gives the results obtained from our performance model for task and data transfer in the experimental results of Chapter 6.

B.1 History-based Model Results

Figure B.1 shows the results of our performance model for task execution using tasks with multi-versioning. In this case, it displays the predicted execution time according to the block sizes applied for each task. These results suggest that CPU time consuming tasks obtain significant speedup on GPUs such as `DGEMM`. Besides, BLAS-2 tasks may not achieve speedup on GPUs such as `DPOTRF` (from Cholesky), `DGETRF` and `DTSTRF` (from LU).

(a) Execution time of each task. (b) Speedup of GPU over CPU.

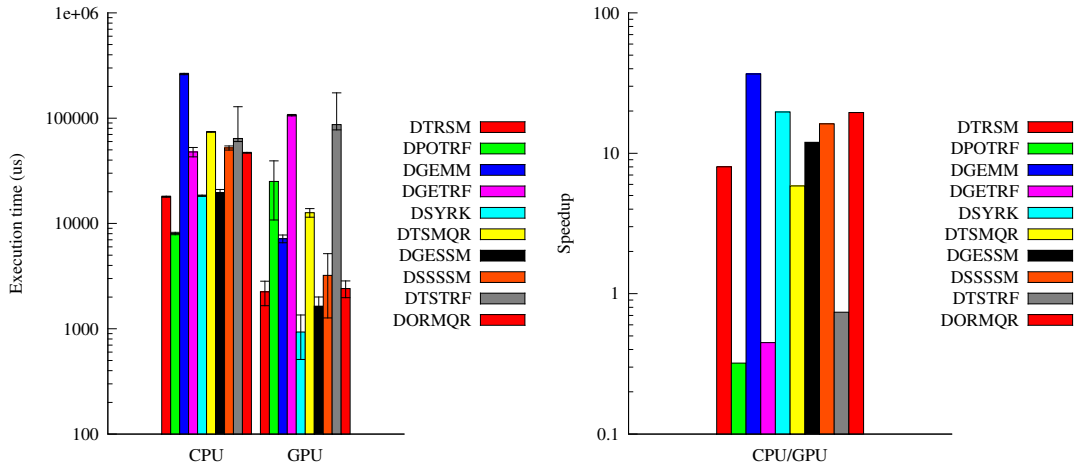


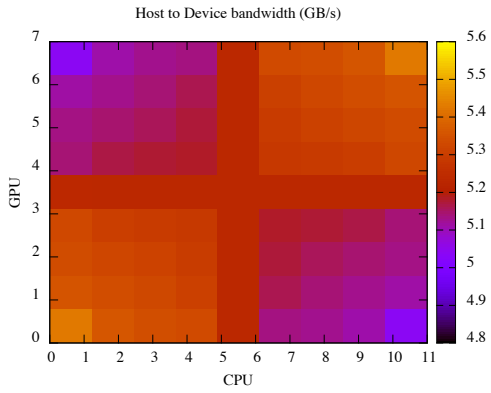
Figure B.1 – Performance model results in time (left) and speedup (right) for tasks with CPU and GPU versions. Logarithm scale was used for y axis.

B.2 Communication Bandwidth Results

The calculated communication bandwidth is illustrated in Figure B.2 on the next page through a color map with an approximation of bandwidth between resources. It appears that the distance between processors and the GPUs is relevant for the overall performance. The communication bandwidth can increase between resources interconnected in the same

QPI-PCIe bridge. For instance, the red areas in Figure B.2a (high bandwidth) were obtained from CPU 0 (cores [0...5]) to GPUs [0...3] and *vice versa*, which are connected in the same QPI-PCIe bridge according to the system topology (see Section 2.1.5 on page 16). Still, the purple squares appeared from CPU 0 to GPUs [4...7] connected by different QPI-PCIe bridge, and forcing the use of two QPI-PCIe bridges to route the transfer.

(a) CPU to GPU map.



(b) GPU to CPU map.

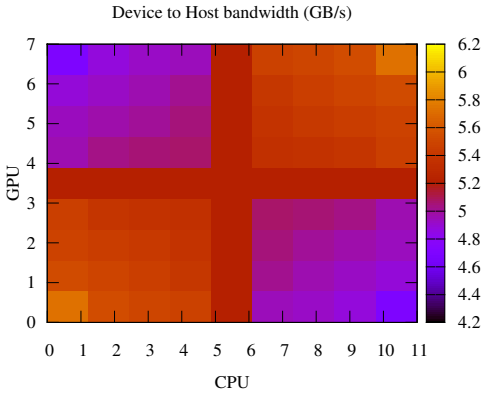


Figure B.2 – Approximation of communication bandwidth used in the performance model.

