UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Daniel Di Domenico

HPSM: UMA API EM LINGUAGEM C++ PARA PROGRAMAS COM LAÇOS PARALELOS COM SUPORTE A MULTI-CPUS E MULTI-GPUS

Santa Maria, RS
2016

Daniel Di Domenico

HPSM: UMA API EM LINGUAGEM C++ PARA PROGRAMAS COM LAÇOS PARALELOS COM SUPORTE A MULTI-CPUS E MULTI-GPUS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Área de Concentração em Ciência da Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Computação**.

ORIENTADOR: Prof. João Vicente Ferreira Lima

Santa Maria, RS
2016

Ficha catalográfica elaborada através do Programa de Geração Automática
da Biblioteca Central da UFSM, com os dados fornecidos pelo(a) autor(a).

Di Domenico, Daniel
HPSM: uma API em linguagem C++ para programas com laços paralelos com suporte a multi-CPUs e multi-GPUs / Daniel Di Domenico.- 2016.
92 p.; 30 cm

Orientador: João Vicente Ferreira Lima
Dissertação (mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2016

1. API C++ 2. Programação paralela 3. Laços paralelos 4. Computação heterogênea 5. GPU I. Lima, João Vicente Ferreira II. Título.

End. Eletr.: ddomenico@inf.ufsm.br

Daniel Di Domenico

HPSM: UMA API EM LINGUAGEM C++ PARA PROGRAMAS COM LAÇOS PARALELOS COM SUPORTE A MULTI-CPUS E MULTI-GPUS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Área de Concentração em Ciência da Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Computação**.

Aprovado em 21 de dezembro de 2016:

João Vicente Ferreira Lima, Dr. (UFSM)
(Presidente/Orientador)

Benhur de Oliveira Stein, Dr. (UFSM)

Claudio Schepke, Dr. (UNIPAMPA)

Santa Maria, RS
2016

DEDICATÓRIA

À minha amada esposa Cristina, por sempre acreditar em mim e incentivar-me durante esta caminhada. Ao meu pai Jaime e minha mãe Marlene, pela sabedoria de sempre estimular os filhos a estudar desde quando eram crianças. Por fim, aos jogadores, delegação e profissionais de imprensa que foram vítimas da tragédia com o avião da Chapecoense na Colômbia, pois eles foram responsáveis por muitos momentos de orgulho e alegria durante estes dois anos.

AGRADECIMENTOS

Primeiramente, agradeço a Deus por permitir e dar forças para que tudo o que fiz até este momento fosse possível.

Também gostaria de agradecer à minha esposa, meu pai, minha mãe e minha irmã Gabrielle. A família é a base de tudo, a razão da motivação diária para alcançar os objetivos. Muito obrigado por tudo que fazem por mim a cada dia.

Ao meu orientador João Vicente Ferreira Lima, pelos ensinamentos, oportunidades e principalmente pela paciência que teve comigo durante esta jornada, estando sempre disposto a ajudar-me em tudo que fosse necessário. Seu conhecimento foi muito importante, não só para o desenvolvimento do trabalho, mas também para a sequência da minha carreira.

Aos professores Benhur, Claudio e Andrea que fizeram parte das bancas do seminário de andamento e da defesa, contribuindo muito para a finalização deste projeto.

À Universidade Federal da Fronteira Sul, por ter concedido a licença que permitiu que este estudo pudesse ser concretizado.

Por fim, a todos os meus colegas e amigos que de alguma forma ajudaram na elaboração e conclusão deste trabalho, seja indicando o melhor caminho para ingressar no programa, seja ajudando a solucionar problemas, seja oferecendo hospedagem ou simplesmente dizendo palavras de incentivo.

Procuramos uma palavra para agradecer tanto carinho e encontramos varias: thank you, gracias, danke, merci, grazie, köszönöm, salamat... Obrigado!

(Associação Chapecoense de Futebol)

RESUMO

HPSM: UMA API EM LINGUAGEM C++ PARA PROGRAMAS COM LAÇOS PARALELOS COM SUPORTE A MULTI-CPUS E MULTI-GPUS

AUTOR: Daniel Di Domenico
ORIENTADOR: João Vicente Ferreira Lima

Arquiteturas paralelas são consideradas ubíquas atualmente. No entanto, o mesmo termo não pode ser aplicado aos programas paralelos, pois existe uma complexidade maior para codificá-los em relação aos programas convencionais. Este fato é agravado quando a programação envolve também aceleradores, como GPUs, que demandam o uso de ferramentas com recursos muito específicos. Neste cenário, apesar de existirem modelos de programação que facilitam a codificação de aplicações paralelas para explorar aceleradores, desconhece-se a existência de APIs que permitam a construção de programas com laços paralelos que possam ser processados simultaneamente em múltiplas CPUs e múltiplas GPUs. Este trabalho apresenta uma API C++ de alto nível, denominada HPSM, visando facilitar e tornar mais eficiente a codificação de programas paralelos voltados a explorar arquiteturas com multi-CPU e multi-GPU. Seguindo esta ideia, deseja-se ganhar desempenho através da soma dos recursos. A HPSM é baseada em laços e reduções paralelas implementadas por meio de três diferentes *back-ends* paralelos, sendo Serial, OpenMP e StarPU. A hipótese deste estudo é que aplicações científicas podem valer-se do processamento heterogêneo em multi-CPU e multi-GPU para alcançar um desempenho superior em relação ao uso de apenas aceleradores. Comparações com outras interfaces de programação paralela demonstraram que o uso da HPSM pode reduzir em mais de 50% o tamanho de um programa multi-CPU e multi-GPU. O uso da nova API pode trazer impacto no desempenho do programa, sendo que experimentos demonstraram que seu sobrecusto é variável de acordo com a aplicação, chegando até 16,4%. Os resultados experimentais confirmaram a hipótese, pois as aplicações N-Body, Hotspot e CFD, além de alcançarem ganhos ao utilizar somente CPUs e somente GPUs, também superaram o desempenho obtido por somente aceleradores (GPUs) através da combinação de multi-CPU e multi-GPU.

Palavras-chave: API C++. Programação paralela. Laços paralelos. Computação heterogênea. GPU.

ABSTRACT

HPSM: A C++ API FOR PARALLEL LOOPS PROGRAMS SUPPORTING MULTI-CPUS AND MULTI-GPUS

AUTHOR: Daniel Di Domenico
ADVISOR: João Vicente Ferreira Lima

Parallel architectures has been ubiquitous for some time now. However, the word ubiquitous can't be applied to parallel programs, because there is a greater complexity to code them comparing to ordinary programs. This fact is aggravated when the programming also involves accelerators, like GPUs, which demand the use of tools with scpecific resources. Considering this setting, there are programming models that make easier the codification of parallel applications to explore accelerators, nevertheless, we don't know APIs that allow implementing programs with parallel loops that can be processed simultaneously by multiple CPUs and multiple GPUs. This works presents a high-level C++ API called HPSM aiming to make easier and more efficient the codification of parallel programs intended to explore multi-CPU and multi-GPU architectures. Following this idea, the desire is to improve performance through the sum of resources. HPSM uses parallel loops and reductions implemented by three parallel *back-ends*, being Serial, OpenMP and StarPU. Our hypothesis estimates that scientific applications can explore heterogeneous processing in multi-CPU and multi-GPU to achieve a better performance than exploring just accelerators. Comparisons with other parallel programming interfaces demonstrated that HPSM can reduce a multi-CPU and multi-GPU code in more than 50%. The use of the new API can introduce impact to program performance, where experiments showed a variable overhead for each application, that can achieve a maximum value of 16,4%. The experimental results confirmed the hypothesis, because the N-Body, Hotspot e CFD applications achieved gains using just CPUs and just GPUs, as well as overcame the performance achieved by just accelerators (GPUs) through the combination of multi-CPU and multi-GPU.

Keywords: C++ API. Parallel programming. Parallel loops. Heterogenous Computing. GPU.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CFD	*Computational Fluid Dynamics*
CPU	*Central Processing Unit*
CLOC	*Commented Lines of Code*
CUDA	*Compute Unified Device Architecture*
DMDA	*Deque Model Data Aware*
DSI	*Delivered Source Instructions*
ES	*Executable Statements*
GPGPU	*General Purpose Graphics Processing Unit*
GPU	*Graphics Processing Unit*
HEFT	*Heterogenous Earliest First Time*
HPC	*High Performance Computing*
HPSM	*High Performance* Santa Maria
LOC	*Lines of Code*
MPI	*Message-Passing Interface*
NCLOC	*Noncommented Lines of Code*
NUMA	*Non-uniform Memory Access*
OpenMP	*Open Multi-Processing*
PAD	Processamento de Alto Desempenho
PHEFT	*Parallel Heterogenous Earliest First Time*
UMA	*Uniform Memory Access*
UVM	*Unified Virtual Memory*

SUMÁRIO

1 INTRODUÇÃO

O processamento de alto desempenho (PAD), do inglês *High Performance Computing* (HPC), é caracterizado por empregar computadores de alta potência e velocidade (os super-computadores) na resolução de problemas computacionais que demandam esta capacidade (NIELSEN, 2016), como aplicações científicas, caracterizadas por possuírem grandes volumes de dados e processamento. A partir da alteração na forma de utilização dos novos transistores trazidos pela Lei de Moore[1], que deixaram de ser empregados no aumento da frequência dos processadores, o paralelismo passou a ser um importante caminho para alcançar o PAD. Atualmente, arquiteturas paralelas são consideradas ubíquas, sendo difícil encontrar computadores que não utilizem dispositivos *multicore* (PACHECO, 2011). No entanto, o termo ubíquo não pode ser aplicado para os programas paralelos. Uma das razões disso é que a sua codificação possui um grau de dificuldade maior em relação à codificação de aplicações convencionais. Apesar de existirem algumas ferramentas que auxiliam na construção dos programas paralelos, há muitas características específicas em cada uma delas, tornando a portabilidade dos códigos e do desempenho uma tarefa desafiadora, principalmente quando envolve CPUs (*Central Processing Units*) e aceleradores, como por exemplo, GPUs (*Graphics Processing Units*) (EDWARDS et al., 2012).

Visando diminuir a complexidade ao desenvolver aplicações paralelas principalmente para explorar GPUs, estudos recentes propuseram novas interfaces e modelos de programação com suporte a laços paralelos e tarefas assíncronas. Alguns deles empregaram a linguagem C++, como o Kokkos (EDWARDS et al., 2012), que possibilita codificar um programa onde define-se o paralelismo por meio de laços que serão executados exclusivamente por CPUs ou por uma GPU. Outra interface semelhante ao Kokkos é o C++ AMP (GREGORY; MILLER, 2012), mas ela visa a execução dos laços paralelos da aplicação somente em um acelerador. Além disso, existem ferramentas baseadas em tarefas que possuem a capacidade de explorar o processamento heterogêneo em multi-CPU e multi-GPU, como a StarPU (HUGO et al., 2013) e o OmpSs (DURAN et al., 2011). Neste cenário, desconhece-se a existência de APIs que permitam a construção de programas com laços paralelos que possam ser processados simultaneamente em múltiplas CPUs e múltiplas GPUs.

[1]Lei de Moore: definida empiricamente por Gordon Moore em 1965 e válida já a mais de 40 anos, menciona que a quantidade de transistores de um processador típico dobra a cada período de 18 a 24 meses (RÜNGER; RAUBER, 2013).

1.1 OBJETIVO

O objetivo principal deste trabalho é implementar uma API C++ de alto nível, denominada HPSM, visando facilitar e tornar mais eficiente a codificação de programas paralelos voltados a explorar arquiteturas com multi-CPU e multi-GPU por meio de diferentes *back-ends* paralelos.

Os objetivos específicos são:

- Implementar um modelo e interface de programação que permita expressar o paralelismo para CPUs e GPUs através de laços e reduções paralelas;
- Aplicar o modelo de programação desenvolvido na paralelização de mini-aplicações científicas;
- Comparar o esforço necessário para o desenvolvimento de uma aplicação paralela com o modelo proposto a outras ferramentas de programação;
- Validar o modelo proposto por meio de experimentos usando as mini-aplicações científicas, comparando o desempenho entre os diferentes *back-ends* que a HPSM suporta.

1.2 CONTRIBUIÇÕES E HIPÓTESE

As principais contribuições deste trabalho são:

- Um modelo e interface de programação de alto nível em C++ com suporte ao processamento heterogêneo (CPUs+GPUs), onde o paralelismo é expressado através de laços e reduções paralelas;
- Um modelo de programação que permite a portabilidade do código, visto que ele poderá ser executado por meio de diferentes bibliotecas de programação paralela (*back-ends*) sem que o programador utilize recursos específicos de cada uma delas, pois o *back-end* é definido em tempo de compilação;
- Implementações de mini-aplicações científicas com a interface HPSM;
- Resultados e análise de experimentos realizados com CPUs, GPUs e CPUs+GPUs.

A hipótese levantada e que foi utilizada como base para os experimentos realizados estima que aplicações científicas podem valer-se do processamento heterogêneo em multi-CPU e multi-GPU para alcançar um desempenho superior em relação ao uso de apenas aceleradores, neste caso, GPUs.

1.3 ORGANIZAÇÃO DO TEXTO

Este trabalho é composto por mais quatro capítulos, sendo:

- **Capítulo 2**: apresenta conceitos inerentes à programação de alto desempenho, focando principalmente em aceleradores. Detalha as arquiteturas utilizadas para PAD e algumas ferramentas de programação paralela, além de abordar os trabalhos relacionados. Inicia na página 15;
- **Capítulo 3**: descreve a HPSM, demonstrando suas funcionalidades e a modelagem de sua estrutura. Na sequência, efetua um paralelo da HPSM com os trabalhos relacionados. Por fim, realiza uma comparação de esforço de codificação de uma aplicação paralela através da API e de outras interfaces também utilizadas para esta finalidade. Inicia na página 31;
- **Capítulo 4**: aborda os resultados experimentais obtidos através da execução das mini-aplicações científicas N-Body, Hotspot e CFD implementadas por meio da nova API. Inicia na página 51;
- **Capítulo 5**: apresenta as conclusões e os trabalhos a serem desenvolvidos no futuro. Inicia na página 77;

2 PROGRAMAÇÃO DE ALTO DESEMPENHO COM ACELERADORES

Este capítulo apresenta os conceitos inerentes a programação de alto desempenho, focando principalmente em aceleradores. Para isso, é realizada inicialmente uma descrição das arquiteturas que a programação de alto desempenho visa explorar (seção 2.1), sendo memória distribuída, memória compartilhada e aceleradores. Na sequência, a seção 2.2 destaca as ferramentas mais difundidas de programação paralela para as arquiteturas previamente abordadas, bem como as técnicas que podem ser utilizadas para representar o paralelismo. Seguindo na linha da programação, a seção 2.3 detalhada as ferramentas empregadas na codificação de aplicações multi-CPU e multi-GPU, visto que as mesmas integram o objetivo deste trabalho que é o desenvolvimento de uma API para a execução de laços e reduções paralelas simultaneamente em CPUs e GPUs. Por fim, o estado da arte é apresentado, onde a seção 2.4 enumera os trabalhos relacionados na área de alto desempenho que envolvem ferramentas com suporte ao processamento heterogêneo (CPU+GPU) e APIs C++ de alto nível para o processamento paralelo.

Alguns dos tópicos abordados neste capítulo possuem exemplos de utilização demonstrados através de códigos fontes de programas. Optou-se por adicionar estes exemplos apenas às ferramentas que foram empregadas no desenvolvimento da HPSM visando manter o foco nos principais temas relacionados ao trabalho.

2.1 ARQUITETURAS DE ALTO DESEMPENHO

Os computadores podem ser organizados por meio de diversos modelos de arquiteturas. Estes modelos influenciam na forma como as aplicações são codificadas, não sendo diferente para programas voltados ao alto desempenho.

Esta seção descreve três tipos de arquiteturas que são utilizadas na computação de alto desempenho. Primeiramente serão abordadas as arquiteturas compostas por CPUs (*Central Processing Units*), como a de computadores com memória distribuída e a de computadores com memória compartilhada. Na sequência, será relatada a arquitetura de aceleradores, destacando-se a utilizada no escopo deste trabalho, a GPU (*Graphics Processing Unit*).

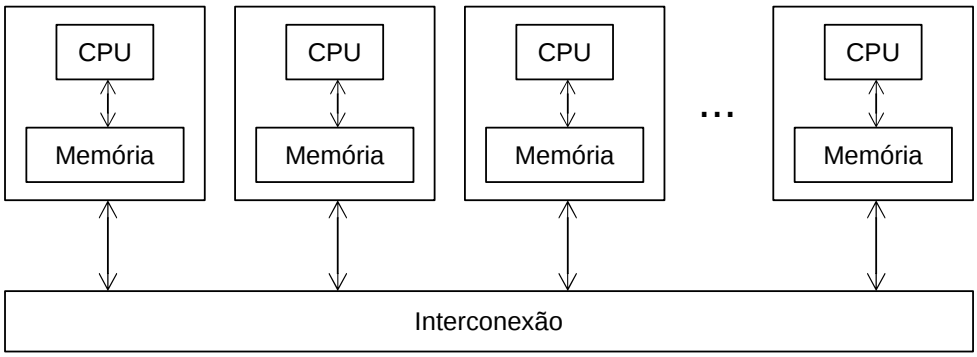
2.1.1 Computadores com memória distribuída

Computadores com memória distribuída são compostos por unidades de processamento independentes denominados nós, sendo estes interconectados por uma rede de dados. Cada nó possui um processador e memória local privada. A interação entre eles normalmente ocorre

através da troca de mensagens fazendo uso da interconexão de rede, o que torna o sistema dependente dela (RÜNGER; RAUBER, 2013).

Na Figura 2.1 é apresentada uma arquitetura de memória distribuída. Nela é possível perceber os nós com acesso apenas a sua memória privada (ou local), bem como a interconexão de rede que permite a comunicação entre eles. Como exemplo de sistema distribuído, destacam-se os *clusters*, que são definidos por Rünger e Rauber (2013) como computadores completos com uma conexão dedicada de rede. Ainda segundo os autores, outra característica de um *cluster* é que ele pode ser endereçado e programado como uma única unidade.

Figura 2.1 – Modelo de arquitetura com memória distribuída.



Fonte: Adaptado de Pacheco (2011).

2.1.2 Computadores com memória compartilhada

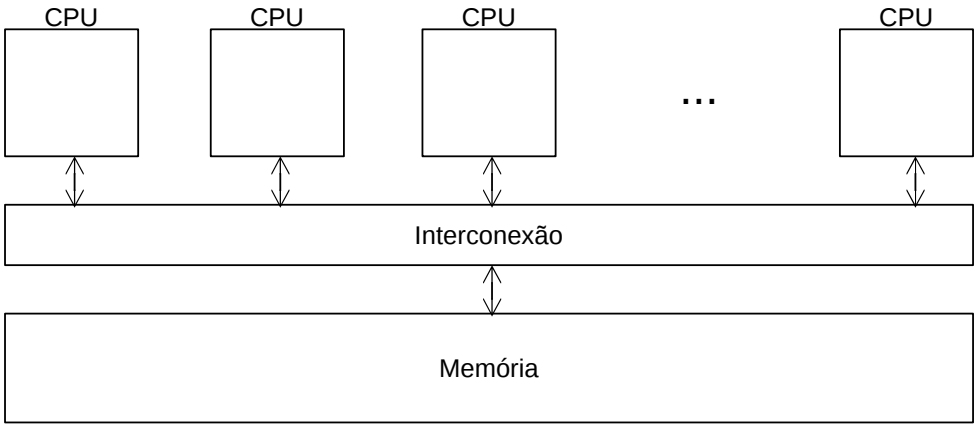
Computadores com memória compartilhada são formados por diversos processadores ou núcleos (*cores*), uma memória compartilhada física (ou memória global) e uma interconexão que conecta os processadores com esta memória. A comunicação entre os processadores é realizada por meio da leitura e escrita de variáveis compartilhadas na memória global. Um exemplo de arquitetura de memória compartilhada são os processadores com mais de um núcleo (*multicore*), que acessam a mesma memória principal de forma simétrica (RÜNGER; RAUBER, 2013). De acordo com Chapman, Jost e Pas (2007), máquinas que possuem essa característica de acesso a memória são denominadas UMA (*Uniform Memory Access*).

Existem máquinas com memória compartilhada que não possuem um acesso uniforme à memória. Normalmente, estes computadores são grandes e possuem distâncias diferentes entre os processadores e a memória, o que implica em diferentes tempos de acesso a ela. Este tipo de máquina é denominada NUMA (*Non-uniform Memory Access*) (CHAPMAN; JOST; PAS, 2007).

Na Figura 2.2 pode-se visualizar uma representação de arquitetura compartilhada, onde diversas CPUs acessam a mesma memória através de uma interconexão. O uso da memória global facilita a comunicação entre os processadores, visto que o processo é realizado através de variáveis compartilhadas. Outra vantagem está no fato de não haver replicação de dados em

múltiplas memórias, situação que ocorre em computadores com memória distribuída (RÜNGER; RAUBER, 2013).

Figura 2.2 – Modelo de arquitetura com memória compartilhada.



Fonte: Adaptado de Pacheco (2011).

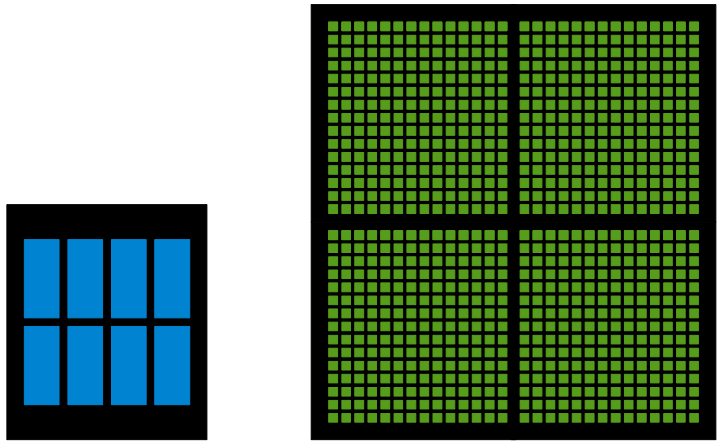
2.1.3 Aceleradores e GPUs

Aceleradores são arquiteturas compostas por dezenas ou centenas de núcleos de processamento distintos projetados para atingir alto desempenho na execução de códigos paralelos. Esta é a principal diferença entre aceleradores e as CPUs com mais de um *core*, que efetivamente só replicam núcleos para processamento sequencial (FENG; MANOCHA, 2007). Fazendo-se uma analogia com CPUs *multicore*, os aceleradores podem ser caracterizados como arquiteturas *manycore*. Outra propriedade dos aceleradores é que eles possuem um espaço de memória próprio separado da memória principal do computador.

