UNIVERSITE DE PARIS SUD XI

**ECOLE DOCTORALE D'INFORMATIQUE
DE PARIS SUD XI (EDIPS)**

Résumé du Manuscrit de Thèse

Défendu par

Pierre ESTÉRIE

Support Multi-Architectural : Une Approche Générique et Générative

Directeurs de Thèse : Brigitte ROZOY
Encadrant de Thèse : Joel FALCOU

Préparé au Laboratoire de Recherche en Informatique
Equipe PARSYS

Défendu le 20 Juin 2014

Jury :

Philippe CLAUSS,	*Rapporteur*	-	Université de Strasbourg (ICube)
Lawrence RAUCHWERGER,	*Rapporteur*	-	Université de Texas A&M (Parasol)
Sylvain CONCHON,	*Examinateur*	-	Université de Paris Sud XI (LRI)
Joel FALCOU,	*Examinateur*	-	Université de Paris Sud XI (LRI)
Sylvain JUBERTIE,	*Examinateur*	-	Université de Orléans (LIFO)
Brigitte ROZOY,	*Directrice*	-	University of Paris Sud XI (LRI)

Résumé

Le besoin constant de puissance de calcul a poussé les développeurs à concevoir de nouvelles architectures : les architectures parallèles. Le calcul scientifique dépend fortement des performances de ces dernières afin de fournir des résultats dans un temps optimal. Les applications scientifiques exécutées sur de tels systèmes doivent alors tirer partie des spécificités de ces nouvelles architectures pour être efficaces.

Cette thèse présente une nouvelle approche pour la conception de logiciels embarquant des optimisations relatives aux architectures : l'approche *AA-DEMRAL* (Architecture Aware DEMRAL). Cette méthodologie a pour but de simplifier le développement de bibliothèques de calcul parallèle avec un support multi-architectural grâce à une approche générique et générative.

Cette nouvelle méthodologie est ensuite intégrée dans trois bibliothèques. La première d'entre elles, Boost.Dispatch , permet de concevoir des logiciels basés sur l'approche *AA-DEMRAL* . Boost.Dispatch est une bibliothèque C++ fournissant une interface générique pour réaliser de la surcharge de fonction avisée de l'architecture sous-jacente. Ensuite nous présentons deux bibliothèques C++ implémentées en tant que langages orientés domaine : Boost.SIMD et NT^2 . Leurs conceptions mettent en œuvre la méthodologie *AA-DEMRAL* et leurs implémentations sont basées sur Boost.Dispatch . Boost.SIMD propose une interface de haut niveau pour la programmation des unités vectorielles. NT^2 se base sur une interface similaire à celle de Matlab et fournie un support pour les systèmes multi-cœurs et les unités vectorielles. Enfin, nous validons les performances de ces deux outils ainsi que la robustesse de notre nouvelle approche en présentant une série de résultats obtenus sur des applications de référence.

Mots clés : Architectures Parallèles, *DSEL* , Bibliothèques Actives, Programmation Générative, Programmation Générique, C++ .

1 Introduction

Le calcul scientifique est une composante importante des domaines de recherche depuis la naissance des calculateurs. Son utilisation a poussé les constructeurs d'architectures à toujours augmenter la puissance de calcul des machines informatiques. Les arcitectures parallèles ont pris le pas sur les machines séquentielles en fournissant une solution au besoin constant de cette puissance. La programmation de tels systèmes n'est pas triviale et le temps de développement des applications scientifiques est devenu de plus en plus important. La difficulté d'une telle tâche est due dans un premier temps à l'expertise matérielle nécessaire pour programmer de manière efficace ces systèmes et dans un second temps au manque d'outils de haut niveau pour l'utilisateur. En effet, les outils disponibles présentent une interface de bas niveau qui manque d'expressivité. En résulte un processus de programmation difficile et l'expressivité du code source diminue. De plus, la multiplicité des architectures parallèles augmente la difficulté de cette tâche quand il s'agit de produire un code portable. C'est dans ce contexte que nous présentons notre travail. Notre première contribution est de proposé une nouvelle méthode pour le développement de logiciel en accord avec l'architecture sous-jacente tout en gardant un haut niveau d'expressivité et de performance. Dans cette thèse nous présentons cette nouvelle méthode ainsi que trois outils intégrant cette dernière.