Appendix C

Experiments with XKaapi, OmpSs, and StarPU

This Appendix gives more results on XKaapi and other two runtime systems for multi-GPU systems: OmpSs and StarPU. Our goal is to evaluate the performance of XKaapi concerning other runtime systems for heterogeneous multi-GPU systems. We designed a Cholesky factorization (Section C.1) and a blocked matrix multiplication (Section C.2) over XKaapi (see Chapter 4 on page 47), StarPU (Augonnet et al., 2011), and OmpSs (Bueno et al., 2012). The algorithms of both benchmarks were based on PLASMA tiled algorithms described in Buttari et al. (2009).

StarPU and OmpSs versions of each algorithm were developed in their native programming model using library calls and compiler annotations, respectively. On XKaapi experiments, we implemented and extended the QUARK API (YarKhan et al., 2011) in XKaapi to support task multi-versioning. Each benchmark calls a registration function responsible to associate one PLASMA task to a GPU version. At the task execution, our QUARK version runs the appropriate task implementation if the target worker is a CPU or a GPU. The algorithms in our experiments are the same and use the same task granularity, *i.e*, the same block size.

In each runtime we chose a scheduling strategy that has published results in the literature. We used as scheduler:

- XKaapi locality-aware work stealing (H2) in order to improve data locality and reduce data invalidations (see Section 5.5.3 on page 66 for details);
- OmpSs locality-aware scheduler (named `affinity` by Nano++) described by Bueno et al. (2012) that reduces the number of transfers between devices;
- StarPU HEFT strategy detailed in Augonnet et al. (2010a) that minimizes the makespan or the schedule length.

All experiments have been conducted on the heterogeneous, multi-GPU system, named "Idgraf" composed of 8 GPUs and 12 CPUs. The machine topology is described in Section 2.1.5 on page 16. We used as software environment GNU/Linux Debian *squeeze* x86/64, the compiler GCC 4.4, CUDA 5.0, and the library ATLAS 3.9.39 (BLAS and LAPACK). We also used StarPU version 1.0.5, and in OmpSs the Mercurium compiler version 1.3.5.8 and Nano++ runtime version 0.7a.

In each experiment, we show in the x-axis the number of resources as the number of CPUs and GPUs for each execution. We employ this notation to clearly distinguish the number of computing CPUs and GPUs at runtime. Since all three runtime systems dedicate a CPU to manage a GPU, the number of computing CPUs is the number of total CPUs minus the number of GPUs.

C.1 Cholesky

Figure C.1 illustrates performance results of Cholesky factorization on the three runtime systems. XKaapi outperformed the other tools for all cases with peak performance of 428.8490 GFlop/s with 5 GPUs and 7 CPUs, almost 1.55 times better than OmpSs. It seems that StarPU incurred more data footprint than other tools and had poor performance on this benchmark. Besides, we believe that StarPU may improve performance with bigger matrices since it would benefit of data prefetch.

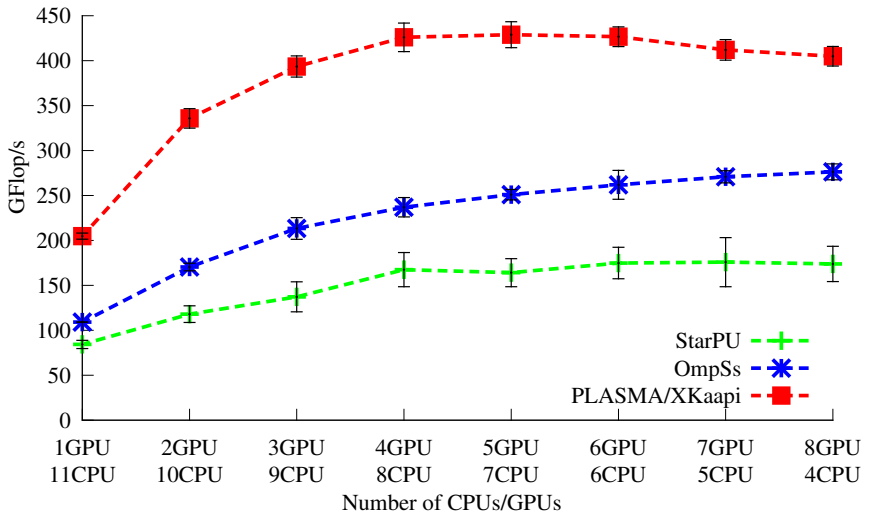


Figure C.1 – Performance results of Cholesky factorization for a matrix size of 10240×10240 and block size 1024×1024.