As GPUs, que originalmente eram arquiteturas empregadas apenas no processamento gráfico, atualmente são classificadas como aceleradores pois vem sendo utilizadas como processador de uso geral (ou GPGPU, do termo em inglês *General Purpose Graphics Processing Unit*). Entre suas características principais estão a capacidade de processamento simultâneo usando diversas *threads*, além de uma alta taxa para o acesso a memória (largura de banda), podendo chegar a ser dez vezes maior que a das CPUs (FENG; MANOCHA, 2007). Ao comparar-se o objetivo de uma CPU e de uma GPU tem-se uma diferença. Uma CPU objetiva reduzir a latência (tempo de execução) do processamento de uma instrução. Já uma GPU visa aumentar a vazão (*throughput*) deste processamento, ou seja, executar um número grande de instruções ao mesmo tempo. Isso ocorre pois seus *cores* possuem uma capacidade menor em relação aos de CPUs, o que resulta em uma latência maior para a execução de uma única instrução. No entanto, esta maior latência é compensada pela maior vazão. De acordo com este cenário, Kirk e Hwu (2013) afirmam que para alcançar um desempenho satisfatório, as GPUs requerem programas projetados para explorar muitas *threads*. Caso contrário, o processamento da aplicação será mais rápido em uma CPU.

A Figura 2.3 apresenta as arquiteturas de uma CPU e de uma GPU. Em azul e verde estão destacadas as suas unidades de processamento (*cores*). A partir dela fica implícita a diferença de objetivo destacada no parágrafo anterior, onde as GPUs e os aceleradores, por serem capazes de processar muitas instruções paralelamente, podem alcançar um desempenho superior ao das CPUs.

Figura 2.3 – Comparação das arquiteturas CPU e GPU.



Fonte: Adaptado de NVIDIA.

Por possuírem uma arquitetura distinta em relação as CPUs, a programação para explorar uma GPU precisa ser realizada utilizando linguagens específicas, como CUDA e OpenCL. Estas linguagens também possuem recursos para controlar as transferências de dados entre a memória principal do programa e memória da GPU, visto que o acelerador somente acessa as informações presentes na sua própria memória. As transferências são inclusive um fator crítico que precisa ser considerado ao utilizar-se os aceleradores. Apesar de robustas, as linguagens CUDA e OpenCL não permitem explorar de forma simultânea as GPUs e CPUs de uma plataforma. Este tipo de computação pode ser obtido através de ferramentas específicas, como as apresentadas na seção 2.3.

Outro acelerador que pode ser destacado é o coprocessador Intel Xeon Phi. Ele possui *cores* baseados na arquitetura *x86*, fato que facilita a codificação de programas destinados a explorá-lo, que podem ser desenvolvidos com ferramentas como OpenMP (JEFFERS; REINDERS, 2013). Além disso, o Phi permite o uso de instruções 64-bit, bem como extensões para instruções vetoriais. A nível de *hardware*, o Xeon Phi é compostos por dezenas *cores* (até 72 dependendo do modelo) que podem suportar até 4 quatro *threads* lógicas, além de unidades vetoriais de 512 bits e uma memória interna GDDR5 (até 16GB). A exemplo das GPUs, sua conexão se dá por uma placa PCIExpress (MISRA et al., 2013), interface que é utilizada para as transferências de códigos e dados para o acelerador.

2.2 PROGRAMAÇÃO PARALELA

Esta seção irá abordar a programação das aplicações paralelas destinadas à alcançar o alto desempenho. Conforme já mencionado, cada tipo de arquitetura requer uma ferramenta especializada para explorá-la. Além disso, também podem ser utilizadas diferentes técnicas para representar o paralelismo. Os próximos tópicos tratarão destes dois temas, iniciando pela descrição das técnicas que podem ser utilizadas para expressar o paralelismo, contemplando-se laços paralelos e tarefas assíncronas. Na sequência, são relatadas as ferramentas de programação clássicas, sendo MPI para memória distribuída, OpenMP para memória compartilhada e CUDA para aceleradores (GPUs).

2.2.1 Técnicas de representação do paralelismo

As técnicas de representação do paralelismo são utilizadas para definir como ele será retratado no código de um programa. Elas influenciam em fatores como o grão do trabalho e as sincronizações entre os recursos e a memória da aplicação paralela. Entre estas técnicas, o presente tópico abordará em detalhes duas: laços paralelos e tarefas assíncronas.

2.2.1.1 Laços paralelos

A programação paralela por meio de laços é utilizada por várias aplicações. Por isso, esta funcionalidade existe em diversas linguagens e ferramentas, como por exemplo, OpenMP e Cilk Plus. Quando um laço é paralelizado, o seu número de iterações é dividido entre as *threads* disponíveis. Assim, uma parte do trabalho existente no laço é setada para cada *thread*, sendo elas executas simultaneamente (AYGUADÉ et al., 2008).

O processamento de um laço paralelo é normalmente realizado através do modelo *fork-join*. De acordo com Pacheco (2011), ele consiste na existência de uma *thread* mestre que executa o programa de forma sequencial. Ao alcançar o laço paralelo, ocorre o processo de *fork*, onde todas as *threads* disponíveis são ativadas para a execução. Ao fim do laço, ocorre a etapa do *join*, onde a execução retorna novamente para a *thread* mestre e o programa volta ao processamento sequencial.

2.2.1.2 Tarefas assíncronas

As tarefas assíncronas são utilizadas para expressar paralelismo de grão fino. Elas são aplicáveis na chamada de funções concorrentes que não possuem dependências entre elas. Além disso, podem ser empregadas na criação de vários níveis de paralelismo em um programa, onde

uma rotina que está sendo executada por uma tarefa efetua uma chamada que origina uma nova tarefa. Assim, esta nova tarefa é um filho da primeira.

Conceitualmente, as tarefas são blocos estruturados executados de modo sequencial e que possuem uma memória privada válida durante o tempo necessário para o seu processamento. Elas não ficam associadas diretamente a uma *thread* do ambiente, pois o sistema de *runtime* que está executando a aplicação irá escaloná-las para o processamento de acordo com a disponibilidade dos recursos. Neste sentido, após lançar uma tarefa o programa não espera ela ser concluída para continuar a sua execução, ou seja, ele prossegue com seu fluxo ainda que a mesma não tenha sido processada. Caso seja necessário, a sincronização do programa com as tarefas previamente lançadas precisa ser efetuada de forma explícita (AYGUADÉ et al., 2008).

As tarefas podem ser utilizadas para paralelizar diversos problemas computacionais que não poderiam ser implementados com laços paralelos. Por isso, várias linguagens e ferramentas possuem suporte a esta técnica de paralelismo, como a Cilk e o OpenMP (a partir da versão 3.0).

2.2.2 Ferramentas clássicas de programação paralela

Existem ferramentas de programação paralela que podem ser denominadas clássicas, ou seja, elas são as mais utilizadas quando propõe-se desenvolver uma aplicação para alguma arquitetura específica. Para os computadores com memória distribuída, está ferramenta é a MPI (*Message-Passing Interface*). Já para os computadores com memória compartilhada, destaca-se a ferramenta OpenMP (*Open Multi-Processing*). Cabe ressaltar que as duas ferramentas são padrões que possuem uma interface de programação. Por ser uma arquitetura recente no contexto de alto desempenho, a programação para GPUs não possui uma ferramenta amplamente consolidada. No entanto, por adequar-se melhor ao contexto deste estudo, nesta seção será detalhada a ferramenta CUDA (*Compute Unified Device Architecture*).

2.2.2.1 MPI

MPI[1] é um padrão para o desenvolvimento de aplicações destinadas a arquiteturas de memória distribuída, sendo que sua interface de programação define uma biblioteca que implementa o modelo de passagem de mensagens. Ela possui funcionalidades que permitem a codificação de grandes e complexos algoritmos. Ao utilizá-la, um ou mais processos comunicam-se enviando e recebendo mensagens. Por padrão, esta comunicação é ponto a ponto, isto é, um processo deve explicitamente enviar uma mensagem, enquanto outro processo deve explicitamente recebê-la (DONGARRA et al., 2003).

De acordo com Pacheco (2011), a codificação de um programa utilizando o modelo

[1] MPI: <http://www.mpi-forum.org/>

de troca de mensagens requer perícia do programador, pois apesar de versátil e poderoso (é utilizado pelas aplicações que são executadas nos computadores mais potentes do mundo), ele é baixo nível. Entre os fatores citados pelo autor que contribuem para isso está a necessidade de se reescrever a maior parte de um código sequencial para torná-lo paralelo, bem como ser preciso replicar estruturas de dados dos programas para cada processo ou mesmo dividí-las explicitamente entre eles. Ainda segundo Pacheco (2011), troca de mensagens é conhecido como o *Assembly* das linguagens paralelas.

Além da passagem de mensagens ponto a ponto, o MPI também suporta comunicações coletivas através de funções de *broadcast* que permitem enviar mensagens para todos os processos do programa. Nesta categoria também enquadram-se as reduções, onde um processo recebe dados acumulados de execuções realizadas por vários outros. Outra funcionalidade do MPI são as comunicações unilaterais (*one-sided*), em que apenas um processo realiza o procedimento de transferência de informações. Isto reduz o custo da comunicação, pois elimina-se a sincronização necessária entre mensagens trocadas por dois processos (PACHECO, 2011). Apesar de ser um modelo portável que pode comunicar diferentes tipos de computadores, a construção de programas com o MPI requer o suporte de um compilador específico, como por exemplo o MPICC.

2.2.2.2 OpenMP

O OpenMP[2] é considerado o padrão *de facto* para o desenvolvimento de algoritmos paralelos visando explorar arquiteturas de memória compartilhada (ADCOCK et al., 2013; DURAN et al., 2009; STOTZER, 2014; LIAO et al., 2007). A principal vantagem do OpenMP é uso do paralelismo incremental, ou seja, ele permite estender um programa sequencial (escrito em linguagem C, C++ ou Fortran) para a execução na forma de um programa paralelo. Isto é alcançado por meio das diretivas de compilação, bibliotecas e variáveis de ambiente disponibilizadas pela sua API (RÜNGER; RAUBER, 2013). Esta abordagem é um facilitador para o seu uso, visto que outras ferramentes de memória compartilhada trabalham com o conceito de paralelização tudo ou nada (CHAPMAN; JOST; PAS, 2007).

Uma aplicação OpenMP é implementada utilizando uma combinação de diretivas de compilação, também chamadas de `pragmas`. São estas diretivas que indicam ao compilador para criar *threads*, executar operações de sincronização e gerenciar o modo de acesso à memória compartilhada. Deste maneira, para construir um programa OpenMP é necessário um compilador especializado que interprete as referidas diretivas (DONGARRA et al., 2003). Caso utilize-se um compilador que não compreenda as diretivas OpenMP ou elas sejam removidas do código, o programa irá comportar-se como uma aplicação sequencial.

O OpenMP é principalmente utilizado para paralelizar laços. Quando um laço é mar-

[2]OpenMP: <http://openmp.org/wp>

cado com um `pragma`, ele torna-se uma estrutura paralela e seu número de iterações é dividido entre todas as *threads* criadas, que por consequência, são executadas simultaneamente (MCCOOL; REINDERS; ROBISON, 2012). A Figura 2.4 mostra um exemplo de laço paralelo com OpenMP para a solução de um problema AXPY. Além dos laços paralelos, o OpenMP ainda possui alguns outros recursos interessantes, como o de tarefas assíncronas, laços com reduções, bem como barreiras para sincronização explícita (CHAPMAN; JOST; PAS, 2007).

Figura 2.4 – Exemplo de um programa OpenMP - problema AXPY.

```
1  #pragma omp parallel for
2  for(int i=0; i<size; i++)
3    y[i] = x[i] * a + y[i];
```

Fonte: Próprio autor.

2.2.2.3 CUDA

A CUDA[3] (*Compute Unified Device Architecture*) é uma plataforma de computação paralela e um modelo de programação voltado a explorar GPUs NVIDIA, permitindo a programação destinada a ambientes CPU-GPU. Ela é considerada complexa, exigindo que os detalhes da computação sejam todos abordados de forma explícita pelo programador, o que pode tornar o desenvolvimento de um programa uma tarefa tediosa e difícil (BUENO et al., 2013).

De acordo com Kirk e Hwu (2013), uma aplicação CUDA é um programa C/C++ convencional que possui seções destinadas ao processamento em dispositivos GPUs. Estas seções são denominadas *kernel* e são executadas de maneira paralela de acordo com parâmetros especificados pelo desenvolvedor. Para compilar um código CUDA, faz-se necessário um compilador específico. A NVIDIA possui uma distribuição do mesmo denominado NVCC.

A Figura 2.5 apresenta um programa CUDA para solucionar o problema AXPY. O *kernel* CUDA está especificado entre as linhas 3 e 7, sendo sua chamada realizada na linha 25. Entre as linhas 19 e 22 são executadas as alocações e cópias dos dados para a GPU, sendo que após o processamento do *kernel*, eles são copiados novamente para a memória principal através do comando da linha 28. Por fim, as linhas 29 e 30 limpam a memória alocada na GPU.

2.3 FERRAMENTAS MULTI-CPU E MULTI-GPU

As GPUs proporcionam maior capacidade de paralelismo e desempenho em comparação às CPUs em virtude de sua arquitetura possuir recursos para processamento de diversas *threads* simultaneamente. Apesar disso, a utilização de ferramentas como a CUDA resulta em

[3]CUDA: <http://www.nvidia.com/object/cuda_home_new.html>

Figura 2.5 – Exemplo de um programa CUDA - problema AXPY.

```
1 #include "utils.hpp"
2
3 __global__ void axpy(int n, float a, float* x, float* y) {
4   int i = blockIdx.x * blockDim.x + threadIdx.x;
5   if(i < n)
6     y[i] = x[i] * a + y[i];
7 }
8
9 int main(int argc, char **argv) {
10  int size = 16384;
11  float *x, *y, *d_x, *d_y, a = 3.41f;
12  x = new float[size];
13  y = new float[size];
14
15  fillArray(x, size); /* Chamadas definidas em utils.hpp */
16  fillArray(y, size);
17
18  /* Alocações e cópias para a GPU */
19  cudaMalloc((void**) &d_x, size * sizeof(float));
20  cudaMalloc((void**) &d_y, size * sizeof(float));
21  cudaMemcpy(d_x, x, size * sizeof(float), cudaMemcpyHostToDevice);
22  cudaMemcpy(d_y, y, size * sizeof(float), cudaMemcpyHostToDevice);
23
24  /* Executa o kernel */
25  axpy<<<((size + (255))/256), 256>>>(size, a, d_x, d_y);
26
27  /* Copia o resultado para memória local e libera a memória na GPU */
28  cudaMemcpy(y, d_y, size * sizeof(float), cudaMemcpyDeviceToHost);
29  cudaFree(d_x);
30  cudaFree(d_y);
31
32  return 0;
33 }
```

Fonte: Próprio autor.

ociosidade no uso das CPUs, pois as rotinas paralelas são executadas apenas nas GPUs da plataforma. Neste sentido, existe uma perda de potencial de execução, visto que se uma parte do trabalho também fosse destina às CPUs, o uso de ambas arquiteturas simultaneamente poderia melhorar ainda mais o desempenho.

A partir do exposto no parágrafo anterior, há algumas ferramentas que permitem aos programadores explorar o processamento simultâneo em CPUs e GPUs. Está seção irá destacar três delas, sendo StarPU, XKaapi e OmpSs.

2.3.1 StarPU

StarPU[4] é uma ferramenta baseada em tarefas que visa explorar arquiteturas heterogêneas formadas por CPUs e GPUs. Ela, que é composta por uma *runtime* e por um ambiente de programação voltado às linguagens da família C, permite ao programador desenvolver aplicações sem a necessidade de adaptá-las especificamente para um determinado dispositivo ou arquitetura (INRIA, 2016). Para isso, o seu sistema de *runtime* utiliza diferentes implementações de uma mesma tarefa para cada arquitetura, bem como gerencia as alocações e transferências de dados entre elas de forma transparente (HUGO et al., 2013).

A implementação das tarefas para as diferentes arquiteturas que a ferramenta StarPU possui suporte se dá por meio de uma estrutura denominada *codelet*. Através dela, pode ser especificado um *kernel* computacional com implementações para três diferentes alvos: CPUs, dispositivos CUDA e dispositivos OpenCL. Deste modo, a partir do momento que uma tarefa é criada e submetida, o escalonador da *runtime* StarPU define onde ela será executada (INRIA, 2016).

O principal recurso oferecido pelo StarPU são as dependências de dados que podem ser adicionadas às tarefas. Estas dependências são levadas em consideração pelo seu escalonador para executá-las, bem como utilizadas para controlar as transferências de dados entre as arquiteturas, que desta maneira são definidas de forma implícita. Em relação à submissão das tarefas, elas podem ser realizadas de forma síncrona ou assíncrona.

A *runtime* StarPU suporta diversos algoritmos de escalonamento, como HEFT (*Heterogeneous Earliest First Time*), PHEFT (*Parallel* HEFT), prioridades e roubo de tarefas (AUGONNET; THIBAULT; NAMYST, 2010). Para as execuções combinando CPUs e GPUs, pode-se utilizar o HEFT, pois ele contempla processadores heterogêneos. Através de predição de desempenho, ele aloca as tarefas para serem processadas na unidade que minimiza o tempo de execução (AUGONNET; THIBAULT; NAMYST, 2010). Neste sentido, seu uso requer uma fase de calibragem, visto que para a execução do programa é necessário já ter medido os tempos de processamento das tarefas para todos os recursos disponíveis. O algoritmo PHEFT também suporta escalonamento para unidades de diferentes arquiteturas e trabalha de forma semelhante ao HEFT. Porém, de acordo com o manual da *runtime* StarPU (INRIA, 2016), as tarefas para CPUs são processadas de forma paralela através de outra ferramenta, como OpenMP. Ainda segundo o manual da *runtime* StarPU (INRIA, 2016), o PHEFT não permite a execução de múltiplas tarefas paralelas em CPUs simultaneamente, devendo todos os núcleos disponíveis serem combinados para o processamento de apenas uma.

A Figura 2.6 apresenta um exemplo de programa StarPU. Ele possui um *codelet* com implementações de funções para CPU e para um dispositivo CUDA. A definição deste *codelet* é ilustrada entre as linhas 16 e 24, sendo que ele dá a origem a uma tarefa nas linhas 26 e 27. Por fim, ela é submetida de forma síncrona na linha 32.

[4]StarPU: <http://starpu.gforge.inria.fr/>

Figura 2.6 – Exemplo de um programa StarPU.

```
1 #include <starpu.h>
2
3 void cpu_func(void *buffers[], void *cl_arg) {
4   //Código CPU
5 }
6
7 void cuda_func(void *buffers[], void *cl_arg) {
8   //Chamada do kernel CUDA
9 }
10
11 int main(int argc, char **argv) {
12   //Inicializa a StarPU
13   starpu_init(NULL);
14
15   //Definição do codelet
16   struct starpu_codelet cl;
17   starpu_codelet_init(&cl);
18   cl.cpu_funcs[0] = cpu_func;
19   cl.cpu_funcs_name[0] = "cpu_func";
20 #ifdef STARPU_USE_CUDA
21   cl.cuda_funcs[0] = cuda_func;
22 #endif
23   cl.where = STARPU_CPU | STARPU_CUDA
24   cl.nbuffers = 0;
25
26   struct starpu_task *task = starpu_task_create();
27   task->cl = &cl; //Ponteiro para o codelet
28
29   //Indica que a tarefa possuirá uma chamada síncrona
30   task->synchronous = 1;
31   //Submete a tarefa para a StarPU executar
32   starpu_task_submit(task);
33
34   //Finaliza a StarPU
35   starpu_shutdown();
36   return 0;
37 }
```

Fonte: Adaptado de INRIA (2016).

2.3.2 XKaapi

XKaapi[5] é um modelo de programação e uma *runtime* para aplicações de alto desempenho voltado à máquinas *multicore* e arquiteturas heterogêneas (MENTEC; DANJEAN; GAUTIER, 2011). Baseado em um escalonador com roubo de tarefas, tanto a codificação como a execução dos programas em XKaapi é feita por meio de tarefas assíncronas, sendo possível definir dependências entre elas a fim de garantir o fluxo correto de processamento (LIMA et al., 2012).

O XKaapi é uma evolução do modelo Athapascan. Para sua *runtime*, o paralelismo deve ser explícito pelo programador, enquanto que a sincronização para a execução das tarefas

[5]XKaapi: <http://kaapi.gforge.inria.fr/>

e transferências de dados é implícita, pois é controlada por ela através das dependências de dados (LIMA et al., 2012). A forma como as tarefas são executadas assemelha-se com a da linguagem Cilk, não só pelo uso do roubo de tarefas, mas também pela capacidade de permitir recursividade (GAUTIER et al., 2013a).

A construção de um programa XKaapi pode ser feita através das linguagens C e C++, onde são expressados blocos de código sequenciais e paralelos. Entre as opções mais utilizadas, estão as chamadas de baixo nível da *runtime* em C e a API Kaapi++, que permite a criação de uma aplicação em alto nível utilizando C++. Ainda existe a possibilidade de utilizar anotações `#pragma kaapi` a exemplo do que ocorre com o OpenMP, porém neste caso é necessário um compilador que possua suporte a este recurso (LIMA, 2014).

Além de permitir a criação de tarefas através de chamadas de funções específicas para esta finalidade, o XKaapi também possui outros recursos, como por exemplo, laços paralelos adaptativos (GAUTIER et al., 2013a). Existe ainda uma implementação de *runtime* denominada LIBKOMP que pode ser utilizada para a execução de programas OpenMP (BROQUEDIS; GAUTIER; DANJEAN, 2012). Por fim, é importante ressaltar que o suporte a aceleradores disponibilizado pelo XKaapi requer a implementação de mais de uma versão do mesmo código, nos moldes do que é realizado através da ferramenta StarPU.