2 Un large paysage architectural et logiciel

Les constructeurs d'architectures ont toujours poussé le développement des machines dès que la technologie le permettait. La diversité des machines permet au calcul scientifique de tirer partie des spécificités de chacune d'elles. Cet aspect du paysage architectural contraint les développeurs à rester avertis des avancés technologiques et à développer l'expertise correspondante. Il devient donc difficile de développer des applications rapidement qui allient optimisations matérielles et logicielles. Ce chapitre rend compte de la diversité des machines et des solutions logicielles.

2.1 La diversité des architectures matérielles

Les systèmes modernes de calcul scientifique intègrent un grand nombre d'architectures spécifiques en partant des extensions vectorielles jusqu'aux systèmes à mémoire distribuée.

Les extensions vectorielles ou SIMD (Single Instruction Multiple

Data) ont été introduites par les constructeurs de processeurs afin d'exploiter le parallélisme de donnée présent dans les applications multimédias qui ont un besoin constant de performance. Elles permettent de manipuler des registres de grande taille capables de stocker plusieurs données et ensuite d'appliquer la même instruction sur l'ensemble du registre.

Les constructeurs fournissent alors des instructions supplémentaires pour exploiter ces extensions. Intel à présenté l'extension MMX en 1997 qui sera la première de la famille SSE. AMD a aussi intégré des extensions de la famille SSE dans ces processeurs. De SSE à AVX2 en passant par SSE4.2, de nombreuses extensions avec de nouvelles instructions sortent au fil des années. Motorola a sorti sa propre extension en 1999 appelée Altivec. En 2014, la prochaine génération du Xeon Phi d'Intel intègrera AVX-512 avec des registres de 512-bit.

Une autre solution a été présentée par les constructeurs pour continuer à augmenter la puissance de calcul de leur processeurs : les multi-cœurs. En augmentant le parallélisme au niveau des cœurs de calcul sur un même processeur, les fabricants ont continué la course à la puissance. En 2001, IBM lance le premier multi-cœur : le POWER4.

Les multi-cœurs sont vite devenus la solution de choix et de nos jours, la tendance est de toujours augmenter le nombre de coeurs. Par exemple, Intel propose un Core i7-4770K avec une fréquence de 3.9 GHz et 4 cœurs physiques.

La solution proposée par les multi-cœurs peut ne pas être suffisante lorsque les applications présentent une importante charge de calculs. En réponse à cette demande, les accélérateurs permettent de décharger le processeur d'un certain nombre de calculs. Les cartes graphiques (GPGPU) sont devenues la solution majoritaire pour les accélérateurs grâce à leur faible coût de fabrication et leur disponibilité dans la plupart des systèmes modernes. Intel lui aussi propose une solution avec le Xeon Phi qui, contrairement au GPGPU, est une architecture x86 plus conventionnelle.

Lorsque les solutions précédentes ne suffisent plus, les systèmes à mémoire distribuée proposent de mettre en parallèle plusieurs machines appelées des *noeuds* en utilisant un réseau de communication. Chaque nœud peut être composé de configurations différentes et embarquer des multi-cœurs ainsi que des GPGPU. Ce type de machine est conçu pour des applications avec de très grosses charges de calculs et d'importants volumes de données.

La diversité du paysage architectural permet de distinguer deux tendances principales.

— Systèmes de petite échelle : Multi-cœurs et extensions SIMD
— Systèmes de grande échelle : Mémoire distribuée + accélérateurs

2.2 Les outils de programmation

Le développement d'applications parallèles est lié aux outils de programmation disponibles pour les architectures introduites précédemment. Chaque solution logicielle présente différents niveaux d'expressivité et supporte différentes architectures.