C.2 Blocked Matrix Multiplication

Figure C.2 on the next page shows performance results of blocked matrix multiplication on the three runtime systems. As seen in experiments with Cholesky, XKaapi outperformed the other tools for all cases with peak performance of 1577.82 GFlop/s with 8 GPUs and 4 CPUs, 1.19 times better than OmpSs. Again, we believe that StarPU high footprint affected its performance for this benchmark.

C.3 Summary

These experiments have demonstrated that XKaapi is able to attain significant performance results compared to other runtime systems with data-flow programming model such as OmpSs and StarPU. These results can be explained by XKaapi scheduling by work stealing and capacity of overlap data transfer with GPU code execution.

OmpSs achieved better performance results than StarPU in our experiments. One possible explanation is that OmpSs locality-aware scheduling would have lower data foot-

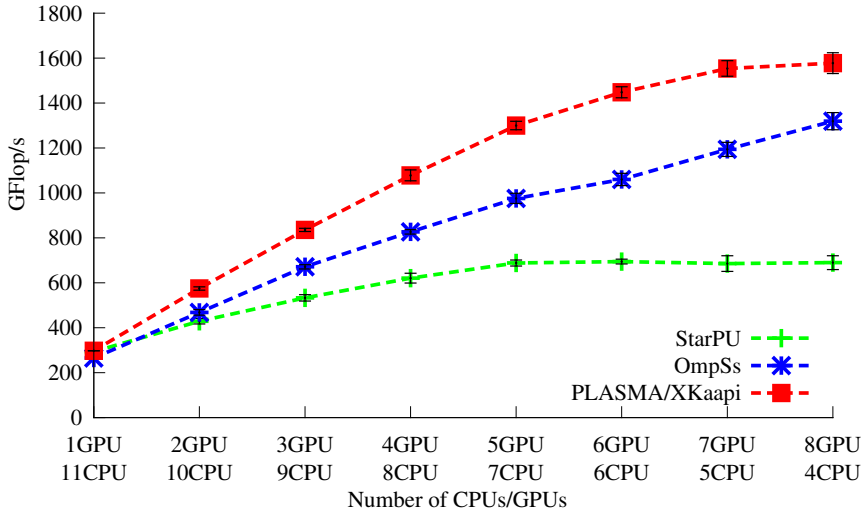


Figure C.2 – Performance results of matrix multiplication for a matrix size of 10240×10240 and block size 1024×1024.

print than StarPU on medium-size problems. However, we previously noticed allocation problems on OmpSs for input sizes bigger than a GPU memory capacity (Lima et al., 2012).

Appendix D

Publications

- Joao V. F. Lima and Nicolas Maillard. Paralelismo de Tarefas em Arquiteturas Híbridas Multi-CPU e Multi-GPU. In *XI Escola Regional de Alto Desempenho (ERAD)*, Porto Alegre, Brazil, 2011.

- Joao V. F. Lima, Thierry Gautier, Nicolas Maillard, and Vincent Danjean. Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs. In *Proc. of the 24th SBAC-PAD*, pages 75–82, New York, USA, 2012. IEEE.

- Thierry Gautier, Joao V. F. Lima, Nicolas Maillard, and Bruno Raffin. Locality-Aware Work Stealing on Multi-CPU and Multi-GPU Architectures. In *6th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*, Berlin, Germany, 2013.

- Joao V. F. Lima and Nicolas Maillard. Implementação da PLASMA para Arquiteturas Heterogêneas Multi-CPU e Multi-GPU em XKaapi. In *XIII Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD-RS 2013)*, Porto Alegre, Brazil, 2013.

- Thierry Gautier, Joao V. F. Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1299–1308, 2013. `10.1109/IPDPS.2013.66`.

- Joao V. F. Lima, François Broquedis, Thierry Gautier, and Bruno Raffin. Preliminary Experiments with XKaapi on Intel Xeon Phi Coprocessor. In *25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Porto de Galinhas, Brazil, October 2013. `10.1109/SBAC-PAD.2013.28`.

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL

UNIVERSITÉ DE
GRENOBLE