2.3.3 OmpSs

OmpSs[6] é uma plataforma para desenvolvimento de programas de alto desempenho do Barcelona Supercomputing Center. Ela é oriunda da integração das características da ferramenta OpenMP (Omp) e do modelo de programação StarSs (Ss) e visa explorar principalmente arquiteturas *multicore* com aceleradores (ANDERSCH; CHI; JUURLINK, 2012).

O paralelismo em um programa OmpSs é representado através de tarefas, que podem ser síncronas ou assíncronas. Essas tarefas podem conter dependências de dados, garantindo assim a possibilidade de não utilizar sincronizações explícitas. Para representar as tarefas são utilizadas, a exemplo do OpenMP, diretivas `#pragma`. Deste modo, a compilação de uma aplicação OmpSs requer um compilador com suporte a tais diretivas. Assim, o *toolkit* da ferramenta é composto por um compilador denominado Mercuriun e por um sistema de *runtime* denominado Nanos++, sendo este responsável por analisar as dependências de dados e escalonar as tarefas nos recursos de processamento disponíveis (BSC, 2015).

O suporte a aceleradores do OmpSs permite a implementação da mesma função em diferentes versões a fim de que ela possa ser executada simultaneamente em mais de uma arquitetura de processamento. Como a ferramenta é baseada em diretivas de compilação, as funções a serem executadas em CPU e GPU devem ser anotadas com clausulas `#pragma` indicando ao compilador que elas representam a mesma rotina. As implementações para aceleradores supor-

[6]OmpSs: <http://pm.bsc.es/ompss>

tadas são CUDA e OpenCL (OZEN; AYGUADÉ; LABARTA, 2014).

2.4 TRABALHOS RELACIONADOS

Após a descrição das arquiteturas e dos conceitos para programação paralela, bem como das ferramentas multi-CPU e multi-GPU, esta seção apresenta os trabalhos já existentes que são relacionados com o presente estudo.

Na área de processamento de alto desempenho, é possível citar como trabalhos relacionados ferramentas com suporte ao processamento heterogêneo (CPU+GPU) e APIs C++ de alto nível que visam o processamento paralelo. Na Tabela 2.1, as características principais das ferramentas aqui abordadas estão sumarizadas.

Tabela 2.1 – Características das interfaces relacionadas.

Característica	**Ferramenta/API**								
	Kaapi++	StarPU	OmpSs	OpenMP 4.0	Kokkos	C++ AMP	Thrust	Phalanx	HPX
multi-CPU	✓	✓	✓	✓	✓			✓	✓
GPU	✓	✓	✓	✓	✓	✓	✓	✓	
multi-GPU	✓	✓	✓					✓	
Processamento CPU+GPU	✓	✓	✓						
Laços paralelos			✓	✓	✓	✓	✓		
Laços paralelos CPU+GPU									
Portabilidade do código				✓	✓	✓	✓	✓	
Transf. de dados implícitas	✓	✓	✓	✓				✓	
Tarefas assíncronas	✓	✓	✓	✓				✓	✓
Processamento distribuído		✓	✓					✓	✓

Fonte: Próprio autor.

As ferramentas StarPU e OmpSs, que possuem suporte a multi-CPU, multi-GPU e ao processamento heterogêneo já foram descritas nas seções 2.3.1 e 2.3.3, respectivamente. Deste modo, os tópicos a seguir iniciam pela descrição da API Kaapi++, que como já descrito na seção 2.3.2, permite a codificação de programas para explorar a *runtime* XKaapi.

2.4.1 Kaapi++

A API Kaapi++ é uma interface C++ para a codificação de programas multi-CPU e multi-GPU. Programas que utilizam a Kaapi++ podem explorar um paralelismo de tarefas com dependências de dados. Uma tarefa é criada fazendo uso de objetos C++ disponibilizados pela API, sendo que nela é possível indicar quais dados são pré-requisitos para sua execução. Após submeter uma tarefa, as dependências são consideradas para definir o momento em que ela será executada pela *runtime* XKaapi (GAUTIER et al., 2013b).

Apesar da Kaapi++ fazer uso dos recursos de alto nível da linguagem C++, a codificação de um programa CPU+GPU através dela é trabalhosa pois demanda do programador a escrita

de duas versões do código, uma para CPU e uma para GPU (em CUDA ou OpenCL). Além disso, ela é baseada em tarefas assíncronas.

2.4.2 OpenMP 4.0

A partir da versão 4.0, o OpenMP passou a oferecer suporte a aceleradores. Para isso, foi adicionada à sua especificação a diretiva `target`, onde o código implementado dentro dela é executado por um único acelerador. Além disso, a diretiva `target` também possui a claúsula `map` que pode ser utilizada para controlar as transferências de dados entre a memória principal e a memória do acelerador (OpenMP, 2016).

O uso de aceleradores em uma aplicação OpenMP requer suporte do compilador, pois o mesmo é implementado através de diretivas `#pragma`. Existem alguns compiladores que já possuem esse suporte, como o ICC 16 e o GCC 4.9.1, porém apenas para o acelerador Intel Xeon Phi. Caso o acelerador seja uma GPU, existe uma implementação disponível do LLVM que compila o código especificado dentro da diretiva `target` para ser executado em uma GPU NVIDIA. No entanto, este suporte ainda está em desenvolvimento e não foi incluído na versão estável, podendo ser acessado apenas através do repositório Git[7].

2.4.3 Kokkos

De acordo com Edwards et al. (2012), a Kokkos é uma API que permite a construção de laços e reduções paralelas através de uma biblioteca puramente C++. O uso da Kokkos para a implementação de um programa paralelo não requer suporte de compilação, pois diferente de outras ferramentas que foram modeladas como uma nova linguagem ou são baseadas em diretivas `#pragma`, a Kokkos é uma biblioteca. Através da Kokkos, o *kernel* da aplicação (implementado através de um *Functor*) possui portabilidade para ser executado tanto em CPUs como em GPU, ou seja, nenhuma alteração precisa ser feita nele para migrar de arquitetura. Seus autores comparam a Kokkos com a biblioteca Thrust, que será abordada em mais detalhes no tópico 2.4.5.

Além de *Functors* para o mapeamento das rotinas, a Kokkos utiliza *Views* para o mapeamento dos dados a serem processados em paralelo. Este processamento é realizado apenas de forma *multicore* ou explorando uma única GPU. No caso de GPU, a Kokkos requer que as transferências de memória entre os dispositivos sejam realizadas de forma explícita, exceto quando é utilizada memória mapeada através do recurso UVM (*Unified Virtual Memory*) da plataforma CUDA.

[7]LLVM Clang OpenMP: <http://clang-omp.github.io/>

2.4.4 C++ AMP

C++ AMP (*Accelerated Massive Parallelism*) é uma biblioteca e uma extensão de linguagem que permite o desenvolvimento de programas heterogêneos através do C++ (GREGORY; MILLER, 2012). A exemplo do que ocorre com a API Kokkos, através dela pode-se criar laços paralelos para GPU que irão percorrer cada um dos elementos de um vetor (WYNTERS, 2016). Ainda de acordo com Wynters (2016), uma vantagem do C++ AMP é que ele permite a execução em diferentes arquiteturas, como GPUs NVIDIA e AMD Radeon, ao contrário de outras bibliotecas utilizadas para a mesma finalidade, como Thrust e Kokkos.

Outra característica da C++AMP é o uso de *Views* para mapear os dados para o processamento em paralelo. As transferências de memória destas *Views* para uma GPU devem ser realizadas de forma explícita. Por fim, sua interface possui suporte para o processamento em blocos, sendo que o particionamento destes dados é realizado apenas para a execução do laço e não contempla as transferências, que precisam ser realizadas com a totalidade das informações.

2.4.5 Thrust

A biblioteca Thrust é uma implementação da C++ *Standard Library* em CUDA, e desta forma, o paralelismo é alcançado em alto nível, apenas passando objetos C++ (como *Containers* e *Functors*) para suas rotinas (como ordenações, reduções e pesquisas). Além do suporte a CUDA para a execução em GPUs NVIDIA, a Thrust ainda oferece interoperabilidade com outras ferramentas de programação paralela, como a OpenMP (THRUST, 2016).

A Thrust é fortemente atrelada as chamadas da *Standard Library*, o que diminui a flexibilidade na criação das rotinas paralelas, pois as mesmas precisam ser codificadas a partir de funções e objetos disponíveis nela. Somado a isso, a exemplo das interfaces já descritas nos dois últimos tópicos (Kokkos e C++ AMP), ela permite o processamento em apenas uma GPU e requer transferências de dados explícitas.

2.4.6 Phalanx

Com suporte não só a CPUs e GPUs como também a arquiteturas de memória distribuída, o modelo de programação Phalanx é descrito por Garland, Kudlur e Zheng (2012) como uma biblioteca C++ que possibilita alcançar paralelismo em máquinas heterogêneas (inclusive multi-nó) através do uso de tarefas assíncronas. A definição da arquitetura onde a tarefa será executada é realizada ao criá-la. Para o gerenciamento da memória em diferentes nós, o Phalanx utiliza uma *runtime* com um espaço de endereço global baseada na ferramenta GASNet[8], sendo que a mesma cobre tanto as memórias das CPUs como as das GPUs de um *cluster*.

[8]GASNet: <http://gasnet.lbl.gov/>

Apesar de ser multi-CPU e multi-GPU, o Phalanx não permite o processamento heterogêneo de forma simultânea, ou seja, ele realiza a execução das tarefas em apenas uma das arquiteturas por vez. Mesmo assim, o Phalanx difere-se das demais ferramentas multi-CPU e multi-GPU apresentadas até o momento por viabilizar o desenvolvimento em alto nível com portabilidade do código para as diferentes arquiteturas.

2.4.7 HPX

A HPX é a única API C++ de alto nível voltada ao alto desempenho e abordada nesta seção que não possui suporte para aceleradores. De acordo com Heller, Kaiser e Iglberger (2013), a HPX (*High Performance ParallelX*) é uma implementação do modelo de execução ParallelX baseada na biblioteca Boost e no padrão C++11. Através dela é possível codificar programas com tarefas assíncronas para execução em ambientes de memória compartilha e distribuída.

O Stellar Group, responsável pelo projeto do HPX, já está trabalhando para disponibilizar o suporte a aceleradores para a API. O objetivo ao introduzir este suporte é garantir a portabilidade do código para as diferentes arquiteturas, bem como deixá-lo adequado aos padrões da linguagem C++ (KAISER, 2016).

2.5 CONCLUSÃO

O presente capítulo detalhou alguns tópicos relacionados a programação voltada ao alto desempenho, dando enfoque principalmente na área de aceleradores. Após apresentar as arquiteturas de alto desempenho, abordou-se as ferramentas de programação utilizadas para explorá-las. Alguns exemplos de código fonte foram adicionados para garantir maior riqueza na explanação do uso destas ferramentas, porém isto restringiu-se apenas às interfaces relacionadas ao contexto deste trabalho.

A última seção deste capítulo abordou os estudos relacionados, onde verificou-se que já existem ferramentas, como StarPU, Kaapi++ e OmpSs, que permitem o processamento em multi-CPU e multi-GPU de forma simultânea. Todavia, nenhuma delas possibilita expressar o paralelismo empregando laços paralelos, pois as três são baseadas em tarefas. Além disso, para utilizá-las existe a necessidade de escrever duas versões do código, uma para CPU e uma para GPU. Apesar de existirem outras interfaces que fazem uso de laços paralelos, como Kokkos, C++ AMP e Thrust, elas não possuem suporte ao processamento CPU+GPU, bem como requerem que as transferências de dados entre os dispositivos sejam realizadas de forma explícita.

3 A INTERFACE DE PROGRAMAÇÃO HPSM PARA CPUS E ACELERADORES

A HPSM é uma API C++ que oferece recursos para a construção de programas paralelos voltados ao processamento em CPUs e GPUs. Sua principal característica é a execução de iterações de laços e reduções paralelas simultaneamente em ambas arquiteturas. Além disso, ela permite processar uma mesma rotina paralela por meio de diferentes *back-ends*, sendo suportados Serial, OpenMP e StarPU. Isto garante que um programa que será executado em CPU e GPU seja implementado com apenas uma versão de código escrita em C++.

Outras funcionalidades importantes da HPSM são as transferências de dados implícitas entre as memórias de diferentes dispositivos, o mapeamento explícito dos dados através de *Views* usando *templates* C++, o mapeamento de rotinas paralelas a partir do objeto C++ denominado *Functor*, a portabilidade de um código para todos os seus *back-ends* e diferentes arquiteturas, o emprego do padrão C++11, bem como a possibilidade de utilizá-la para execuções com somente CPUs, somente GPUs e CPUs+GPUs. No entanto, a nova API não possui suporte a tarefas e nem ao processamento distribuído.

No restante deste capítulo, a HPSM será detalhada. Assim, o mesmo está sub-dividido em quatro seções. Na primeira (3.1), as funcionalidades da API são abordadas, sendo relatados diversos aspectos inerentes a sua aplicação no desenvolvimento de programas paralelos. Já a segunda seção (3.2) é voltada a caracterizar e documentar a estrutura da API, detalhando suas classes e funções principais. Na seção seguinte (3.3), é realizada uma comparação da nova API com as interfaces relacionadas descritas na seção 2.4 (página 27). Por fim, a última seção (3.4) descreve o esforço de programação necessário para codificar aplicações por meio da HPSM em comparação a outras interfaces com a mesma finalidade.

3.1 FUNCIONALIDADES DA HPSM

A HPSM é uma biblioteca composta por classes e funções na linguagem C++. Para utilizar suas funcionalidades, deve-se importá-la ao código fonte do programa paralelo, conforme apresentado na linha 1 do código da Figura 3.1. A partir deste momento, todas as chamadas da API devem ser realizadas através do *namespace* `hpsm`, que contempla rotinas de mapeamentos de dados e execução de funções. As mesmas estão descritas em detalhes nos tópicos a seguir.

3.1.1 Execução através de *back-ends*

A HPSM foi modelada de modo a permitir que o processamento paralelo de um programa pudesse ser realizado por meio de diferentes *back-ends*. Cada um deles representa uma

Figura 3.1 – Importando a biblioteca “hpsm.hpp” da HPSM.

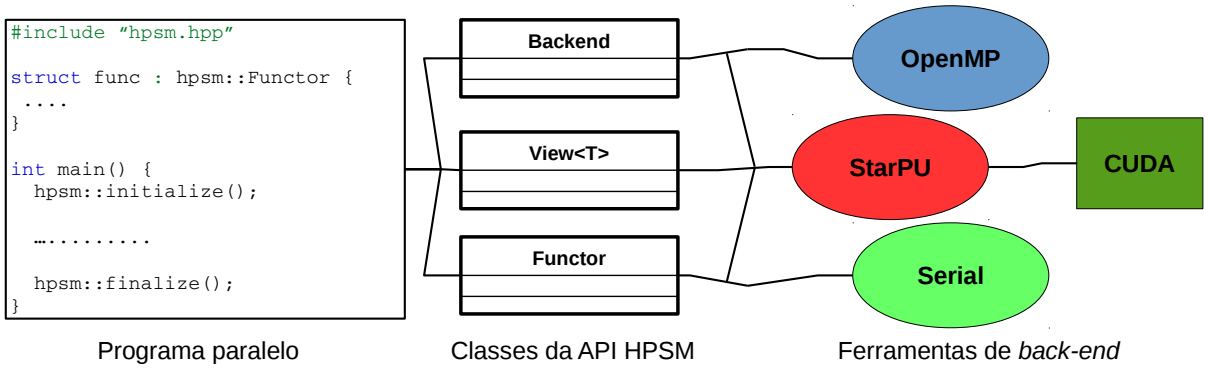
```
1 #include "hpsm.hpp"
2
3 /* Após incluir a biblioteca "hpsm.hpp",
4  * pode-se fazer uso do namespace "hpsm" */
5 using namespace hpsm;
```

Fonte: Próprio autor.

ferramenta de *runtime* para execução da aplicação, como OpenMP ou StarPU. Durante a sua codificação, o *back-end* para o qual o código está sendo construído não precisa ser considerado pelo programador, ou seja, suas rotinas são acessadas de forma transparente. Isso ocorre pois a API garante a portabilidade do código paralelo independente da maneira como ele será executado, isto é, o mesmo fonte funciona para todos os *back-ends*.

O acesso às rotinas da API se dá, conforme já mencionado, através do *namespace* `hpsm`. Todas as chamadas existentes neste *namespace* possuem um correspondente em todos os *back-ends* suportados. Neste sentido, as classes principais da HPSM tornam-se uma camada que abstrai o acesso do programa paralelo aos *back-ends*, conforme pode ser observado na Figura 3.2. Para o programador, o *back-end* passa a ter importância somente na fase de compilação do código, pois é nela que ele será selecionado. Apenas um *back-end* pode ser escolhido por compilação. Mais detalhes sobre a implementação da API para ligar as rotinas do *namespace* `hpsm` com os *back-ends* serão abordados na seção 3.2.

Figura 3.2 – Modelagem da HPSM através de *back-ends*.



Fonte: Próprio autor.

O desenvolvimento de um código sem as especifidades de uma determinada ferramenta ou arquitetura é uma importante vantagem trazida pelo uso de *back-ends*. Atualmente, a HPSM suporta três *back-ends*, sendo dois deles paralelos: StarPU e OpenMP. As ferramentas que deram origem a eles foram descritas nos tópicos 2.3.1 e 2.2.2.2, respectivamente. O *back-end* Serial, que como o nome sugere não é paralelo, pode ser utilizado para realizar o processamento de um programa em blocos. O *back-end* StarPU é utilizado para processamento paralelo tanto nas CPUs como nas GPUs (através da CUDA) disponíveis na plataforma. Já o *back-end* OpenMP permite explorar arquiteturas *multicore*.

3.1.2 Mapeamentos dos dados

O processamento paralelo realizado pela HPSM ocorre sobre estruturas de dados que precisam ser mapeadas para ela. A exemplo do que acontece nas APIs Kokkos e C++ AMP, este mapeamento é feito através de *Views*, sendo efetuado de forma explícita. Ao realizá-lo, o dado será registrado no *back-end* e poderá, caso necessário, ser particionado em blocos. É possível mapear para uma *View* estruturas de *arrays* (como `int*` ou `float*`) da linguagem C, sendo que elas podem ser adicionadas como vetor (1 dimensão) ou matriz (2 dimensões). A escolha deste modo de mapeamento (vetor ou matriz) irá influenciar no acesso aos dados da estrutura dentro da rotina paralela. Na Figura 3.3 pode-se visualizar um código onde são mapeadas duas estruturas para as *Views* da API.

Figura 3.3 – Mapeamento de *arrays* para *Views* da HPSM.

```
1 //Tamanho das entradas e do bloco
2 int size = 16384, lines = 16384, columns = 16384, block_size = 512;
3
4 //Declaração e alocação de memória
5 float *vector, *matrix;
6 vector = new float[size];
7 matrix = new float[lines*columns];
8
9 //View mapeando um vetor - 1 Dimensão
10 hpsm::View<float> view_vector(vector, size, block_size,
11         hpsm::AccessMode::In);
12
13 //View mapeando uma matriz - 2 Dimensões
14 hpsm::View<float> view_matrix(matrix, lines, columns, block_size,
15         hpsm::PartitionMode::Matrix_Vert_Horiz, //Particionamento
16         hpsm::AccessMode::InOut);
```

Fonte: Próprio autor.

Além do modo de mapeamento (vetor ou matriz), uma *View* exige que seja informado o tamanho da estrutura, o tamanho do bloco, a forma de particionamento e o modo de acesso. Em relação ao tamanho, deve-se informar um único valor para vetores e a quantidade de linhas e colunas quando tratar-se de matrizes. O tamanho do bloco e a forma de particionamento indicam a maneira como a estrutura será particionada. Para os vetores, o particionamento do *array* considera apenas o tamanho do bloco. Já para as matrizes, existem três formas de particionamento. Os itens abaixo que fazem o detalhamento destas formas consideram L para linhas e C para colunas:

- **Horizontal:** particiona uma matriz pelas suas colunas, ou seja, uma matriz de tamanho LxC com blocos de tamanho B gerará partições de tamanho LxB;
- **Vertical:** particiona uma matriz pelas suas linhas, ou seja, uma matriz de tamanho LxC com blocos de tamanho B gerará partições de tamanho BxC;

- **Vertical e Horizontal:** particiona uma matriz pelas suas linhas e colunas, ou seja, uma matriz de tamanho LxC com blocos de tamanho B gerará partições de tamanho BxB. Considerando a Figura 3.3, este particionamento é aplicado ao mapeamento realizado na linha 14.

O último dado a ser informado para uma *View* é o modo de acesso. Esta informação é utilizada para gerenciar as cópias de memória para um acelerador (GPU), que são realizadas de forma implícita. No *back-end* StarPU, ela também é usada no controle das dependências entre as tarefas.

- **In:** indica que os dados são de entrada. Considerando as cópias para um acelerador, este modo disponibilizará os dados apenas para leitura;
- **Out:** indica que os dados são de saída. Considerando as cópias para um acelerador, este modo disponibilizará os dados apenas para escrita;
- **InOut:** engloba os dois modos anteriores, indicando que os dados são de entrada e saída. Considerando as cópias para um acelerador, este modo disponibilizará os dados para leitura e escrita.

3.1.3 Execução paralela

Um programa paralelo utilizando a HPSM deve ser implementado para executar laços (`parallel_for`) ou reduções paralelas (`parallel_reduce`). Para isso, é preciso mapear a rotina paralela para um *Functor*, um tipo de classe C++ que caracteriza-se por sobrescrever o seu operador de aplicação (`operator()`) (STROUSTRUP, 2013). Mais detalhes sobre *Functor* podem ser obtidos no apêndice A (página 83). As Figuras 3.4 e 3.5 demonstram a declaração de *Functors* no formato requisitado pela API, bem como a sua utilização na chamada das funções paralelas.

Na Figura 3.4, está exemplificado um algoritmo AXPY que será executado paralelamente utilizando um laço. Considerando que as *Views* para os vetores `x` e `y` foram previamente mapeadas conforme o exemplo da Figura 3.3, bem como sua passagem para o *Functor* na linha 35, a chamada do laço paralelo ocorre na linha 37. Nele, será executado o `operator()` do `funcAXPY` para cada índice contido no intervalo do *range* definido na linha 36. O segundo parâmetro passado à função `parallel_for` (`view_y.block_range()`) indica qual *View* será utilizada como base para gerar o *range* de cada bloco.