Les extensions SIMD sont principalement utilisées via des appels à des fonctions C de bas niveau appelées *intrinsèques*. Elles représentent chaque instruction assembleur SIMD et sont donc plus accessibles. Mais cette interface en langage C introduit un style de programmation lourd. De plus, d'une extension à l'autre, les appels de fonctions changent et le code source de l'application doit être réécrit. Les compilateurs sont capables de générer du code SIMD à partir d'un code standard mais cette approche trouve ses limites lorsque le code source ne présente pas des opportunités claires de vectorisation. Des compilateur dédiés tels que ISPC proposent une alternative à ce problème en fournissant de nouveaux mots clés. Cette approche impact le code source qui devient alors non standard.

Du côté des multi-cœurs, les outils principaux permettent de travailler avec des fils d'exécution ou *threads* qui vont s'exécuter en parallèle sur les différents cœurs. Par exemple pThread propose à l'utilisateur de créer et de manipuler les threads lui même grâce à des fonctions de bas niveau. Cette interface fournit un style de programmation verbeux et nécessite une bonne connaissance du mécanisme des threads. OpenMP propose de s'abstraire d'une interface bas niveau en utilisant des directives de compilation qui rendent le code source plus clair. OpenMP décharge aussi l'utilisateur de l'expertise nécessaire pour la programmation avec des threads. Pour tirer partie de cette approche, le support OpenMP doit être implémenter dans le compilateur cible. Intel propose la bibliothèque TBB qui permet de manipuler des objets C++ de haut niveau et ainsi d'ajouter un niveau d'expressivité.

Les systèmes distribués sont programmables via différentes solutions comme MPI qui permet de réaliser les communications entre les nœuds de calcul. MPI est un outil puissant et portable mais qui manque d'expressivité avec une interface C de bas niveau. Des outils comme HPX ou Stapl tirent partis du langage C++ pour fournir une interface de plus haut niveau et ainsi faciliter la programmation de tels systèmes.

La programmation des architectures parallèles modernes nécessite un niveau d'expertise non négligeable car la diversité des outils existant rend cette tâche difficile. Lorsque l'utilisateur veut combiner plusieurs

niveaux de parallélisme, cette difficulté augmente en raison des différents modèles de programmation disponibles. Les librairies orientées domaine telles que BLAS, LAPACK, MAGMA ou PLASMA proposent des fonctions pour l'algèbre linéaire déjà optimisées et utilisable directement par le développeur. FFTW, OpenCV et Intel IPP sont aussi de bons exemples pour le traitement d'images et du signal. La sémantique du domaine de chaque fonction permet au développeur de s'affranchir de la gestion du parallélisme car celui-ci est enfoui à l'intérieur des fonctions. Cette approche fonctionnelle n'atteint pas un haut niveau d'expressivité car la gestion de la mémoire ou l'initialisation des bibliothèques sont laissées à l'utilisateur.

Les langages orientés domaine (DSL) fournissent un haut niveau d'abstraction et sont conçus pour un domaine dédié. Leur syntaxe embarque la sémantique du domaine et donne à l'utilisateur l'expressivité nécessaire. Matlab est un DSL orienté vers les mathématiques et la physique, il fournit un environnement complet pour l'utilisateur. Les DSL sont majoritairement compilé dynamiquement ou exécute dans des machines virtuelles ce qui leur donne un faible niveau de performance. Les DSEL sont des DSL embarqués dans un langage généraliste et permettent de fournir de meilleures performances. Blitz++ ou Eigen en sont des exemples.

La diversité des solutions logicielles est grande et souffre soit d'un manque d'expressivité soit d'un manque de performance. Dans cette thèse, notre approche a pour but de développer des outils pour le calcul scientifique avec un maximum d'expressivité tout en conservant les performances d'un code parallèle optimisé. Nous avons choisi les DSEL pour leur haut niveau d'expressivité. Le langage cible retenu est C++ pour son interopérabilité avec les outils de programmation parallèle et son mécanisme de template.

3 Une approche générique et générative des DSL

Dans ce chapitre, nous présentons l'approche choisie pour nos travaux. Le chapitre précédent rend compte de la nécessité d'un haut niveau d'expressivité pour l'outil de programmation tout en conservant un maximum de performance c'est à dire en garantissant un support pour les architectures parallèles. Le développement d'un DSEL en C++ pour le calcul scientifique est l'approche retenue.

La conception d'outils en C++ est majoritairement réalisée par l'intermédiaire de bibliothèques. L'approche courante pour développer ces

dernières est l'utilisation de patrons de conception (*i.e.* design patterns). Les composants logiciels récurrents lors du développement ont été généralisés. Un ensemble de patrons standards a été proposé pour aider les développeurs. Cette approche datant d'une vingtaine d'années trouve ses limites lorsque qu'un grand nombre de composants logiciels doivent interagir. De plus, l'abstraction proposée par les patrons de conception ne prend pas en compte la sémantique du domaine applicatif sur laquelle l'expressivité repose.

Czarnecki a proposé une nouvelle approche, la programmation générative. Cette méthode consiste à définir un modèle pour implémenter plusieurs composants logiciels. La bibliothèque standard C++ repose sur une méthode similaire et propose à l'utilisateur des composants qu'il peut agréger. La programmation générative pousse cette approche plus loin en proposant d'automatiser la composition des composants logiciels. Cette méthode peut être implémenter grâce à des bibliothèques actives. En opposition aux bibliothèques classiques, elles vont jouer un rôle actif durant la compilation pour générer un code. Elles proposent des abstractions orientées domaine grâce à des composants logiciels génériques et définissent le générateur associé pour contrôler la composition de ces mêmes composants. De telles bibliothèques sont alors implémentées sous la forme d'un DSEL.

La technique utilisée pour concevoir ces bibliothèques s'appelle la meta- programmation par template. Cette dernière fournit une technique de programmation générative dans laquelle les templates sont utilisées par le compilateur pour générer un code source temporaire. Le compilateur exécute des meta-programmes générant un code source temporaire qui sera ensuite fusionné avec le reste du code source. Grâce à son support des templates, C++ permet l'utilisation de cette technique.

En C++, la technique sur laquelle la conception d'un DSEL repose s'appelle les expressions templates. Elles sont implémentées par l'intermédiaire d'un système d'évaluation partielle. La composition récursive des types en C++ permet de construire un Arbre de Syntaxe Abstraite (*i.e.* AST : Abstract Syntax Tree) qui sera ensuite manipulé, reconstruit et évalué. Par ce mécanisme, des expressions de haut niveau peuvent être construites et évaluées après une phase de transformation de l'AST. Les expressions templates sont difficiles à maintenir et des bibliothèques telles que Boost.Proto ou PETE automatisent ce mécanisme. La possibilité de construire des expressions de haut niveau permet alors de concevoir une interface simple et intuitive pour l'utilisateur et donc de définir un DSEL au sein de C++. Mais les techniques présentées souffrent d'un manque de règles pour concevoir un DSEL.

Une bibliothèque classique manipulant N structures de données et P algorithmes nécessite l'implémentation de $N \times P$ fonctions. Czarnecki a étudié une méthodologie appelée DEMRAL (DEMRAL : Domain Engineering Method for Reusable Algorithmic Libraries). Cette technique vise à réduire l'effort nécessaire pour développer une bibliothèque en limitant la quantité de code à écrire. Grâce à l'approche DEMRAL, seulement N structures de données et P algorithmes doivent être implémentés. Mais cette méthode de conception ne prend pas en compte l'architecture matérielle sous- jacente. Ainsi pour garantir la prise en compte des potentielles optimisations architecturales disponibles, nous proposons d'étendre cette méthode pour qu'elle prennent en compte la composante matérielle. Dans une approche classique d'une bibliothèque devant supporter Q architectures, $N * P * Q$ fonctions devraient être maintenues.

Notre contribution à la méthode DEMRAL consiste à injecter des informations sur l'architecture matérielle dans le processus génératif d'un DSEL. Nous réalisons cela par l'intermédiaire d'un composant générique et génératif additionnel au processus de programmation générative. Il permet de générer des composants logiciels optimisés en fonction de l'architecture et les passe ensuite au générateur principale qui se chargera de les agréger et de les composer comme dans un processus de génération de code classique introduit précédemment. Cette nouvelle méthodologie s'appelle AADEMRAL (Architecture Aware DEMRAL). Ainsi, seulement $N + P + Q$ composants logiciels doivent être développés et maintenus.