Em relação ao *range*, o terceiro parâmetro indica a forma como os blocos serão escalonados no caso das *Views* do laço possuírem números distintos de partições. Ele é necessário pois a HPSM processa apenas um bloco de cada *View* por iteração, sendo preciso indicar como

Figura 3.4 – Execução de laço paralelo com a rotina `parallel_for` da HPSM.

```
1 using View = hpsm::View<float>; //Alias para o tipo da View
2
3 /* "struct funcAXPY" deve herdar a classe "hpsm::Functor"
4  * para ter acesso aos métodos do back-end */
5 struct funcAXPY : hpsm::Functor {
6   View x, y; float a;
7
8   /* Construtor deve registrar as View's no
9    * back-end chamando "register_data" */
10  funcAXPY(View _x, View _y, float _a) : x(_x), y(_y), a(_a) {
11    register_data(x, y);
12  }
13
14  /* Operador de cópia necessário para copiar
15   * o objeto "struct funcAXPY" para a GPU */
16  funcAXPY(const funcAXPY& f) : hpsm::Functor(f),
17    x(f.x), y(f.y), a(f.a) {
18    clear_data();
19    register_data(x, y);
20  }
21
22  /* Operador de aplicação que executa de forma paralela
23   * o kernel do programa.
24   * "PARALLEL_FUNCTION" é uma macro de compilação,
25   * como "__device__" para CUDA.
26   * "hpsm::index<1>" indica 1 dimensão para o acesso às View's */
27  PARALLEL_FUNCTION
28  void operator()(hpsm::index<1> i) {
29    y(i) = x(i) * a + y(i);
30  };
31 };
32
33 /*** ------ PROGRAMA PRINCIPAL ------ ***/
34 /* Chamada do laço paralelo, onde percorre-se o range<1> (0..N) */
35 funcAXPY func(view_x, view_y, 3.41);
36 hpsm::range<1> rg(0, N, hpsm::BlockTile::Intercalary);
37 hpsm::parallel_for(rg, view_y.block_range(), func);
38
39 /* Remoção das View's do back-end da API */
40 func.remove_data();
```

Fonte: Próprio autor.

será feita a combinação em caso de valores divergentes no total de blocos de cada uma. Existem duas formas de escalonamento:

- **Intercalada:** escalonamento realizado intercalando os blocos. Supondo que existam duas *Views* em um laço (A e B), sendo a primeira particionada em quatro blocos (A1, A2, A3, A4) e a segunda em dois (B1 e B2), o escalonamento dos quatro blocos será: A1 e B1, A2 e B1, A3 e B2 e A4 e B2;
- **Sequencial:** escalonamento realizado agrupando os blocos sequencialmente. Supondo que existam duas *Views* em um laço (A e B), sendo a primeira particionada em quatro

Figura 3.5 – Execução de redução paralela com a rotina `parallel_reduce` da HPSM.

```
1 using View = hpsm::View<float>; //Alias para o tipo da View
2
3 /* Declaração do Functor */
4 struct funcRed : hpsm::Functor {
5   View mat;
6
7    /* Construtor e operador de cópia */
8   funcRed(View _mat) : mat(_mat) {
9     register_data(mat);
10  }
11
12  funcRed(const funcRed& f) : hpsm::Functor(f), mat(f.mat) {
13    clear_data(); register_data(mat);
14  }
15
16  /* "hpsm::index<2>" indica 2 dimensões para o acesso às View's */
17  PARALLEL_FUNCTION
18  void operator()(hpsm::index<2> i) {
19
20  /* "hpsm::atomic_add" para proteger o acesso pela GPU ao
21   * ponteiro da variável de redução "reduction_var<float>()" */
22    hpsm::atomic_add(reduction_var<float>(), Mat(i));
23  };
24 };
25
26 /*** ------ PROGRAMA PRINCIPAL ------ ***/
27 float red; //Variável de redução
28
29 /* Percorre-se os range<2> (0..N) e (0..N) para as linhas e colunas,
30  * reduzindo a variável "red" usando "ReduxMode::Sum" */
31 funcRed func(view_mat);
32 hpsm::range<2> rg(0, N, 0, N);
33 hpsm::parallel_reduce(rg_mat, MAT.block_range(), func_mat,
34       red, hpsm::ReduxMode::Sum);
35
36 /* Remoção das View's do back-end da API */
37 func_mat.remove_data();
```

Fonte: Próprio autor.

blocos (A1, A2, A3, A4) e a segunda em dois (B1 e B2), o escalonamento dos quatro blocos será: A1 e B1, A2 e B2, A3 e B1 e A4 e B2.

Como exemplo de redução paralela, a Figura 3.5 apresenta um programa que executa uma soma de todos os valores de uma matriz. Seu código e forma de execução são semelhantes aos detalhados na Figura 3.4, porém há dois itens que precisam ser destacados. Primeiro, a variável de redução deve ser passada para a função `parallel_reduce` (linha 35) junto com seu o modo de redução, onde são suportados soma, multiplicação (somente CPU), máximo e mínimo. Ao final do processamento do laço, esta variável retornará ao programa principal com o valor da redução. Segundo, no `operator()` do `funcRed` o acesso a variável de redução deve ser realizado por meio do método `reduction_var<T>()` (linha 23), sendo `T` seu tipo.

A portabilidade do código paralelo para as arquiteturas de CPU e GPU trazida pela HPSM é resultante da utilização de *Functors*. Isso é possível através dos *back-ends*, onde as suas classes realizam os procedimentos necessários para criar uma versão CPU e uma versão GPU do mesmo código. Somado a isto, a macro de compilação `PARALLEL_FUNCTION` é utilizada para tratar a compilação do operador de aplicação do *Functor*, visto que caso opte-se pelo *back-end* StarPU, ele precisa ser construído para ambas arquiteturas.

3.1.4 Compilação

A compilação de um código com a HPSM destaca-se pelo fato de ser preciso selecionar o *back-end* com o qual ele será executado. Além da compilação dos arquivos do programa que foi codificado, também precisam ser construídos os arquivos com as rotinas da API referentes ao *back-end* que foram acessadas por meio do *namespace* `hpsm`. Para facilitar este processo, é disponibilizado no diretório raiz da API o arquivo `Makefile.parallel` que é responsável por realizar a compilação dos arquivos do *back-end*. O mesmo funciona em plataformas Linux e precisa ser incluído ao GNU Make da aplicação. A Figura 3.6 mostra um exemplo de `Makefile` que pode ser utilizado para compilar o programa AXPY da Figura 3.4.

No fonte do arquivo `Makefile`, é preciso ressaltar três linhas principais. Primeiro, a linha 2 onde é indicado o diretório onde a API está disponível. Em seguida, a linha 5 define qual *back-end* será utilizado para a compilação. Após as definições das *flags* de compilação de acordo com o *back-end* entre a linhas 11 e 28, a linha 33 inclui o arquivo `Makefile.parallel` que indica as dependências que também serão construídas para que o programa possa ser executado pelo *back-end* selecionado.

Outro fator de destaque no código de compilação da Figura 3.6 é a definição do compilador para o *back-end* StarPU. Como a execução através desta *runtime* poderá ser realizada em CPUs ou GPUs, é preciso utilizar o compilador NVCC da NVIDIA, visto que ele possui suporte a CUDA.

3.2 ESTRUTURA DA HPSM

A HPSM foi codificada utilizando recursos oferecidos pela linguagem C++, como classes e *templates*. A título de documentação, é importante detalhar a sua estrutura a fim de que seja possível conhecer como ela funciona internamente. Neste sentido, está seção descreve a organização de classes da API, abordando também as rotinas principais que precisam ser acessadas para a construção dos programas paralelos.

Figura 3.6 – GNU Make para compilação de um programa utilizando a HPSM.

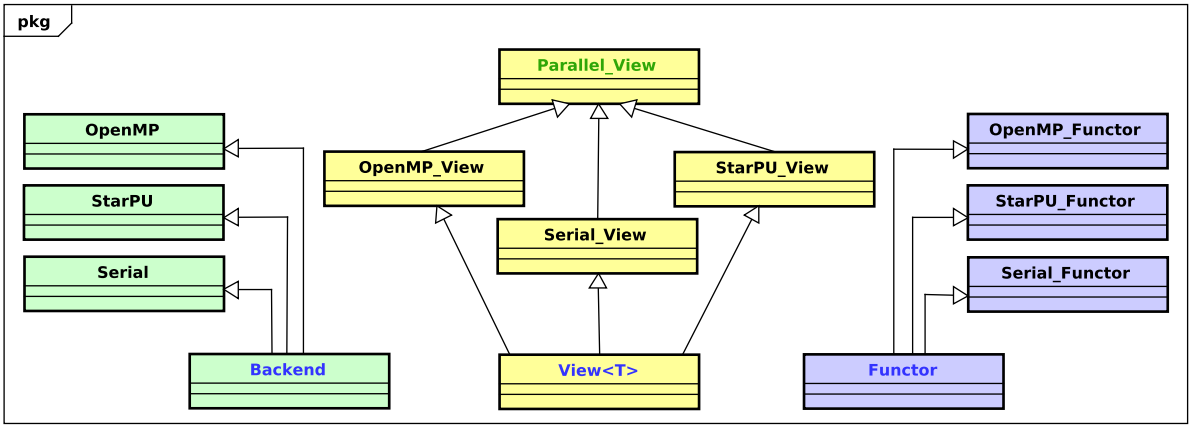
```
1 #Diretório da API
2 PARALLEL_PATH := $(HOME)/api
3
4 #Back-ends: OpenMP, StarPU ou Serial
5 PARALLEL_BACKEND = "StarPU"
6
7 NAME := "axpy"
8 SRC = $(wildcard *.cpp)
9 OBJ = $(SRC:.cpp=.o)
10
11 ifneq (,$(findstring StarPU,$(PARALLEL_BACKEND)))
12   CXX= nvcc
13   LINK = $(CXX)
14   EXE = $(NAME).starpu
15   CXXFLAGS = -g -x cu -rdc=true
16 else
17   ifneq (,$(findstring OpenMP,$(PARALLEL_BACKEND)))
18     CXX= g++
19     LINK = $(CXX)
20     EXE = $(NAME).omp
21     CXXFLAGS = -g -fopenmp
22   else
23     CXX= g++
24     LINK = $(CXX)
25     EXE = $(NAME).serial
26     CXXFLAGS = -g
27   endif
28 endif
29
30 default: $(EXE)
31
32 #Inclusão do Makefile.parallel para compilação do back-end
33 include $(PARALLEL_PATH)/Makefile.parallel
34
35 $(EXE): $(OBJ) $(PARALLEL_LINK_DEPENDS)
36   $(LINK) $(PARALLEL_LDFLAGS) $(LDFLAGS) $(OBJ) $(PARALLEL_LIBS) -o $(EXE)
37
38 %.o:%.cpp $(PARALLEL_CPP_DEPENDS)
39   $(CXX) $(CXXFLAGS) $(PARALLEL_CPPFLAGS) $(PARALLEL_CXXFLAGS) -c $<
40
41 clean: parallel-clean
42   rm -f *.o $(EXE)
```

Fonte: Próprio autor.

3.2.1 Classes

A HPSM é orientada a objetos. Devido a quantidade de classes, neste tópico serão consideradas apenas as que são pertinentes para o uso da API, ou seja, as classes que são acessadas para a implementação de uma aplicação. Na Figura 3.7 é apresentado um diagrama onde é possível visualizar os três grupos de classes principais com suas respectivas heranças, sendo `Backend` em verde, `View` em amarelo e `Functor` em azul. Cada uma destas heranças representa um *back-end*.

Figura 3.7 – Diagrama de classes da HPSM com *back-ends*.



Fonte: Próprio autor.

A classe `Backend` é utilizada para executar as rotinas paralelas (laço ou redução), bem como para realizar a inicialização e finalização da API durante o fluxo de um programa. Já a classe `View` possui como finalidade o mapeamento dos dados para o processamento em paralelo. Ao instanciar um objeto a partir dela, é obrigatório informar como parâmetro de *template* qual o tipo do dado que está sendo mapeado (no caso `<T>`). Este parâmetro garante flexibilidade para seu uso com diferentes tipagens de dados. Por fim, a classe `Functor` é aplicada na implementação das funções paralelas que serão executadas através da API. Além disso, suas funcionalidades englobam as alocações, transferências e liberações dos dados das *Views* durante a execução do programa.

As heranças mapeadas para as três classes principais da HPSM propiciaram que sua modelagem utilizasse *back-ends*. Seus atributos e métodos são herdados da sua classe pai que, como já mencionado, representa um *back-end*. Por exemplo, a classe `Backend` herda o conteúdo das classes `OpenMP`, `StarPU` e `Serial`, configurando uma herança múltipla. No entanto, esta herança múltipla ocorre apenas em tempo de desenvolvimento. Em tempo de compilação, a classe `Backend` irá herdar apenas a classe do *back-end* selecionado. Para garantir essa funcionalidade, fez-se uso de metaprogramação. Mais detalhes sobre metaprogramação podem ser obtidos no apêndice A (página 83).

O uso das heranças múltiplas e sua seleção através de metaprogramação garantem abstração para a HPSM, pois isso propicia a portabilidade de um mesmo código para qualquer *back-end*. Também pode-se destacar a facilidade para adicionar novos *back-ends* sem a necessidade de fazer uma grande alteração nos códigos já existentes. Como cada um é independente, basta criar novas classes para serem herdadas pelas três principais, que por serem o código base, não precisam ser alteradas. Por fim, cabe ressaltar que algumas rotinas podem ser herdadas pelas classes de *back-end*, o que facilita ainda mais a sua inclusão. Este é o caso das classes `OpenMP_View`, `StarPU_View` e `Serial_View`, que herdam a classe `Parallel_View`.

3.2.2 Rotinas principais

As principais rotinas da HPSM contemplam as funções, macros, constantes e classes que precisam ser adicionadas a um programa para torná-lo paralelo. Os tópicos a seguir descrevem a finalidade e forma de uso de cada uma delas, lembrando que todas pertencem ao *namespace* `hpsm`.

3.2.2.1 Funções principais

As funções principais são métodos da classe `Backend`, porém são acessadas diretamente dentro do *namespace* `hpsm`.

- `void initialize()`: função de inicialização. Deve ser chamada antes das demais rotinas da HPSM;
- `void finalize()`: função para finalização. Deve ser chamada após todas as rotinas da API;
- `void parallel_for(range<N>, block_range, Functor)`: função que executa um laço de forma paralela. Recebe como primeiro parâmetro um `range<N>` indicando os limites da iteração (início e fim). O `block_range` determina os limites da iteração dentro do bloco, sendo obtido a partir de um objeto do tipo `View`. O último parâmetro é o `Functor` que contém a função paralela;
- `void parallel_reduce<T>(range<N>, block_range, Functor, T, Parallel_Redux)`: função que executa uma redução de forma paralela. Recebe como parâmetros um `range<N>`, um `block_range` e um `Functor` que possuem finalidades idênticas às descritas no item anterior. Além disso, recebe uma variável do tipo `T` utilizada para retornar o total da redução. O último parâmetro, do tipo `Parallel_Redux`, indica o modo de redução a ser aplicado no laço.

3.2.2.2 Classe `View`

A classe `View` é utilizada para o mapeamento das estruturas de dados (*arrays*) e o acesso às mesmas dentro dos laços ou reduções paralelas.

- `View<T>(T*, unsigned, unsigned, AccessMode)`: construtor para mapear um *array* de uma dimensão (vetor) para uma *View* da HPSM. Recebe como primeiro parâmetro o dado a ser mapeado que deve possuir o tipo `T`. Os dois parâmetros do tipo `unsigned`

(segundo e terceiro) indicam o tamanho do `array` e o tamanho do bloco para particionamento. O último parâmetro é o modo de acesso. Caso seja omitido, assume o valor `AccessMode:InOut`;

- `View<T>(T*, unsigned, unsigned, unsigned, PartitionMode, AccessMode)`: construtor para mapear um *array* de duas dimensões (matriz) para uma *View* da API. Recebe como primeiro parâmetro o dado a ser mapeado que deve possuir o tipo `T`. Os três parâmetros do tipo `unsigned` (segundo ao quarto) indicam o número de linhas e colunas da matriz, bem como o tamanho do bloco para particionamento. O quinto parâmetro, do tipo `PartitionMode`, define o modo de particionamento da matriz. Já último parâmetro é o modo de acesso. Caso seja omitido, assume o valor `AccessMode:InOut`;
- `block_range block_range()`: método que retorna um objeto do tipo `block_range`. Ele indica os limites da iteração dentro de um bloco, devendo ser utilizado nas chamadas dos laços e reduções paralelas;
- `T& operator(index<N>)`: operador de aplicação que acessa uma posição do *array* mapeado para a *View*. Somente pode ser chamado dentro do `Functor` onde será executada a função paralela. O parâmetro do tipo `index<N>` indica a posição do dado que deve ser acessada, sendo que uma referência do mesmo será retornada com o tipo `T`;
- `unsigned size(unsigned)`: método que retorna o tamanho do bloco da *View*. O parâmetro `unsigned` indica de qual dimensão o tamanho deve ser considerado. Dessa forma, aceita os valores 0 e 1, sendo 1 aplicável apenas caso a *View* represente uma matriz. Ele pode ser omitido, considerando assim a dimensão 0. Somente é possível chamá-lo dentro do `Functor` onde será executada a função paralela.

3.2.2.3 Classe `Functor`

A classe `Functor` precisa ser herdada pela classe que irá mapear a função que deverá ser executada no laço ou redução paralela. A partir disso, estarão disponíveis os seguintes métodos:

- `void register_data(View...)`: método que registra as *Views* para a classe `Functor`. Deve ser chamado no construtor e no operador de cópia da classe que realiza o mapeamento da função paralela. Como parâmetro, pode receber de 1 a *n* `View`'s;
- `void clear_data()`: método que limpa as *Views* mapeadas para a classe `Functor`. Deve ser chamado no operador de cópia antes da chamada do método `register_data()`;
- `void remove_data()`: método que remove as *Views* mapeadas para a classe `Functor`. Deve ser chamado de forma externa ao `Functor`, após a execução do mesmo pelo laço ou redução paralela;

- `T* reduction_var<T>()`: método que retorna o ponteiro da variável de redução dentro da função paralela do `Functor`. O tipo dessa variável deve ser passado como *template*.

3.2.2.4 Outras classes

Além das classes principais destacadas nos tópicos anteriores, a HPSM é composta por algumas classes secundárias que normalmente são utilizadas com parâmetros para as chamadas dos métodos das classes principais.

• **Classe** `index<N>`:

A classe `index<N>` possui vários métodos úteis para acesso aos dados das *Views* durante a execução da função paralela. O *template* `N` indica o número de dimensões do laço, podendo receber os valores 1 (laço para o processamento de vetores) ou 2 (laço de processamento de vetores e matrizes). Seus método são:

- `unsigned block()`: método que retorna o número do bloco que está sendo processado;
- `unsigned block_dim(unsigned)`: método que retorna o número do bloco que está sendo processado considerando o número da dimensão recebida como parâmetro. Ele aceita os valores 0 e 1, sendo 1 aplicável apenas caso o laço seja de duas dimensões. Pode ser omitido, assumindo assim o valor 0;
- `unsigned block_qtd()`: método que retorna o número total de blocos que foram ou serão processados pelo laço;
- `unsigned block_qtd_dim(unsigned)`: método que retorna o número total de blocos que foram ou serão processados pelo laço considerando o número da dimensão recebida como parâmetro. Ele aceita os valores 0 e 1, sendo 1 aplicável apenas caso o laço seja de duas dimensões. Pode ser omitido, assumindo assim o valor 0;
- `unsigned size(unsigned)`: método que retorna a quantidade de índices que existem em cada uma das dimensões do laço. A dimensão é recebida por parâmetro, aceitando os valores 0 e 1 (1 é aplicável apenas caso o laço seja de duas dimensões). Se omitido, o parâmetro assume o valor 0;
- `unsigned operator()(unsigned)`: operador de aplicação que retorna o valor atual do índice durante o processamento do laço. O parâmetro deve ser utilizado para indicar a dimensão do índice desejado, aceitando os valores 0 e 1 (1 é aplicável apenas caso o laço seja de duas dimensões). Se omitido, o parâmetro assume o valor 0;

• **Classe** `range<N>`:

A classe `range<N>` é utilizada para indicar o início e o fim de um laço paralelo. As funções `parallel_for` e `parallel_reduce` requerem que um objeto do tipo `range` seja passado a elas. O *template* `N` indica o número de dimensões do *range*, podendo receber os valores 1 (para laços que irão processar apenas vetores) ou 2 (para laços que irão processar vetores e matrizes). Seus método construtores são:

- `range<1>(unsigned, unsigned, BlockTile)`: construtor para um `range` de uma dimensão. Os dois parâmetros do tipo `unsigned` recebem os valores de início e fim de um intervalo. O terceiro parâmetro, do tipo `BlockTile`, indica a forma como os blocos serão escalonados para processamento caso as *Views* possuam particionamentos distintos. Ele pode ser omitido, assumindo neste caso o valor `Intercalary`;
- `range<2>(unsigned, unsigned, unsigned , unsigned, BlockTile)`: construtor para um `range` de duas dimensões. Os quatro parâmetros do tipo `unsigned` recebem os valores dos intervalos, sendo que os dois primeiros determinam o início e fim da primeira dimensão e os dois últimos o início e fim da segunda dimensão. O quinto parâmetro, do tipo `BlockTile`, possui a mesma funcionalidade descrita no item anterior;

3.2.2.5 Funções atômicas

A HPSM também é composta por algumas outras funções, como as descritas nos tópicos abaixo, que são responsáveis por executar operações atômicas para GPUs. Caso elas sejam processadas por uma CPU a instrução atômica é desconsidera, visto que na execução em CPUs o acesso a uma variável por diferentes *threads* é controlado pelos *back-ends*.