L'ajout d'un composant génératif architectural permet de prendre en compte au plus tôt les optimisations matérielles potentielles et de les inclure dans le processus de génération de code. En étendant l'approche DEMRAL, la conception du DSEL reste scalable et générique. Nous proposons alors une bibliothèque permettant d'utiliser cette méthodologie : Boost.Dispatch .

4 La Bibliothèque Boost.Dispatch

Boost.Dispatch est une bibliothèque template C++ permettant l'implémentation de solutions logicielles basées sur la methodologie AADEMRAL introduite dans le chapitre précédent. Le principale challenge d'un tel outil est de fournir une interface générique permettant à l'utilisateur de concevoir des composants logiciels réutilisables. Boost.Dispatch est conçue autour du principe de surcharge de fonctions disponble en C++.

La bibliothèque permet d'injecter des informations sur l'architecture sous- jacente à l'appel d'une fonction et ainsi de sélectionner la version la plus optimisée de cette dernière. Boost.Dispatch réalise cette sélection grâce à une mécanisme avancé de *Tag Dispatching*. Le point d'entrée de ce mécanisme est le Concept de *Hiérarchie*. Des Hiérarchie pour les types classiques et les fonctions sont disponibles. Une Hiérarchie pour les architectures est également fournie et permet ainsi à l'utilisateur de spécialiser les appels de fonctions suivant la Hiérarchie des architectures.

Boost.Dispatch propose une serie de macros à l'utilisateur pour déclarer et définir les fonctions. L'utilisateur peut aussi introduire de nouvelles Hiérarchies dans l'outil.

Par l'intermédiaire des Hierarchies, la bibliothèque fournie une solution pour l'implémentation de logiciels conçus avec la méthodologie AADEMRAL c'est à dire une sélection d'appels de fonctions prenant en compte l'architecture matérielle. Nous proposons d'utiliser Boost.Dispatch pour le développement de deux outils de programmation parallèle : Boost.SIMD et NT^2 .

5 La Bibliothèque Boost.SIMD

Boost.SIMD est une bibliothèque C++ template implémentée par l'intermédiaire de Boost.Dispatch . Elle est donc conçue avec la méthode AADEMRAL. Son but est de simplifier la programmation des extensions SIMD en gardant un maximum d'expressivité et de performance. Pour ce faire, Boost.SIMD est un DSEL fournissant une interface de haut niveau avec la sémantique des extensions SIMD.

La bibliothèque fournit une abstraction pour manipuler les registres SIMD appelée `pack<T>`. Ce type C++ est capable de selectionner automatiquement le meilleure type de registre SIMD pour l'architecture cible en fonction du type arithmétique `T` que l'utilisateur souhaite manipuler. Les operateurs C++ classiques sont surchargés et la bibliothèque fournit un nombre conséquent de fonctions pour `pack`.

Listing 1– Utilisation de l'abstraction `pack`

```
#include <iostream>
#include <boost/simd/sdk/simd/io.hpp>
#include <boost/simd/sdk/simd/pack.hpp>
#include <boost/simd/include/functions/splat.hpp>
#include <boost/simd/include/functions/plus.hpp>
#include <boost/simd/include/functions/multiplies.hpp>

int main(int argc, const char *argv[])
{
  typedef pack<float> p_t;

  p_t res;
  p_t u(10);
  p_t r = boost::simd::splat<p_t>(11);

  res = (u + r) * 2.f;

  std::cout << res << std::endl;

  return 0;
}
```

Boost.SIMD est aussi capable de détecter des optimisations au niveau des expressions de `pack<T>` tels que les opérations arithmétiques FMA (Fused Multiply- Accumulate) et de générer le code optimisé en conséquence si l'architecture cible supporte cette opération. De plus, la bibliothèque supporte le standard C++ nativement et fournit les composants nécessaires pour être intégrable facilement dans la bibliothèque standard C++. Ainsi, Boost.SIMD comprend des allocateurs, des itérateurs SIMD ainsi qu'un support pour le Concept de Range. Boost.SIMD peut donc interagir avec les algorithmes de la bibliothèque standard. Boost.SIMD intègre également les opérations de permutation (*i.e. shuffle*) de manière générique et portable. Le listing 1 présente un exemple simple de l'utilisation de Boost.SIMD .