- `void atomic_add<T>(T*, T)`: função de operação atômica de adição, onde o segundo parâmetro do tipo `T` é adicionado ao valor presente no ponteiro do primeiro parâmetro;
- `void atomic_multi<T>(T*, T)`: função de operação atômica de multiplicação, onde o segundo parâmetro do tipo `T` é multiplicado ao valor presente no ponteiro do primeiro parâmetro. Não possui suporte para GPU, porém a API a disponibiliza por padronização;
- `void atomic_max<T>(T*, T)`: função de operação atômica de máximo, onde o segundo parâmetro do tipo `T` é comparado com o valor presente no ponteiro do primeiro parâmetro. Caso ele seja maior, o ponteiro do primeiro parâmetro recebe o valor do segundo;
- `void atomic_min<T>(T*, T)`: função de operação atômica de mínimo, onde o segundo parâmetro do tipo `T` é comparado com o valor presente no ponteiro do primeiro parâmetro. Caso ele seja menor, o ponteiro do primeiro parâmetro recebe o valor do segundo;

3.2.2.6 Macros e constantes

As macros e constantes da HPSM são utilizadas para definir o comportamento do processamento paralelo. As mesmas estão descritas abaixo:

- `PARALLEL_FUNCTION`: macro de compilação. Deve estar presente na declaração de todas as funções que serão processadas de forma paralela pela API;
- `AccessMode`: grupo de constantes de modo de acesso utilizadas para controlar as transferências de dados implícitas entre a memória principal e a memória das GPUs:
  - `AccessMode::In`: indica que os dados são de entrada, podendo ser acessados apenas para leitura;
  - `AccessMode::Out`: indica que os dados são de saída, podendo ser acessados apenas para escrita;
  - `AccessMode::InOut`: indica que os dados são de entrada e saída, podendo ser acessados para leitura e escrita;
- `PartitionMode`: grupo de constantes com as formas de particionamento das matrizes. Mais detalhes podem ser obtidos na tópico 3.1.2 (página 33):
  - `PartitionMode::Matrix_Horiz`: particiona uma matriz pelas suas colunas;
  - `PartitionMode::Matrix_Vert`: particiona uma matriz pelas suas linhas;
  - `PartitionMode::Matrix_Vert_Horiz`: particiona uma matriz pelas suas linhas e colunas;
- `ReduxMode`: grupo de constantes com os modos de redução utilizados na definição de uma redução paralela:
  - `ReduxMode::Sum`: indica que a redução retornará uma soma de valores;
  - `ReduxMode::Multi`: indica que a redução retornará uma multiplicação de valores;
  - `ReduxMode::Max`: indica que a redução retornará o maior valor;
  - `ReduxMode::Min`: indica que a redução retornará o menor valor;
- `BlockTile`: grupo de constantes com as formas de escalonamento dos blocos durante o processamento. Deve ser adicionado um valor para cada *range* de laço ou redução paralela. Mais detalhes podem ser obtidos no tópico 3.1.3 (página 35):
  - `BlockTile::Intercalary`: escalona os blocos de forma intercalada;
  - `BlockTile::Sequentially`: escalona os blocos de forma sequencial.

3.3 COMPARAÇÃO DA HPSM COM AS INTERFACES RELACIONADAS

Esta seção realiza uma comparação da HPSM com as interfaces relacionadas descritas na seção 2.4 (página 27). A Tabela 3.1 apresenta uma sumarização das características da nova API em analogia a estas interfaces. A partir dela, percebe-se que a HPSM é a única que possui suporte ao processamento de laços paralelos simultaneamente em CPUs e GPUs. Dessa forma, esta é a sua principal diferença em relação às demais ferramentas.

Tabela 3.1 – Características da HPSM e das interfaces relacionadas.

Característica	**Ferramenta/API**									
	HPSM	Kaapi++	StarPU	OmpSs	OpenMP 4.0	Kokkos	C++ AMP	Thrust	Phalanx	HPX
multi-CPU	✓	✓	✓	✓	✓	✓			✓	✓
GPU	✓	✓	✓	✓	✓	✓	✓	✓	✓	
multi-GPU	✓	✓	✓	✓					✓	
Processamento CPU+GPU	✓	✓	✓	✓						
Laços paralelos	✓			✓	✓	✓	✓	✓		
Laços paralelos CPU+GPU	✓									
Portabilidade do código	✓				✓	✓	✓	✓	✓	
Transf. de dados implícitas	✓	✓	✓	✓	✓				✓	
Tarefas assíncronas		✓	✓	✓	✓				✓	✓
Processamento distribuído			✓	✓					✓	✓

Fonte: Próprio autor.

Considerando as interfaces Kaapi++, StarPU e OmpSs, que suportam multi-CPU e multi-GPU, a diferença para a HPSM consiste na representação do paralelismo, que precisa ser feito por meio de tarefas assíncronas. Somado a isso, também há necessidade de se escrever duas versões de código para explorar ambas arquiteturas, ou seja, um para CPU e um para GPU. Apesar de suportar a portabilidade de código, a interface Phalanx não permite execuções CPU+GPU e também requer o uso de tarefas assíncronas.

Dentre as interfaces destacadas na Tabela 3.1, a Kokkos é a que mais assemelha-se com a HPSM. Destas semelhanças, pode-se destacar o uso de laços e reduções paralelas, o mapeamento dos dados a serem processados em paralelo através de *Views* e o mapeamento das rotinas através de *Functors*. Apesar disso, o processamento com a Kokkos é realizado apenas de forma *multicore* ou explorando uma única GPU, visto que não há suporte para multi-GPU e execuções CPU+GPU. Ademais, não é permitida a divisão do trabalho do *kernel* em blocos, de modo que as transferências de memória entre os dispositivos é feita apenas uma vez e engloba os dados na sua totalidade. Ainda, estas transferências precisam ser realizadas de forma explícita.

O C++ AMP é outra ferramenta que possui similaridades com a HPSM, pois sua principal funcionalidade é a facilidade de adaptação de laços para a execução paralela em um acelerador. Ela ainda faz uso de *Views* e *Functors* para o mapeamento de dados e rotinas, respectivamente. No entanto, o que a difere é o processamento paralelo que ocorre em uma única GPU, pois assim como a Kokkos, não há suporte para multi-GPU e CPU+GPU. Apesar de ter suporte ao processamento em forma de blocos, as transferências de dados entre a memória principal e da GPU devem ser realizadas de uma única vez e de forma explícita. A biblioteca Thrust, apesar de ser uma implementação da C++ *Standard Library* em CUDA, possui as mesmas diferenças

em relação à HPSM que as da C++ AMP.

Por fim, as interfaces OpenMP 4.0 e HPX não possuem muitas proximidades com a HPSM. A primeira pois é baseada em diretivas de compilação e não suporta multi-GPU. Já a segunda, apesar de utilizar C++, é voltada ao processamento *multicore* e distribuído sem aceleradores.

Após a comparação da HPSM com as interfaces relacionadas, é possível afirmar que não foram encontradas APIs que permitam a construção de programas com laços paralelos que possam ser processados simultaneamente em múltiplas CPUs e múltiplas GPUs, justificando o objetivo deste estudo.

3.4 ANÁLISE DE ESFORÇO PARA CODIFICAÇÃO UTILIZANDO A HPSM

A presente seção tem por objetivo medir o esforço necessário para codificação de um programa paralelo. Para isso, será utilizada a métrica de *software* de contagem de linhas de código (LOC), conforme descrição e justificativas apresentadas na primeira parte deste tópico (3.4.1). Esta métrica faz sentido neste contexto pois serão comparados programas oriundos de linguagens da família C (C e C++), sendo que desta maneira eles possuem estruturas semelhantes. Na segunda parte desta seção (tópico 3.4.2), serão apresentados os valores obtidos através da métrica LOC para as interfaces HPSM, StarPU e OpenMP.

3.4.1 Métricas de *software*

Métricas de *software* na área da engenharia de *software* são recursos usados para a obtenção de dados e informações do produto (neste caso, o programa) e de seu processo de desenvolvimento. Sem as métricas, é difícil realizar afirmações a respeito da saúde de um programa ou do andamento da sua implementação, além de classificá-lo como bom ou ruim. O uso das métricas traz diversos benefícios, mas mesmo assim, sua aplicação na engenharia de *software* foi e continua sendo negligenciada por muitos projetos. Todavia, isso vem mudando, visto que os programas de computador estão desempenhando papeis de destaque nos mais variados campos, resultando na aplicação de quantidades consideráveis de dinheiro e energia na sua concepção. Assim, as métricas passaram a possuir importância também no contexto da engenharia de *software*, fato que já era verdadeiro e trazia excelentes resultados em outras áreas (FENTON; BIEMAN, 2014).

As métricas são obtidas através de medições. O conceito de medições permite afirmar que elas fazem parte do dia a dia do ser humano, sendo aplicadas nas mais diversas áreas com o intuito de ilustrar atributos que descrevem algo, como por exemplo, a altura de uma pessoa ou o peso de um alimento. Fenton e Bieman (2014) afirmam exatamente isso, definindo medições

como o processo onde números e símbolos são atribuídos para as propriedades de entidades do mundo real a fim de descrevê-las seguindo regras previamente definidas. No contexto da computação, a entidade do mundo real pode ser o programa e uma propriedade o seu código fonte, onde medições podem ser realizadas para auxiliar a definir seu tamanho, complexidade ou mantenibilidade.

No escopo deste trabalho, as métricas de *software* serão utilizadas com o objetivo de realizar comparações do esforço necessário para a implementação de um programa paralelo. De acordo com Jones (2010), o esforço ou custo de um programa tem como tarefa precursora a previsão ou medição do seu tamanho. No entanto, este tamanho não deve considerar apenas a quantidade de código fonte, visto que a codificação é apenas uma das etapas do processo de desenvolvimento e corresponde somente a 40% do esforço total. Muitas outras tarefas precisam ser incluídas para medir o tamanho de um *software*, entre elas, levantamento de requisitos, especificações, documentação, testes e até possíveis correções. Ainda segundo Jones (2010), existem várias técnicas voltadas a aferição do tamanho de uma aplicação, como métricas de caso de uso e pontos por função.

Voltando ao escopo deste estudo, a HPSM é empregada apenas na codificação de um programa paralelo. Neste sentido, métricas destinas a medir o esforço necessário para sua utilização precisam concentrar-se basicamente no código que será escrito pelo programador. Desta maneira, a métrica selecionada para mensurar o esforço necessário para a implementação de um programa foi a de linhas de código, ou simplesmente LOC (do inglês, *Lines of Code*). Ela é inclusive a métrica mais aplicada para medir o tamanho de um código fonte (FENTON; BIEMAN, 2014).

Datada do início da década de 1960, a métrica LOC possui algumas desvantagens, sendo que em muitos casos não é aconselhada a sua utilização. Jones (2010) inclusive é um autor que a considera prejudicial, citando fatores como a falta de um padrão para a contagem da linhas, dificuldade de comparações entre códigos de diferentes linguagens de programação (que possuem estruturas diferentes) e que a métrica não pode ser utilizada para medir outras atividades do processo de desenvolvimento de um *software*. Pensando no processo como um todo, as afirmações do autor fazem sentido. Entretanto, existe um contra-ponto que pode ser explorado. De acordo com Fenton e Bieman (2014), apesar de LOC não ser indicada para todas as situações, a métrica pode ser útil quando os demais atributos do contexto onde um código está inserido são similares. Ainda segundo Fenton e Bieman (2014), como comparação, um código de 100 mil linhas será mais difícil de ser testado, mantido e mais suscetível a falhas do que um de apenas 10 mil. Portanto, seguindo-se alguns critérios, LOC pode ser utilizado para a obtenção de informações importantes sobre o código dos programas.

Para o uso do LOC, um padrão precisa ser definido, ou seja, quais linhas devem ou não ser consideradas durante o procedimento de contagem. De acordo com Jones (2010), uma técnica de contagem pode resultar em um número até cinco vezes maior do que outro obtido através da aplicação de um método diferente. Assim, Fenton e Bieman (2014) definem alguns

destes padrões, conforme apresentado abaixo:

- **NCLOC**: abreviação de *Noncommented Lines of Code*. Este padrão considera na contagem somente as linhas de código efetivas, ou seja, são desconsideradas as linhas em branco e os comentários;
- **CLOC**: abreviação de *Commented Lines of Code*. Este padrão considera apenas as linhas com comentários. Ele pode ser utilizado, por exemplo, para calcular a densidade dos comentários em relação ao total do código fonte;
- **DSI**: abreviação de *Delivered Source Instructions*, um padrão semelhante ao NCLOC, porém ele considera os comandos de programação (ou instruções) para a contagem. Assim, caso um comando esteja em mais de uma linha, ele será contado apenas uma vez. Além disso, caso mais de um comando esteja em uma mesma linha, cada um será contado separadamente;
- **ES**: abreviação de *Executable Statements*. Este padrão também considera os comandos da mesma forma que o DSI. No entanto, ele desconsidera, além dos comentários e linhas em branco, as declarações de dados e cabeçalhos. Portanto, o método ES conta as linhas que são efetivamente executadas.

Para as medições que foram realizadas neste trabalho, optou-se por utilizar os padrões NCLOC e ES, visto que assim será possível ter duas visões de um mesmo código paralelo.

3.4.2 Comparação entre HPSM, StarPU e OpenMP

A metodologia para avaliação do esforço de codificação será comparativa entre a medida apresentada por um fonte escrito através da HPSM, a medida de um fonte codificado através de outra ferramenta multi-CPU e multi-GPU, neste caso a StarPU, e a medida de uma ferramenta voltada a explorar arquiteturas *multicore*, neste caso a OpenMP. A StarPU e OpenMP foram escolhidas pois fazem parte do contexto deste estudo, visto que os *back-ends* da HPSM utilizam seus recursos.

A comparação será feita através de programas para solucionar o problema AXPY. Seus códigos fontes estão disponíveis no anexo A na página 89. Primeiramente, a Figura A.1 detalha o código AXPY implementado com OpenMP. Na sequência, a Figura A.2 exibe o fonte da aplicação utilizando a HPSM. O código do programa StarPU é mais extenso e precisou ser dividido em duas partes, sendo apresentado nas Figuras A.3 e A.4. De acordo com o mencionado no tópico anterior (3.4.1), a métrica LOC irá utilizar os padrões NCLOC e ES. Os dados obtidos a partir dos fontes detalhados nas figuras estão sumarizados na Tabela 3.2.

Conforme demonstrado pelos dados obtidos através da métrica LOC, o fonte da HPSM apresenta números inferiores para ambos padrões de contagem em comparação com o da StarPU.

Tabela 3.2 – Métrica LOC do problema AXPY com HPSM, StarPU e OpenMP.

Versão	**Padrão da métrica**	
	NCLOC	ES
HPSM	37	29
StarPU	86	64
OpenMP	15	11

Fonte: Próprio autor.

Com o método NCLOC, houve uma redução de 56,98% no número de linhas do programa, enquanto que com o padrão ES, a contração foi de 54,69%. Contribuem para isso três fatores principais:

- **1)** o código StarPU exige que seja realizado o registro dos dados na *runtime* e seu particionamento, além das configurações referentes ao *codelet* e ao modelo de desempenho. Com a HPSM, realiza-se apenas os registros das estruturas de dados, sendo que os demais passos ficam transparentes para o programador;

- **2)** para executar as tarefas em CPUs e GPUs, a StarPU requer a escrita de duas versões do código, uma CPU (linha 5 da Figura A.3) e outra GPU (linha 15 da Figura A.3). No caso do programa demonstrado ainda houve uma economia de linhas de código, pois não foi necessário codificar um *kernel* CUDA, visto que fez-se uso da função `cublasSaxpy` (linha 21 da Figura A.3). Utilizando a HPSM, o *kernel* da aplicação fica todo contido no objeto *Functor*, pois ela garante a portabilidade do código entre diferentes arquiteturas;

- **3)** a submissão das tarefas para a execução pela *runtime* é outra etapa que consome diversas linhas no código StarPU (linhas 85 a 97 da Figura A.4), enquanto no programa explorando a HPSM há apenas a chamada da rotina que irá executar o laço paralelo.

Em comparação com o fonte OpenMP, o HPSM possui 2,5 e 2,6 vezes mais código de acordo com métodos NLOC e ES, respectivamente. A razão disso é que o programa OpenMP é basicamente o mesmo de uma versão sequencial, onde acrescentou-se apenas uma linha com a diretiva `#pragma omp parallel for`. No entanto, isso traz como requisito a necessidade de suporte de compilação.

Considerando as versões dos códigos para execução em CPU e GPU (HPSM e StarPU), a diferença nos tamanhos sugere que pode-se obter uma redução no esforço necessário para codificá-lo caso seja feita a opção por utilizar a nova API C++. Sugere-se ainda que o código StarPU seja mais complexo, pois como é possível observar nas Figuras A.3 e A.4, é necessário interagir diretamente com as chamadas de baixo nível da *runtime*. Esta complexidade pode contribuir para aumentar ainda mais o esforço necessário para desenvolvimento, fato também agravado pela interação com CUDA ou OpenCL para explorar GPUs. Todavia, o mesmo ganho não é obtido caso a necessidade seja a implementação de um código para explorar arquiteturas *multicore*, visto que neste caso a versão OpenMP requer menos trabalho de codificação.

Como ressalva, é preciso considerar que a StarPU é baseada em tarefas, uma estrutura mais complexa do que os laços paralelos propostos pela HPSM. Isso justifica a necessidade de se escrever mais linhas de código. Entretanto, caso a opção seja para a construção de um programa baseado em um laço paralelo, como é o caso do AXPY, a API torna-se uma alternativa para diminuir o esforço de desenvolvimento de um programa que visa explorar arquiteturas multi-CPU e multi-GPU.

Por fim, podemos destacar também a ferramenta OmpSs, já descrita na seção 2.3.3. Ela, assim como a OpenMP, é baseada em diretivas `#pragma` e provavelmente permite que programas sejam implementados com menos código. Porém, além de depender de um suporte de compilador para interpretar as diretivas, a OmpSs também requer a construção de duas versões de uma tarefa para explorar CPU e GPU, o que pode aumentar a complexidade da codificação.

3.5 CONCLUSÃO

Este capítulo descreveu a HPSM, uma API C++ que é o objeto deste trabalho. Primeiramente, apresentou-se a forma de uso da HPSM para a construção de programas paralelos, demonstrando-se códigos com exemplos das funcionalidades. Em seguida, abordou-se a sua estrutura, onde foram detalhadas as suas classes focando no modelo orientado a *back-ends*. Na sequência, foi exposta uma documentação das rotinas principais da API. Após isso, abordou-se os trabalhos relacionados, onde foram efetuadas comparações para expor as semelhanças e diferenças em relação a este estudo. Desta maneira, foi possível posicioná-lo dentro do estado da arte.

A ultima seção realizou uma análise de esforço de codificação, comparando a HPSM com outras ferramentas que suportam o desenvolvimento de aplicações paralelas, sendo StarPU e OpenMP. Para isso, foi empregada a métrica de contagem de linhas de código (LOC), com suas variações NCLOC e ES. Nos dados apresentados, o uso da API resultou em um programa em torno de 50% menor ao codificado por meio da StarPU, sugerindo que o esforço para a implementação de uma aplicação com laços paralelos para multi-CPU e multi-GPU é menor. Todavia, o mesmo não ocorreu na comparação com um código OpenMP, onde a HPSM requereu um código 2,5 vezes maior. A razão para este fato é o uso de diretivas `#pragma` pelo OpenMP, que fazem que a versão paralela seja igual a versão sequencial, apenas com uma linha a mais contendo a diretiva.

4 RESULTADOS EXPERIMENTAIS

A HPSM foi empregada na implementação das mini-aplicações científicas N-Body, Hotspot e CFD. Elas foram utilizadas em experimentos visando validar o modelo de programação da API. Além disso, os experimentos propõem-se em comprovar a hipótese estabelecida para este estudo, de que aplicações científicas podem valer-se do processamento heterogêneo em multi-CPU e multi-GPU para alcançar um desempenho superior em relação ao uso de apenas aceleradores.

O capítulo está dividido em três seções. Na primeira (4.1) são expostos resultados com o intuito de demonstrar o sobrecusto (*overhead*) da API sobre a versão sequencial. Na seção 4.2, são descritos os experimentos realizados com mini-aplicações científicas. Por fim, a seção 4.3 analisa e discute os resultados experimentais. Mais detalhes sobre os resultados, como dados, *scripts*, rastros e análises, podem ser obtidos no repositório Git[1].

4.1 SOBRECUSTO DA HPSM

A presente seção visa demonstrar o impacto no desempenho da utilização da API para paralelizar um programa. Este impacto, também chamado de sobrecusto, foi medido a partir da execução de aplicações utilizando os *back-ends* OpenMP, Serial e StarPU da HPSM com apenas uma *thread*. A fórmula empregada no cálculo foi $T_1/T_{sequencial}$, onde dividiu-se o tempo alcançado pela versão com a API pelo tempo obtido pela versão sequencial da aplicação, conforme demonstrado no gráfico[2] da Figura 4.1. Nele, caso o valor registrado pelo programa tenha sido maior que 1, o tempo obtido pela versão da API foi superior e portanto, seu uso trouxe custo para a execução.

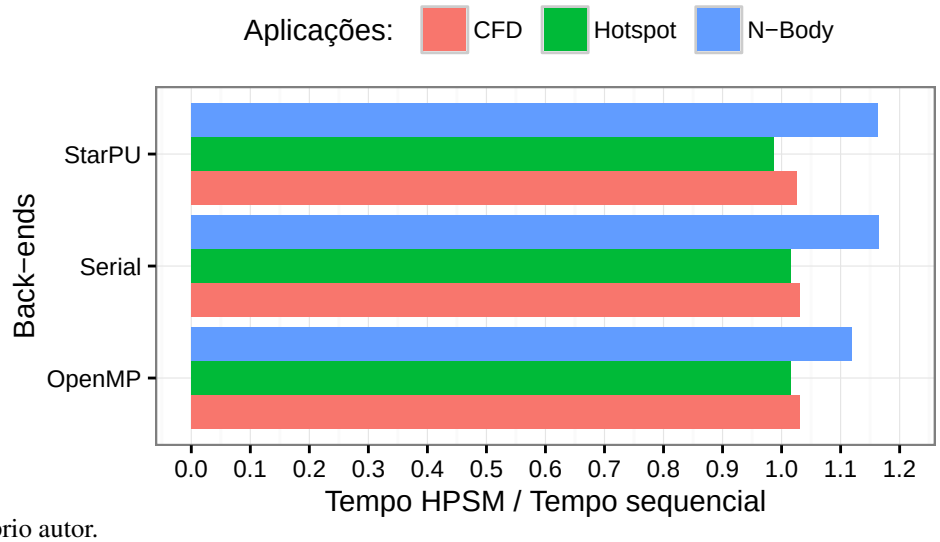
Os dados apresentados na Figura 4.1 indicam que o sobrecusto da API varia de acordo com a aplicação utilizada. Considerando a Hotspot com a qual obteve-se o melhor caso, percebe-se que o sobrecusto é praticamente nulo, com valores de no máximo 1,01. Entretanto, com a N-Body o sobrecusto foi maior, variando de 1,12 (OpenMP) a 1,16 (Serial e StarPU). Neste caso, o uso da HPSM resultou em um acréscimo de até 16,4% nos tempos de execução dos programas.

Os dados de sobrecusto indicam que a HPSM pode impactar no desempenho do programa, sendo que este impacto não é igual para todas as aplicações. A incidência de um custo no uso da API já era esperado, visto que sua camada de classes introduz rotinas que também

[1] Git HPSM: <https://github.com/danidomenico/hpsm>

[2] Gráfico Figura 4.1: para a aplicação CFD utilizou-se o tamanho de 131.072 elementos, blocos de 2.048 e 100 iterações. Para a aplicação Hotspot empregou-se uma matriz de ordem 16.384 com blocos de ordem 1.024 e 5 iterações. Já para a aplicação N-Body fez-se uso de um ambiente de 98.304 partículas, blocos de 2.048 e 5 iterações.

Figura 4.1 – Sobrecusto da HPSM.



Fonte: Próprio autor.

precisam ser processadas.

4.2 EXPERIMENTOS COM MINI-APLICAÇÕES CIENTÍFICAS

Nesta seção, serão descritos os experimentos realizados com mini-aplicações científicas implementadas com a HPSM. Os testes foram realizados a partir de cinco ferramentas no papel de *back-end*, sendo:

- **OpenMP:** *back-end* nativo da API com a compilação do programa realizada diretamente para ele. Foi utilizado para as execuções sem GPUs. O escalonamento aplicado para o OpenMP foi o `dynamic,[tamanho_bloco]`, onde o tamanho do bloco variou para cada aplicação;

- **Kaapi:** por não ser um *back-end* nativo da API, sua versão foi obtida através do programa compilado para o OpenMP. Assim, empregou-se o OpenMP do compilador GCC com a *runtime* do Kaapi chamada LIBKOMP (BROQUEDIS; GAUTIER; DANJEAN, 2012). Visando avaliar o desempenho na arquitetura NUMA, como escalonador foi utilizado roubo de tarefas otimizado para explorá-la, conforme descrito no trabalho de Virouleau et al. (2016);

- **StarPU:** também é um *back-end* nativo da API, sendo utilizado para explorar plataformas heterogêneas compostas por multi-CPU e multi-GPU através da *runtime* StarPU. Para as execuções das versões deste *back-end*, foi empregado o escalonador DMDA (*Deque Model Data Aware*). Ele é similar ao escalonador HEFT, porém considera também o tempo das transferências de dados entre as diferentes memórias;

- **StarPU+OpenMP:** versão gerada a partir da StarPU que possui as tarefas para CPUs paralelizadas com OpenMP. Através da variante StarPU, cada tarefa escalonada para uma CPU é executada por apenas um núcleo do processador, enquanto que nesta ela é processada por diversos núcleos de forma paralela. Deste modo, a diferença de tempo entre as tarefas executadas por GPUs e CPUs tende a diminuir, pois as tarefas alocadas para CPUs podem ser processadas por mais de uma *thread*. Para isso ser possível, foi utilizado o escalonador PHEFT. Mais detalhes sobre ele estão disponíveis na seção 2.3.1 da página 24;

- **StarPU+Kaapi:** versão semelhante à descrita no tópico anterior, todavia utiliza o Kaapi para a paralelização das tarefas escalonadas para CPU. Isto é realizado da mesma forma que na variante do *back-end* Kaapi, ou seja, utilizando a *runtime* LIBKOMP.

As variáveis de ambiente empregadas para as execuções dos programas com os *back-ends* estão detalhadas nos itens a seguir, sendo que os termos `[t]`, `[g]` e `[b]` devem ser considerados como parâmetros a serem substituídos pelo número de *threads*, GPUs e tamanho dos blocos, nesta ordem.

- **OpenMP:** `OMP_SCHEDULE=dynamic,[b] OMP_NUM_THREADS=[t]`

- **Kaapi:** `KAAPI_WSPUSH_INIT_DISTRIB=cyclicnumastrict KAAPI_WSSELECT=hws_N_P KAAPI_WSPUSH=Whws OMP_NUM_THREADS=[t] komp-run`

- **StarPU:** `STARPU_SCHED=dmda STARPU_NCPU=[t] STARPU_NCUDA=[g]`

- **StarPU+OpenMP:** `STARPU_SINGLE_COMBINED_WORKER=1 STARPU_SCHED=pheft STARPU_NCPU=[t] STARPU_NCUDA=[g]`

- **StarPU+Kaapi:** `STARPU_SINGLE_COMBINED_WORKER=1 STARPU_SCHED=pheft STARPU_NCPU=[t] STARPU_NCUDA=[g] KAAPI_WSPUSH_INIT_DISTRIB=cyclicnumastrict KAAPI_WSSELECT=hws_N_P KAAPI_WSPUSH=Whws komp-run`

A plataforma onde os experimentos foram conduzidos é uma máquina NUMA Dell PowerEdge T630 equipada com dois processadores Intel Xeon E5-2697 v3 de 2,60 GHz com 14 núcleos (totalizando 28 *cores*) e 256 GB de memória RAM. O computador também possui quatro GPUs NVIDIA Titan X com 3.072 CUDA *cores*, 1.000 MHz de frequência e 12 GB de RAM DDR5. Em relação ao *software*, o ambiente é GNU/Linux Debian 8 (codinome "Jessie") de 64 bits. O compilador GCC versão 4.9.3 foi utilizado para todas as compilações, exceto para as do *back-end* Kaapi, onde empregou-se a 5.4.0. Em ambas versões do GCC aplicou-se a *flag* de otimização `-O3`. As versões das ferramentas de programação foram CUDA 7.5, StarPU 1.2.0 e XKaapi do *branch public/europar2016* do repositório Git[3].

[3]Git XKaapi: <https://scm.gforge.inria.fr/anonscm/git/kaapi/xkaapi.git>

Os experimentos realizados para verificar o desempenho alcançado por cada uma das mini-aplicações científicas foram:

- **Escalabilidade:** tem como objetivo avaliar o desempenho à medida que mais recursos são adicionados para o processamento. Para isso, são executados testes variando o número de *threads*, GPUs e GPUs+*threads*.

- **Máxima configuração:** tem como propósito verificar se os ganhos de desempenho são mantidos quando há redução ou aumento da carga de dados. Sempre são utilizadas duas entradas menores e duas maiores que a empregada no experimento de escalabilidade. Os resultados considerando somente GPUs são representados pela série StarPU_GPU, sendo eles a base para comparar com os resultados obtidos pelas execuções com GPUs+CPUs. Para os testes com GPUs+CPUs, onde são empregados os *back-ends* StarPU, StarPU+ OpenMP e StarPU+Kaapi, as configurações máximas são: 1GPU+27CPUs, 2GPUs+ 26CPUs, 3GPUs+25CPUs e 4GPUs+24CPUs. O número de CPUs diminui pois a *runtime* StarPU dedica uma CPU para controlar cada GPU da plataforma (AUGONNET et al., 2011). Caso elas sejam utilizadas também para o processamento tende a ocorrer uma queda no desempenho, pois o escalonamento das tarefas para as GPUs pode ser prejudicado;

- **Melhor configuração:** os objetivos e parâmetros deste experimento são os mesmos do máxima configuração. No entanto, como nem sempre é com a configuração máxima que uma aplicação alcança o melhor desempenho, optou-se por realizar estes testes a partir dos melhores resultados obtidos durante o experimento de escalabilidade. Assim, a configuração utilizada muda para cada aplicação, sendo esta descrita no tópico onde os resultados são apresentados;

- **Escalabilidade por bloco (StarPU):** este experimento avalia qual é o comportamento das aplicações ao utilizar-se um tamanho de bloco duas vezes maior em relação ao empregado no de escalabilidade. Para isso, fez-se uso das três versões baseadas na *runtime* StarPU (StarPU, StarPU+OpenMP e StarPU+Kaapi). Espera-se que com o bloco maior o desempenho das execuções com GPUs melhore. Porém, como o grão das tarefas ficará maior, tende a haver prejuízo no processamento das CPUs. Neste cenário, o uso de tarefas paralelas para CPUs (versões StarPU+OpenMP e StarPU+Kaapi) também terá sua eficiência analisada, pois estima-se que o uso de um grão maior melhore seu desempenho.

Nos tópicos 4.2.1, 4.2.2 e 4.2.3, estão apresentados os resultados dos experimentos para cada uma das três aplicações utilizadas, sendo N-Body, Hotspot e CFD. Todos os dados presentes nos gráficos foram obtidos a partir de uma média de no mínimo 30 execuções coletadas de forma aleatória. O *speedup* foi calculado por meio da fórmula $T_{sequencial}/T_P$, onde dividiu-se o tempo sequencial da aplicação pelo tempo alcançado pelo programa implementado com a

HPSM. A margem de erro é indicada pelas barras verticais pretas nos gráficos desenhadas sobre os pontos, sendo calculada para um intervalo de confiança de 95% utilizando a distribuição *t* de Student.

4.2.1 N-Body

A simulação de N-Body calcula a evolução de um sistema de corpos (ou partículas) que interagem entre si. Estas interações ocorrem em razão da influência que um corpo exerce sobre os outros, considerando para isso variáveis como distância e aceleração. A evolução se dá pela mudança na posição de cada corpo, que é definida pelas coordenadas X, Y e Z.

A versão utilizada nestes experimentos foi adaptada da disponível no repositório do Barcelona Supercomputing Center[4]. Nesta adaptação, as posições dos corpos são calculadas a partir de dois vetores, sendo um de entrada e um de saída. Isto resulta na necessidade de alteração no particionamento de dados dos vetores ao fim de cada iteração, pois apenas o vetor de saída é divido em blocos e é preciso inverter os vetores após finalizar uma iteração. Mais detalhes sobre esta implementação estão disponíveis no apêndice B (página 85).

Escalabilidade

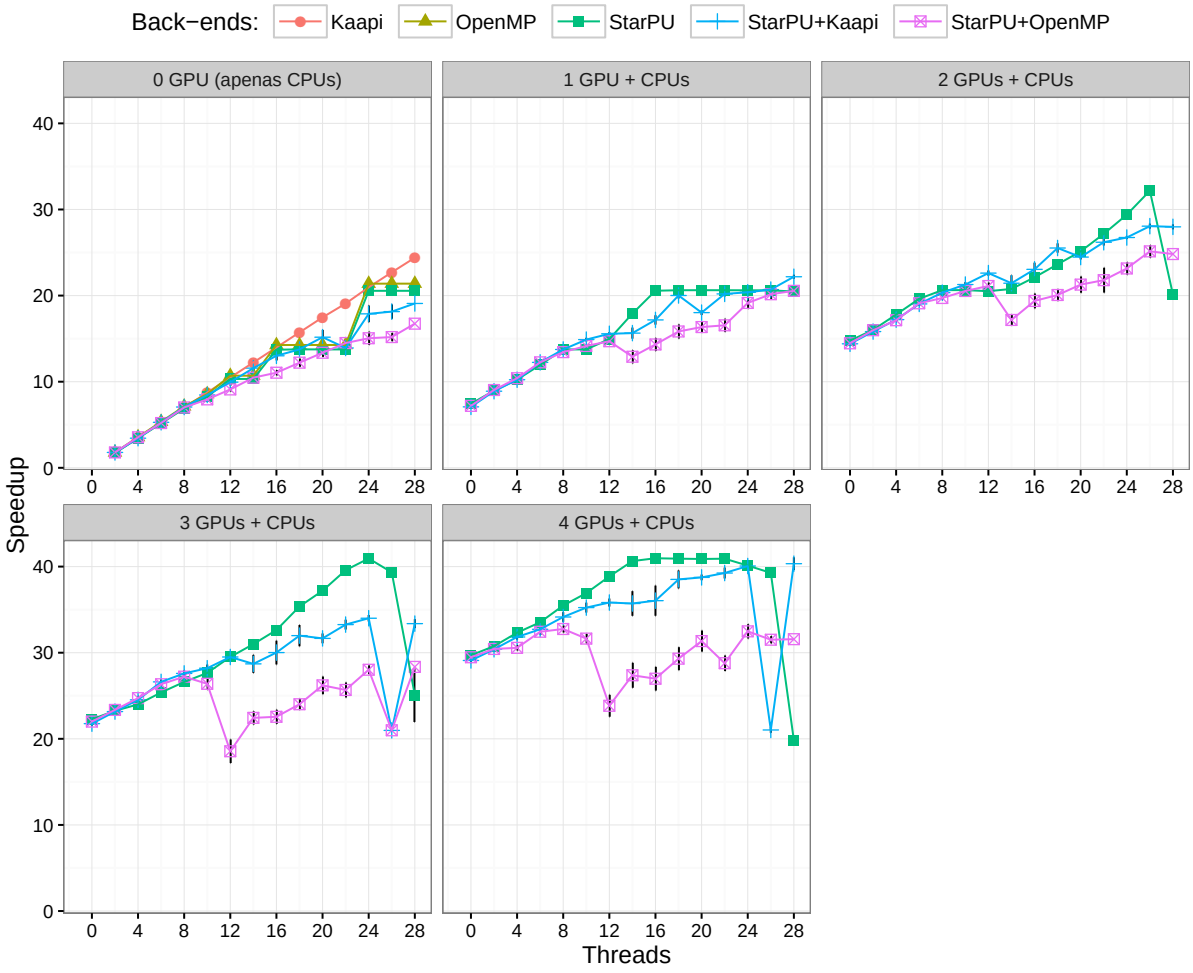
A Figura 4.2 exibe os resultados obtidos para os cinco *back-ends* do primeiro experimento com a N-Body. Nela é possível observar que a aplicação escalou para todos os *back-ends* quando foram empregadas apenas CPUs para o processamento. Neste cenário, o melhor desempenho foi obtido nas execuções com o Kaapi (aceleração de 24,4 com 28 *threads*). Os *back-ends* OpenMP e StarPU alcançaram *speedup* com 28 *threads* de 21,4 e 20,6, respectivamente. No entanto, em ambas versões ocorreu uma estabilização nos ganhos entre 16 e 22 e entre 24 e 28 *threads*. Já as execuções com StarPU+OpenMP e StarPU+Kaapi ficaram abaixo das demais, atingindo acelerações máximas de 16,7 e 18,8 também com 28 *threads*.

Nas execuções com somente GPUs (0 *thread*) percebem-se ganhos, com *speedup* de 7,4 com 1 GPU e de 29,7 com 4 GPUs. Combinando GPUs e CPUs, os melhores resultados foram alcançados com o *back-end* StarPU, sendo que a aceleração máxima foi obtida combinando 3 GPUs com 24 *threads* (40,9). Este valor foi semelhante ao registrado com 4 GPUs entre 16 e 22 *threads*, o que leva a percepção de um saturamento na combinação de CPUs com 4 GPUs. Saturamento também ocorreu com 1 GPU a partir de 16 *threads*. Há ainda uma queda de desempenho com 28, 26 e 24 *threads* para 2, 3 e 4 GPUs, nesta ordem.

Os experimentos combinando StarPU+OpenMP e StarPU+Kaapi também resultaram em ganhos, no entanto, eles foram menores que os obtidos apenas com StarPU. Com StarPU+OpenMP ocorreu uma parada no crescimento do *speedup* entre 10 e 12 *threads*. A aceleração

[4]Barcelona Supercomputing Center: <https://pm.bsc.es/projects/bar/wiki/Applications>

Figura 4.2 – N-Body: escalabilidade variando GPUs e *threads*. Partículas: 98.304. Bloco: 2.048. Iterações: 5.



Fonte: Próprio autor.

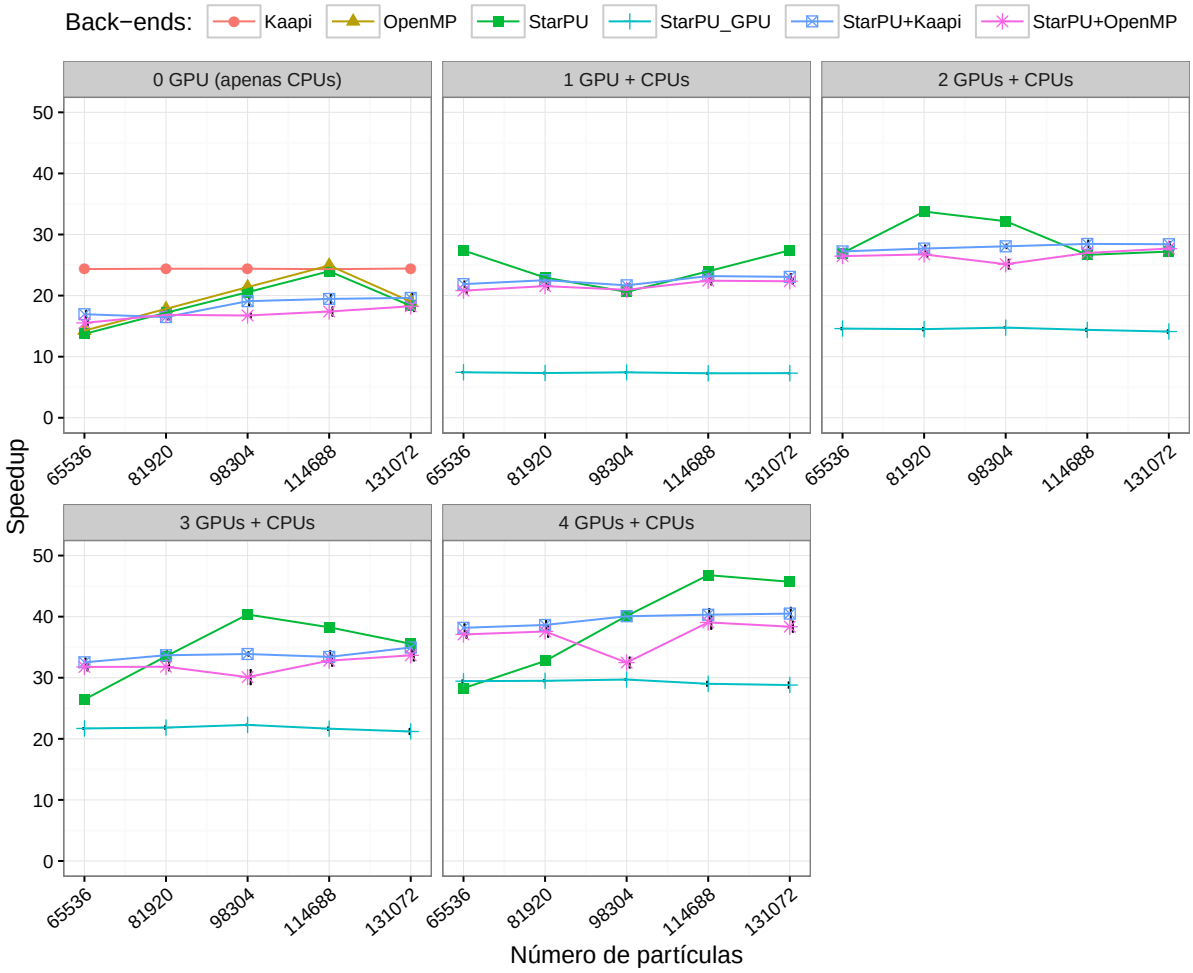
máxima com StarPU+OpenMP foi de 32,7 (4GPUs+8CPUs). Já com StarPU+Kaapi, o máximo foi de 40,2 (4GPUs+28CPUs), onde aconteceram picos para 3 e 4 GPUs com 28 *threads*.

Máxima configuração

O segundo experimento com a aplicação N-Body considerou a configuração máxima permitida pela plataforma variando-se os tamanhos da entrada. A Figura 4.3 exibe os resultados obtidos. Em seu primeiro quadro onde utilizou-se somente CPUs, constatou-se que os ganhos com os *back-ends* Kaapi, StarPU+OpenMP e StarPU+Kaapi permanecem praticamente estáveis para as diferentes entradas. Todavia, variações ocorreram para as versões OpenMP e StarPU. Os piores desempenhos foram computados com entrada de 65.536 partículas com *speedup* de 13,7 (StarPU) e 14,3 (OpenMP), enquanto que os melhores aconteceram para a entrada de 114.688 partículas com acelerações de 24,0 (StarPU) e 24,9 (OpenMP). As diferenças entre os piores e melhores giram em torno de 43%.

Seguindo a análise da Figura 4.3, porém agora considerando as execuções com somente

Figura 4.3 – N-Body: máxima configuração variando o tamanho da entrada. Bloco: 2.048. Iterações: 5.