L'implémentation de la bibliothèque est entièrement détaillée dans ce chapitre et des mesures de performance attestent de l'efficacité de la bibliothèque. La table 1 présente les résultats de l'éxécution du noyau SAXPY écrit avec Boost.SIMD en se comparant à une version SIMD du code écrite à la main pour l'extension SSE4.2. Les résultats montre que Boost.SIMD génère le code SIMD attendu et que la bibliothèque n'ajoute pas de surcoût.

L'approche AADEMRAL fournit de bons résultats pour la conception de Boost.SIMD . L'expressivité est conservée avec une interface de haut niveau générique sans impacter la qualité du code SIMD généré.

Table 1 – SAXPY : Boost.SIMD comparée avec un code SIMD écrit à la main (*GFlop/s*)

Type	Size	Version	SSE4.2
float	2^9	Ref. SIMD	4.03
		Boost.SIMD	4.70
	2^{14}	Ref. SIMD	3.40
		Boost.SIMD	3.49
	2^{19}	Ref. SIMD	3.41
		Boost.SIMD	3.98

6 La Bibliothèque NT^2

NT^2 ajoute un niveau d'abstraction au-dessus de Boost.SIMD . Conçue elle aussi comme un DSEL, la bbliothèque NT^2 propose une interface utilisateur très proche de Matlab avec la même sémantique. Ainsi, la conversion d'un code Matlab vers un code NT^2 requiert très peu de changement. Seule la définiton des variables et le remplacement de l'opérateur `colon` doit être réalisé.

L'élément principale de l'interface de NT^2 est la classe `table`. Cette classe template se comporte exactement comme un tableau multi-dimensionnel de Matlab . Une gamme de paramètres (*settings*) est disponible pour cette classe, par exemple, la stratégie d'indexage, la forme de la `table` (triangulaire, diagonale, etc) ou encore la définition d'une taille statique pour la `table`.

Listing 2– Un exemple de code Matlab

```
A1 = (1.0:1000.0);
A2 = A1 + randn(size(A1));
rms = sqrt( sum(sqr(A1(:) - A2(:))) / numel(A1) );
```

Le listing 2 présente un code Matlab et le listing 3 montre les changements mineurs pour utiliser NT^2 .

Listing 3– Un exemple de code NT^2

```
table<double> A1 = _(1.0,1000.0);
A2 = A1 + randn(size(A1));
double rms = sqrt( sum(sqr(A1(_) - A2(_))) / numel(A1) );
```

NT^2 propose à l'utilisateur un nombre important de fonctions pour la classe `table` (500+) ainsi que les opérations d'algèbre linéaire les plus courantes. Les opérateurs classique de C++ sont eux aussi disponibles. NT^2 ajoute le support pour le multi-threading et utilise Boost.SIMD en interne pour générer du code SIMD. La figure 1 compare le support de NT^2 aux bibliothèques qui présentent une approche similaire.

Feature	Armadillo	Blaze	Eigen	MTL	uBlas	NT^2
MATLAB API conformance	✓	–	–	–	–	✓
AST optimization	✓	✓	–	–	–	✓
SSEx support	✓	✓	✓	–	–	✓
AVX support	✓	✓	–	–	–	✓
Altivec support	–	–	✓	–	–	✓
Shared memory parallelism	–	–	–	–	–	✓
BLAS/LAPACK binding	✓	✓	✓	✓	✓	✓

FIGURE 1 – Comparaison de NT^2 aux bibliothèques similaires

L'impléméntation de NT^2 repose sur BOOST.DISPATCH . Tout comme BOOST.SIMD , NT^2 est capable de détecter les optimisations orientées domaines au niveau de ses expressions et de générer le code correspondant. Le parallélisme au niveau des coeurs est géré grâce à l'utilisation de squelettes parallèles qui permettent à la bibliothèque de facilement composer différents scénarios de génération de code parallèle.