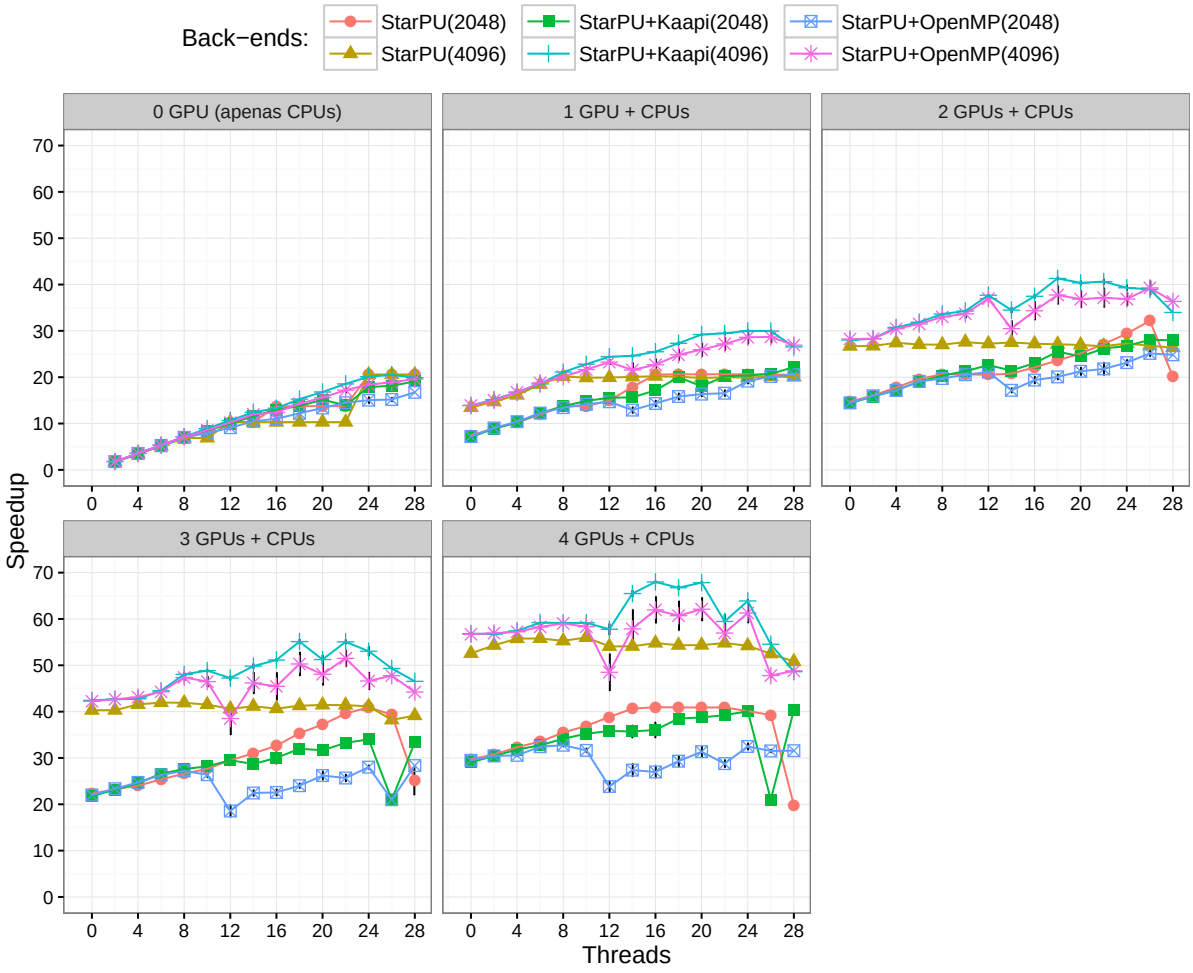
Fonte: Próprio autor.

GPUs (série StarPU_GPU), percebe-se que os ganhos de aceleração mantiveram-se constantes para todas as entradas. Isto também ocorreu para as combinações de GPUs+CPUs com os *back-ends* StarPU+OpenMP e StarPU+Kaapi, onde houve variações somente para a entrada de 98.304 com perdas da versão StarPU+OpenMP em relação a StarPU+Kaapi. Nas execuções com StarPU combinando GPUs+CPUs os ganhos foram superiores aos com somente GPUs. A exceção é para 4 GPUs e entrada de 65.536, onde houve uma ligeira perda de 4,08% (de 29,4 para 28,2).

Escalabilidade por bloco (StarPU)

O experimento de melhor configuração que viria na sequência do máxima configuração não precisou ser realizado para o N-Boby, pois as configurações máximas resultaram nos melhores desempenhos. Desta forma, o próximo e último experimento com a presente aplicação é o de escalabilidade com diferentes tamanhos de blocos. Os resultados obtidos estão expostos na Figura 4.4.

Figura 4.4 – N-Body: escalabilidade variando o tamanho do bloco com os *back-ends* da *runtime* StarPU. Partículas: 98.304. Blocos: 2.048 e 4.096. Iterações: 5.



Fonte: Próprio autor.

Levando-se em conta somente CPUs, as variantes StarPU+OpenMP e StarPU+Kaapi melhoraram seus resultados com o bloco maior, enquanto a StarPU permaneceu estável. Esta melhora foi de 16,77% para StarPU+OpenMP e de 5,85% para StarPU+Kaapi.

Nas execuções apenas com GPUs (0 *thread*), percebe-se que os ganhos com bloco de 4.096 foram maiores para todas as versões. Todavia, é possível observar a ocorrência de uma estabilização no desempenho com o bloco de 4.096 na combinação de GPUs+*threads* com o *back-end* StarPU. Os *back-ends* StarPU+OpenMP e StarPU+Kaapi melhoraram a aceleração com o grão maior (4.096) e superaram a versão StarPU. Com a variante StarPU+Kaapi obteve-se o melhor desempenho com o bloco de 4.096, uma aceleração de 68,0 para 4GPUs+16CPUs. Já com a versão StarPU+OpenMP aconteceram quedas que interromperam os ganhos no intervalo de 10 a 12 CPUs.

4.2.2 Hotspot

A aplicação Hotspot é uma ferramenta de simulação térmica empregada para estimar a temperatura de um processador baseando-se na sua arquitetura e em medições de energia também simuladas (CHE et al., 2009). A entrada da simulação é uma matriz que representa as dimensões do processador. Para estes experimentos, adaptou-se a versão disponível na *suite* Rodinia[5].

A simulação Hotspot é calculada a partir de duas matrizes que possuem as temperaturas de cada área do processador, sendo uma de entrada e uma de saída. A cada iteração é necessário alterar a forma de particionamento delas, pois é preciso inverter as matrizes de entrada e saída sendo que apenas a matriz de saída é dividida em blocos. Mais detalhes da implementação da aplicação com a HPSM podem ser obtidos no apêndice B (página 86).

Escalabilidade

A Figura 4.5 apresenta os resultados dos experimentos de escalabilidade com Hotspot. No primeiro quadro a esquerda onde empregou-se apenas CPUs, percebe-se que os *back-ends* OpenMP, Kaapi e StarPU alcançaram resultados semelhantes para todas as variações de *threads*. A aceleração máxima com 28 *threads* foi de 27,6 (OpenMP). Já as execuções com StarPU+ OpenMP e StarPU+Kaapi registraram *speedup* menores com 28 CPUs (18,2 e 23,2).

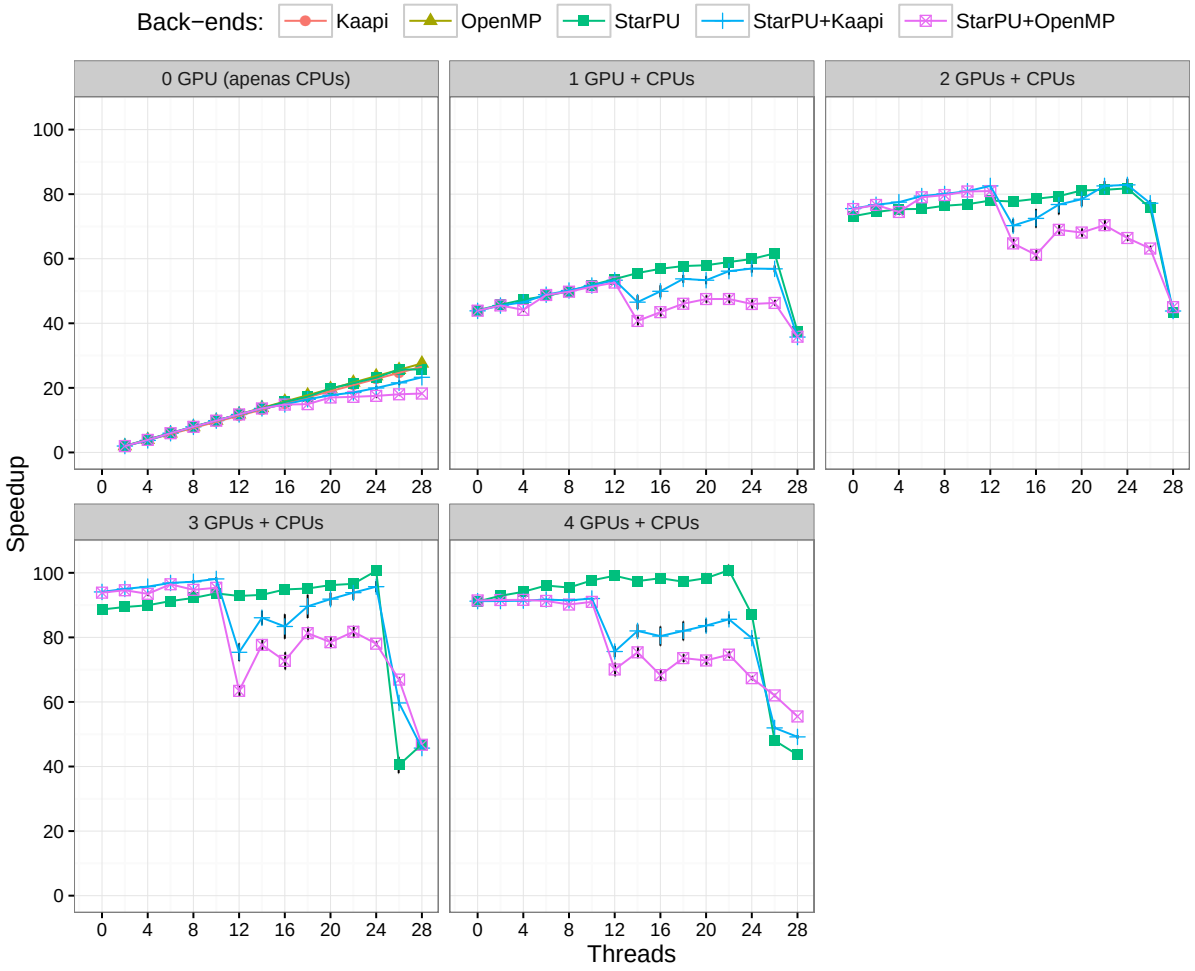
A partir do segundo quadro da Figura 4.5 pode-se visualizar que ocorreram ganhos com apenas GPUs (0 *thread*) para todos os *back-ends*. A versão StarPU alcançou *speedup* máximo de 91,1 com 4 GPUs, mas de 3 para 4 GPUs o ganho é pequeno (apenas 2,82%), indicando assim um saturamento. A combinação de GPUs com CPUs garantiu melhora no *speedup*. A versão StarPU teve a melhor escalabilidade, sendo ela constante e progressiva para 1, 2 e 3 GPUs. Com 4 GPUs é nítido a ocorrência de um saturamento nos ganhos a partir de 12 *threads*. Mesmo assim, a maior aceleração de 100,6 foi alcançada com 4GPUs+22CPUs. Para as variantes StarPU+OpenMP e StarPU+Kaapi o ponto de destaque foram as quedas que ocorrem a partir de 10 e 12 *threads*. Por fim, as execuções com 28 *threads* para 1 GPU, a partir de 26 *threads* para 2 GPUs e a partir 24 *threads* para 3 e 4 GPUs resultaram em redução da aceleração para todas as versões.

Máxima configuração

Os resultados do experimento de máxima configuração estão apresentados na Figura 4.6. No cenário com somente CPUs, a variação do tamanho das entradas não implicou em mudanças significativas nas acelerações de cada *back-end*. A exemplo do que já havia sido constatado no teste de escalabilidade, as versões StarPU+OpenMP e StarPU+Kaapi atingiram um desempenho

[5]Rodidina Benchmark Suite: <https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia: Accelerating_Compute-Intensive_Applications_with_Accelerators>

Figura 4.5 – Hotspot: escalabilidade variando GPUs e *threads*. Tamanho: 16.384x16.384. Bloco: 1.024x1.024. Iteracões: 5.



Fonte: Próprio autor.

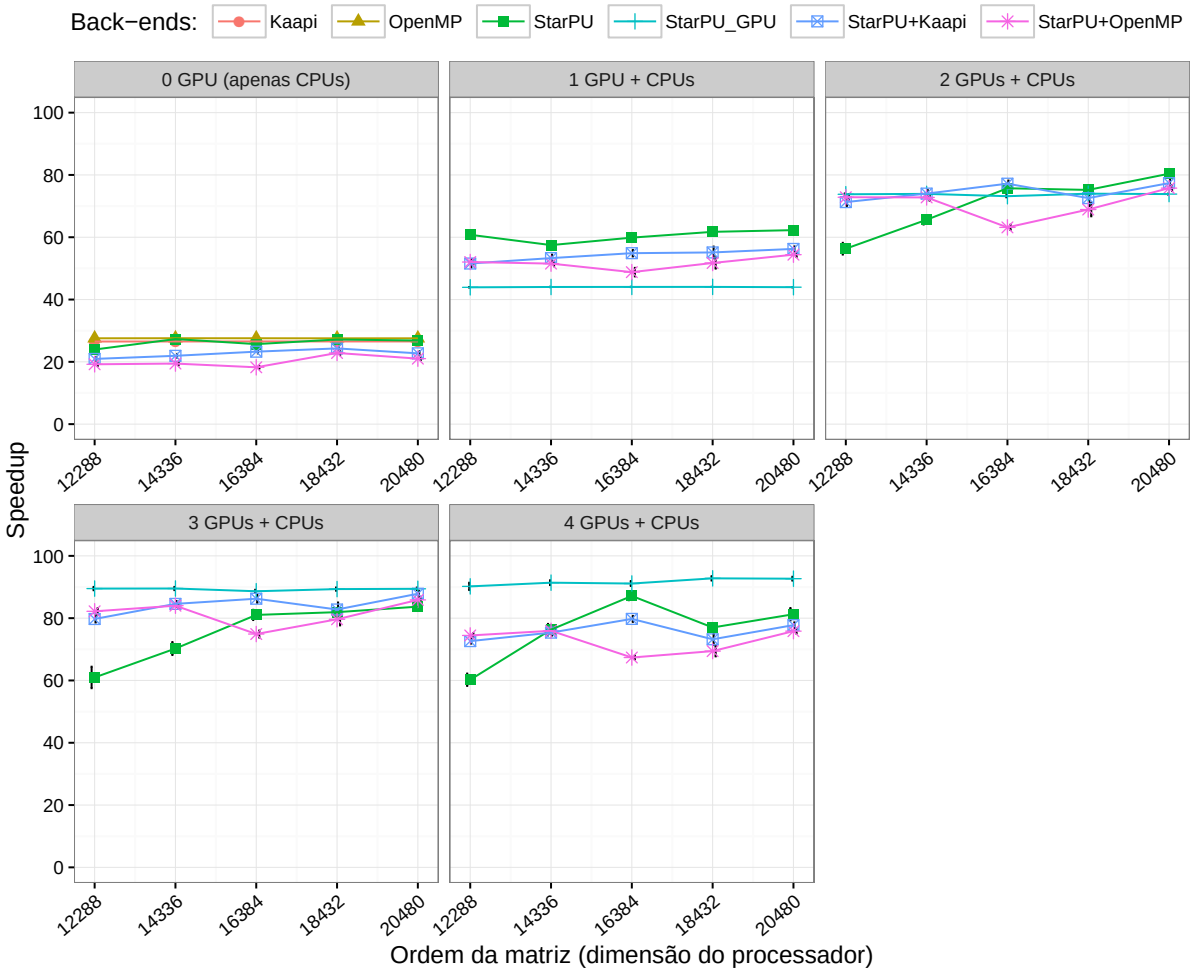
inferior às demais.

As execuções com somente GPUs (série StarPU_GPU) da Figura 4.6 indicam um desempenho uniforme para todos os tamanhos de entrada, com estabilização nos ganhos a partir de 3 GPUs. Os resultados combinando GPUs e CPUs apontam duas situações. Com 1 GPU, houve melhoria em relação às acelerações obtidas por somente GPUs. Já com 2, 3 e 4 GPUs, os ganhos foram iguais ou inferiores aos com apenas GPUs, ocorrendo também variações no *speedup* das diferentes entradas, principalmente para o *back-end* StarPU. Com 4 GPUs, por exemplo, a aceleração foi de 59,8 para a entrada de ordem 12.288, passou para 87,0 com a matriz de ordem 16.384 (variação de 45,45%), voltando a cair para 80,9 na entrada de ordem 20.480 (variação de 7,01%).

Melhor configuração

Conforme dados obtidos através do experimento de escalabilidade (gráfico da Figura 4.5), o melhor configuração foi executado empregando as seguintes combinações:

Figura 4.6 – Hotspot: máxima configuração variando o tamanho da entrada. Bloco: 1.024x1.024. Iterações: 5.



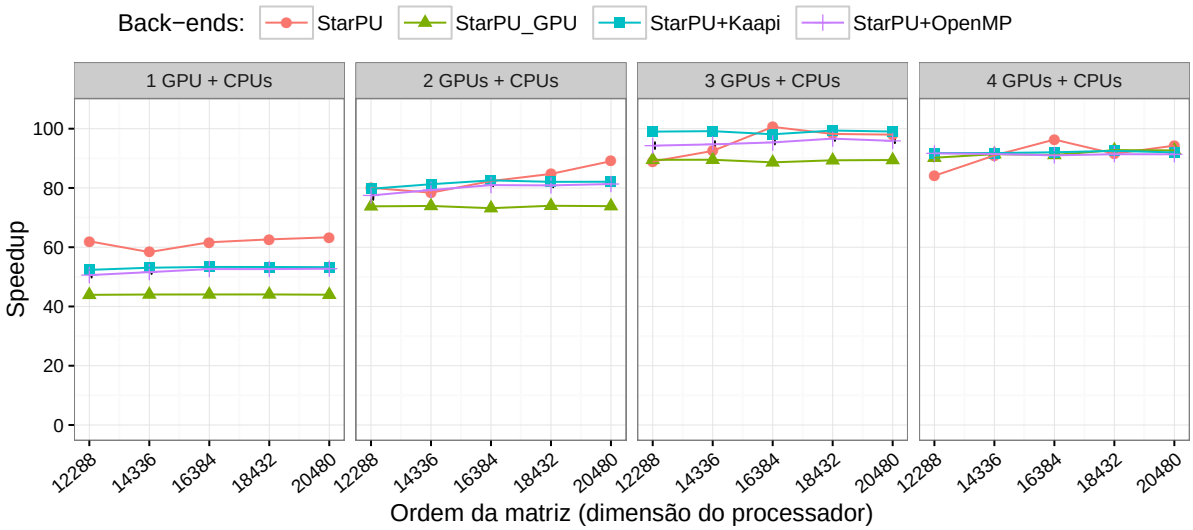
Fonte: Próprio autor.

- ***Back-end* StarPU**: 1GPU+26CPUs, 2GPUs+25CPUs, 3GPUs+24CPUs e 4GPUs+23CPUs;
- ***Back-end* StarPU+OpenMP**: 1GPU+12CPUs, 2GPUs+12CPUs, 3GPUs+10CPUs e 4GPUs+10CPUs;
- ***Back-end* StarPU+Kaapi**: 1GPU+12CPUs, 2GPUs+12CPUs, 3GPUs+10CPUs e 4GPUs+10CPUs.

No gráfico da Figura 4.7, os dados com somente CPUs foram omitidos, pois contém os mesmos resultados do experimento de máxima configuração. Também contém os mesmos resultados as execuções com somente GPUs, representados pela série StarPU_GPU. Nas combinações de GPUs e CPUs, os desempenhos alcançados por todos os *back-ends* foram iguais ou superiores aos obtidos utilizando apenas GPUs nos cenários com 1, 2 e 3 GPUs. Já com 4 GPUs, o gráfico indica um desempenho praticamente constante e próximo ao registrado com apenas GPUs para todas versões. Este comportamento também apareceu nos experimentos anteriores

(Figuras 4.5 e 4.6) e aponta um saturamento com o número máximo de GPUs.

Figura 4.7 – Hotspot: melhor configuração variando o tamanho da entrada. Bloco: 1.024x1.024. Iterações: 5.



Fonte: Próprio autor.

Escalabilidade por bloco (StarPU)

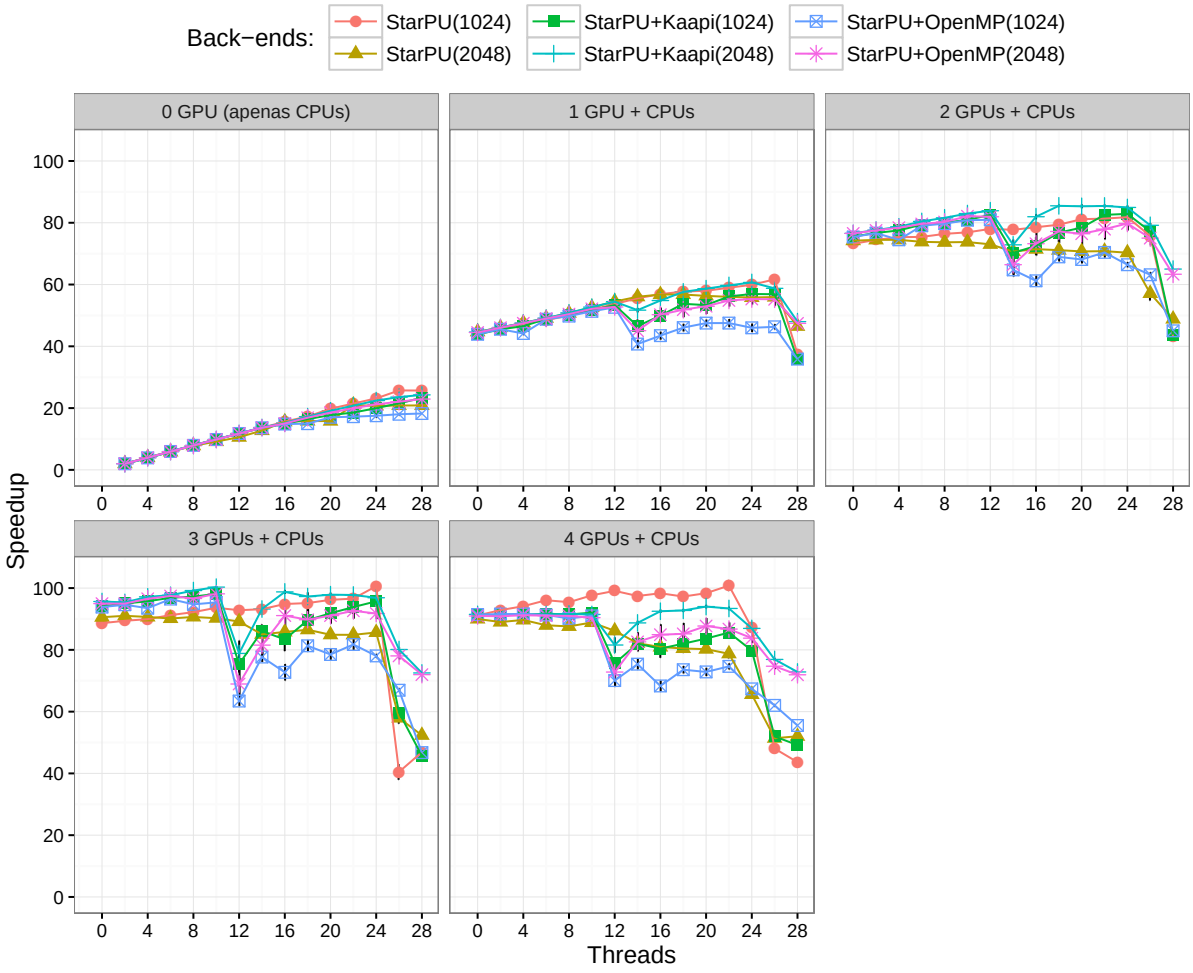
O último experimento com a aplicação Hotspot foi o de escalabilidade com diferentes tamanhos de blocos a partir dos *back-ends* que utilizam a *runtime* StarPU. Estes resultados estão apresentados na Figura 4.8. Considerando apenas CPUs, o melhor desempenho foi obtido pela versão StarPU com bloco de ordem 1.024 e 28 *threads* (*speedup* de 25,7). O uso do bloco maior (ordem 2.048) melhorou a aceleração apenas para os *back-ends* StarPU+OpenMP (de 18,2 para 22,8) e StarPU+Kaapi (de 23,2 para 24,3).