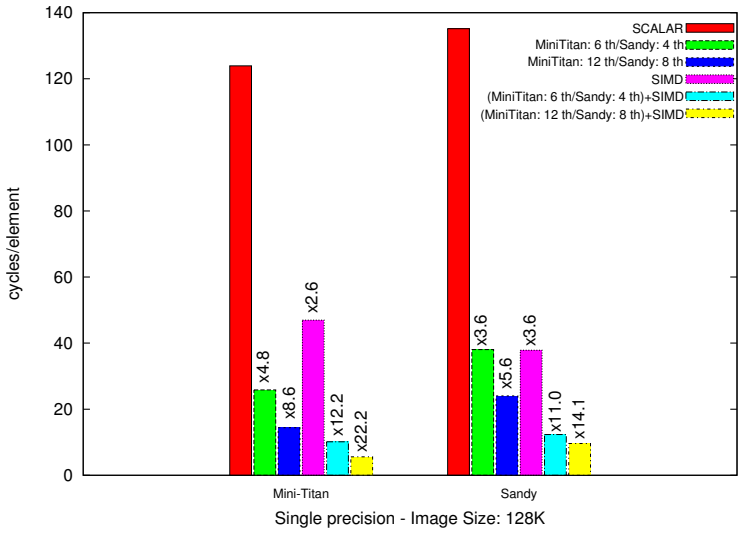

FIGURE 2 – Résultats de l'algorithme Black&Scholes en simple précision

La figure 2 présente les résultats obtenus avec NT^2 sur l'algorithme Black&Scholes utilisé en finance. On remarque que NT^2 génére un code multi-threadé performant en accord avec les accélérations attendues sur deux machines différentes. Le génération du code SIMD est toujours efficace par l'intermédiaire de BOOST.SIMD . L'algorithme Black&Scholes a une densité de calcul arithmétique assez lourde qui impacte les performances sur Mini-Titan en SIMD mais NT^2 fournit

quand même un gain non négligeable. Sur la machine Sandy, le version SIMD se comporte comme attendue avec une accélération proche de $\times 4$.

La bibliothèque NT^2 montre que l'approche AADEMRAL permet de développer un outil de calcul scientique de haut niveau avec un support multi-architectural sans sacrifier les performances.

7 Conclusion et Perspectives

7.1 Conclusion

Dans cette thèse nous avons présenté deux bibliothèques actives dont le but est de simplifier le développement des applications de calcul scientifique. Notre contribution est basée sur plusieurs approches. Après avoir étudié les avantages et inconvénients des approches existantes, nous avons proposé une nouvelle methodologie pour le développement d'outils de programmation parallèle : la méthode AADEMRAL. La bibliothèque Boost.Dispatch a été développée pour intégrer cette méthode dans la conception d'outils de calcul scientifique. Boost.Dispatch a ensuite été intégrée à Boost.SIMD et NT^2 . Ces deux bibliothèques procurent un niveau d'expressivité élevé à l'utilisateur tout en générant du code optimisé pour les architectures cibles. Ces outils ont montré de bons résultats sur des applications réelles et ont ainsi validé la methodologie AADEMRAL.

Les trois bibliothèque présentées dans cette thèse montrent qu'une approche générique et générative permet de conserver un haut niveau d'abstraction qui simplifie le support multi-architectural des outils de programmation.

7.2 Perspectives

En ce qui concerne Boost.SIMD , le support pour les extensions SIMD ARM Neon et Intel Phi est en développement. Nous étudions aussi la possibilité d'intégrer des optimisations tels que le déroulage de boucle dans Boost.SIMD . Du côté de NT^2 , les supports pour les systèmes à mémoire distribuée et les GPGPU (General Purpose Graphic Processing Unit) sont pris en considération et une approche basé sur le *multi-stage programming* est à l'étude.