Na Figura 4.8, também percebe-se que diferentemente do que ocorreu com a aplicação N-Body, o uso de um bloco maior (2.048) não resultou em ganhos utilizando somente GPUs (0 *thread*), pois os desempenhos foram iguais ou apenas ligeiramente superiores aos alcançados com o bloco menor (1.024). Já para as combinações de GPUs com CPUs, o *back-end* StarPU com bloco menor obteve o melhor desempenho para todas as *threads*. Ele só foi superado pela versão StarPU+Kaapi com o grão maior (2.048) empregando 2 GPUs. Destaca-se também as quedas de aceleração que aconteceram para as versões StarPU+OpenMP e StarPU+Kaapi no intervalo de 10 a 12 *threads*.

4.2.3 CFD

A aplicação CFD possui um algoritmo para solucionar um problema de dinâmica de fluidos. Ele é caracterizado por ser de grade não estruturada e de volume limitado, aplicando

Figura 4.8 – Hotspot: escalabilidade variando o tamanho do bloco com os *back-ends* da *runtime* StarPU. Tamanho: 16.384x16.384. Blocos: 1.024x1.024 e 2.048x2.048. Iterações: 5.



Fonte: Próprio autor.

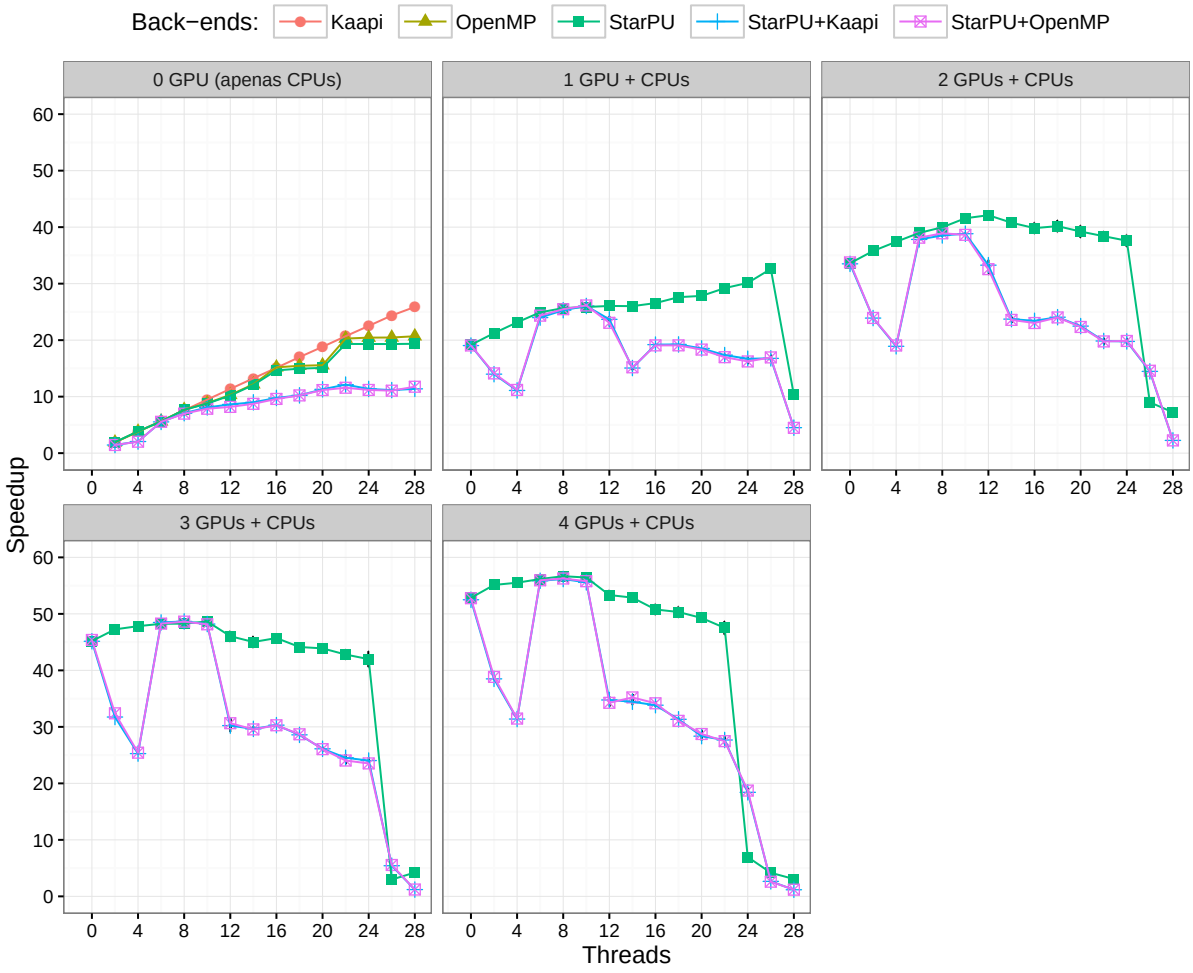
as equações de Euler sobre um fluido compressível em um ambiente de três dimensões (CHE et al., 2010). Mais dados sobre a aplicação e as equações utilizadas podem ser encontrados no trabalho de Corrigan et al. (2009). A versão implementada com a HPSM foi adaptada da disponível na *suite* Rodinia. A entrada utilizada é uma asa de aeromodelo do tipo NACA0012 em fluxo supersônico. Esta aplicação é a mais complexa dentre as presentes neste trabalho, pois a paralelização foi procedida em quatro rotinas, enquanto que para N-Body e Hotspot isto ocorreu em apenas uma.

A cada iteração da CFD são executadas estas quatro rotinas com o objetivo de mensurar o fluxo sobre os elementos (ou variáveis). A rotina que executa o cálculo deste fluxo requer acesso às variáveis dos elementos vizinhos que estão armazenadas em um vetor, o que impede sua divisão em blocos. Neste sentido, ocorre uma mudança na forma de particionamento no vetor com as variáveis, visto que nas outras rotinas ele é segmentado em blocos. Mais detalhes da implementação da CFD com a HPSM estão disponíveis no apêndice B (página 87).

Escalabilidade

Na Figura 4.9 podem ser visualizados os resultados dos experimentos de escalabilidade. Nas execuções com apenas CPUs, o melhor desempenho foi alcançado pelo Kaapi com aceleração de 25,9 com 28 *threads*. As execuções com OpenMP e StarPU apresentaram estabilização nos ganhos entre 16 e 20 e entre 22 e 28 *threads*. Já as versões versões StarPU+OpenMP e StarPU+Kaapi registraram baixo desempenho.

Figura 4.9 – CFD: escalabilidade variando GPUs e *threads*. Tamanho: 131.072. Bloco: 2.048. Iterações: 100.



Fonte: Próprio autor.

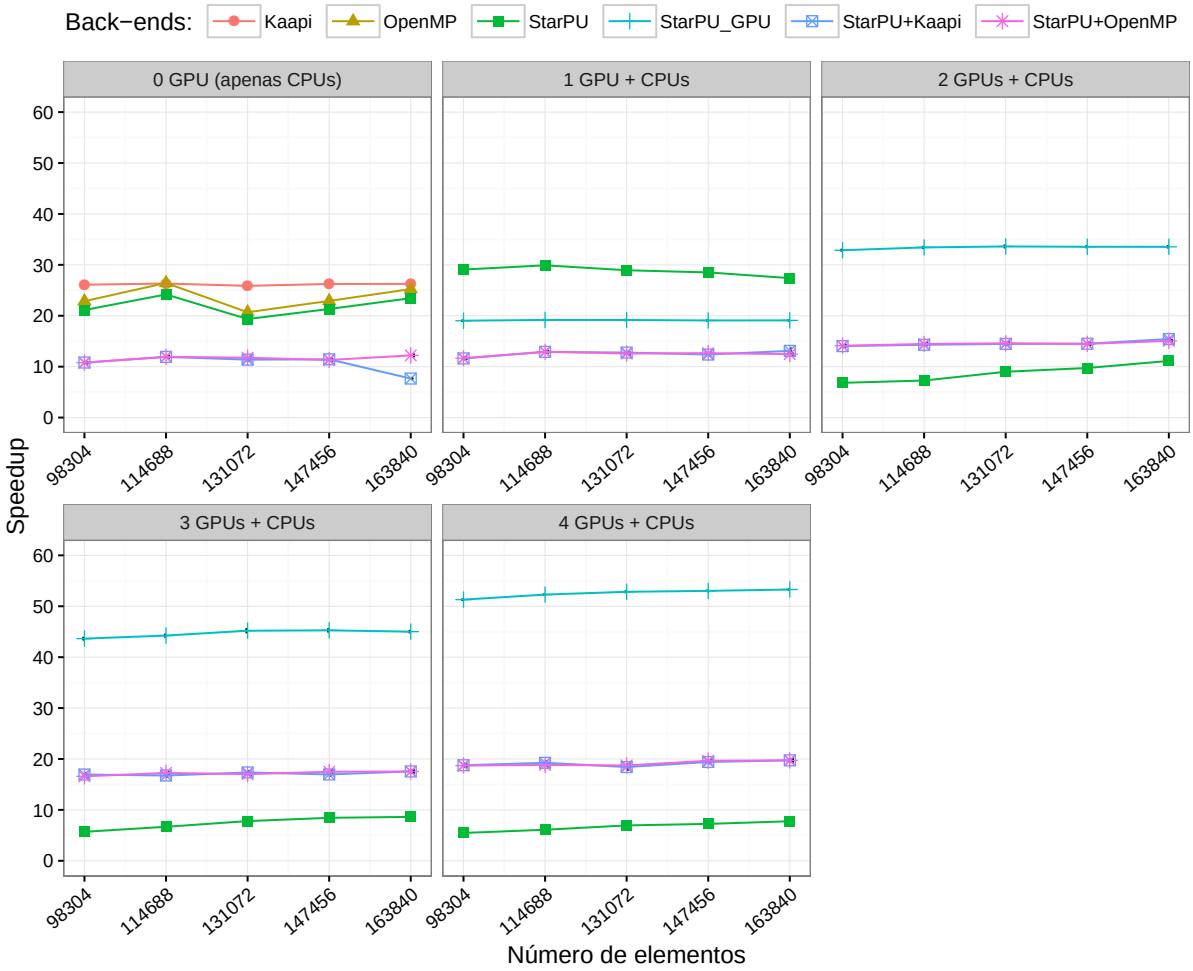
Considerando as execuções somente com GPUs (0 *thread*), a Figura 4.9 indica ganhos, com a aceleração passando de 19,2 com 1 GPU para 52,8 com 4 GPUs. Na combinação de GPUs e CPUs as melhoras de desempenho obtidas foram irregulares, inclusive para o *back-end* StarPU. Com 2, 3 e 4 GPUs, o crescimento manteve-se somente até 10 ou 12 CPUs. Já nas execuções com StarPU+OpenMP e StarPU+Kaapi, que obtiveram praticamente os mesmos resultados, além das quedas que aconteceram com 10 ou 12 *threads*, há quedas também com 2 e 4 CPUs em todos os cenários. Por fim, cabe ressaltar que o melhor desempenho foi alcançado combinando 4GPUs+10CPUs com o *back-end* StarPU (*speedup* de 56,4), valor próximo ao

obtido pelos *back-ends* StarPU+OpenMP e StarPU+Kaapi com a mesma configuração.

Máxima configuração

A Figura 4.10 apresenta os resultados de máxima configuração para a aplicação CFD. No cenário com somente CPUs ocorreram oscilações entre as entradas para os *back-ends* OpenMP e StarPU. As execuções com Kaapi, StarPU+OpenMP e StarPU+Kaapi permaneceram estáveis para todas as entradas, exceto para a de 163.840, onde a versão StarPU+Kaapi apresentou uma queda de 37,70% na aceleração em relação à variante StarPU+OpenMP (de 12,2 para 7,6).

Figura 4.10 – CFD: máxima configuração variando o tamanho da entrada. Bloco: 2.048. Iterações: 100.



Fonte: Próprio autor.

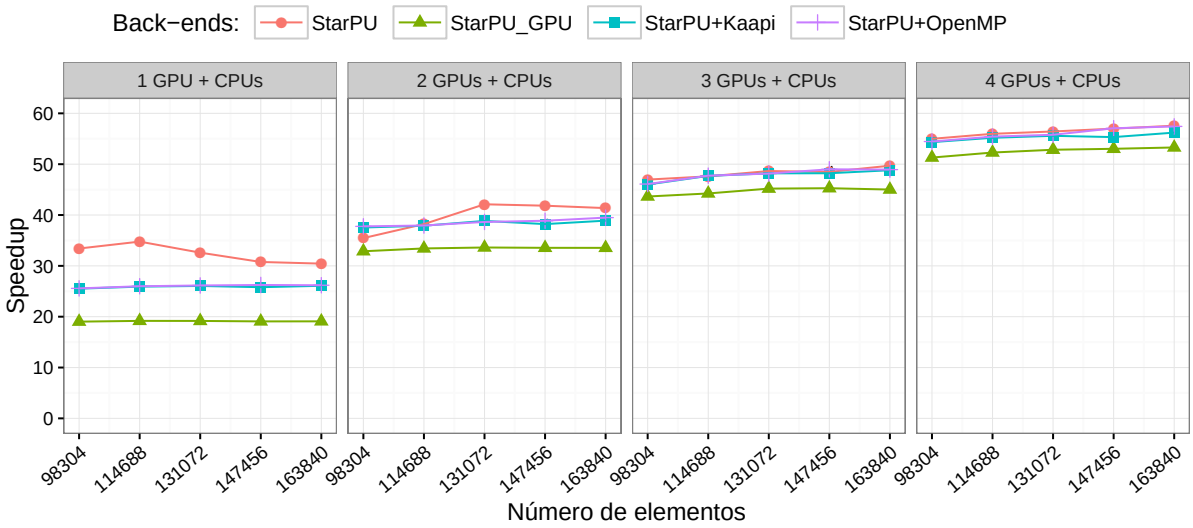
Nas execuções com somente GPUs (série StarPU_GPU), a aceleração permaneceu regular diante das mudanças no tamanho das entradas. Já a combinação de GPUs com CPUs não resultou em ganhos superiores ao de somente GPUs em nenhum caso, (com exceção do *back-end* StarPU com 1 GPU), mesmo com o *speedup* permanecendo praticamente uniforme para os diferentes números de elementos.

Melhor configuração

Na Figura 4.11 estão detalhados os melhores desempenhos alcançados pela aplicação CFD para cada tamanho de entrada. As execuções com somente CPUs não são mostradas, visto que os resultados são os mesmos do experimento de máxima configuração. Já para os testes com GPU+CPU, as configurações utilizadas foram:

- ***Back-end* StarPU**: 1GPU+26CPUs, 2GPUs+12CPUs, 3GPUs+10CPUs e 4GPUs+10CPUs;
- ***Back-end* StarPU+OpenMP**: 1GPU+10CPUs, 2GPUs+10CPUs, 3GPUs+10CPUs e 4GPUs+10CPUs;
- ***Back-end* StarPU+Kaapi**: 1GPU+10CPUs, 2GPUs+10CPUs, 3GPUs+10CPUs e 4GPUs+10CPUs.

Figura 4.11 – CFD: melhor configuração variando o tamanho da entrada. Bloco: 2.048. Iterações: 100.



Fonte: Próprio autor.

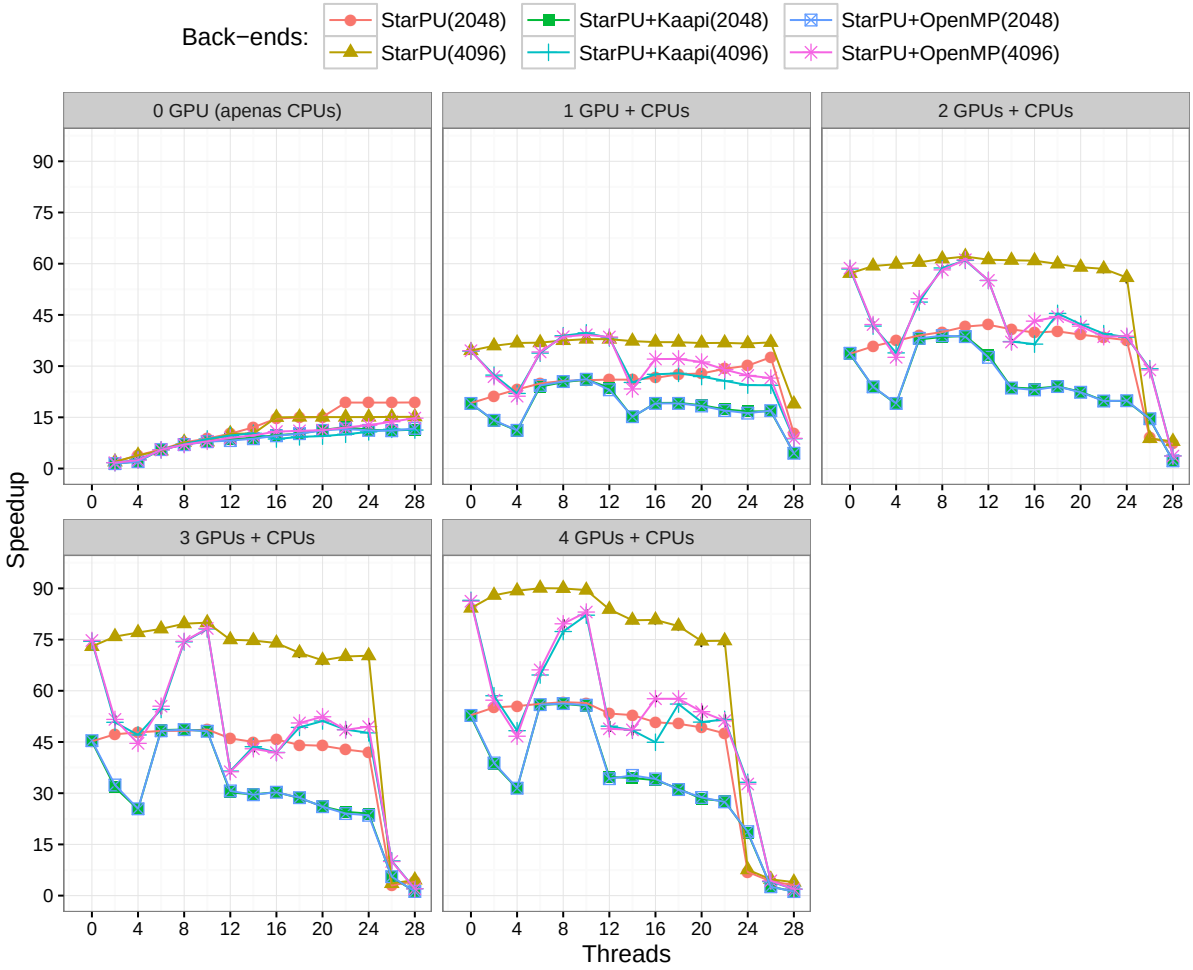
Os resultados da Figura 4.11 para somente GPUs (série StarPU_GPU) são os mesmos já apresentados no experimento de máxima configuração (Figura 4.10). Para as execuções combinando GPUs e CPUs, percebe-se que todos os *back-ends* obtiveram desempenhos superiores ao de somente GPUs independente do tamanho da entrada e do número de GPUs utilizadas.

Escalabilidade por bloco (StarPU)

Por fim, o último experimento realizado com a CFD foi o de escalabilidade variando o tamanho do bloco. Seus resultados estão disponíveis no gráfico da Figura 4.12. Nos testes

empregando somente CPUs, o uso de um bloco maior (4.096) não trouxe ganhos, sendo que o melhor desempenho com 28 *threads* foi alcançado pelo *back-end* StarPU com bloco de 2.048 (aceleração de 19,3).

Figura 4.12 – CFD: escalabilidade variando o tamanho do bloco com os *back-ends* da *runtime* StarPU. Tamanho: 131.072. Blocos: 2.048 e 4.096. Iterações: 100.



Fonte: Próprio autor.

Nos cenários utilizando somente GPUs (0 *thread*), a Figura 4.12 indica ganhos com o uso do bloco maior (4.096). O desempenho da versão StarPU também aponta ganhos na combinação de GPUs e CPUs com o bloco maior, porém as acelerações diminuem a partir de 12 e 10 *threads* com 2, 3 e 4 GPUs. Com as versões StarPU+OpenMP e StarPU+Kaapi, o desempenho com o bloco de 4.096 é melhor se comparado ao com o bloco de 2.048, sendo que ambas configurações apresentaram as mesmas oscilações para 2 e 4 *threads*, bem como as perdas a partir de 10 *threads*.

4.3 DISCUSSÃO DOS RESULTADOS

Esta seção discute os resultados apresentados para as três mini-aplicações científicas utilizadas. Ela está dividida em quatro subseções, sendo cada uma destinada a um dos quatro experimentos realizados com cada programa. As descrições destes experimentos estão disponíveis na introdução da seção 4.2 (página 54).

4.3.1 Escalabilidade

Os experimentos de escalabilidade sugerem que, de forma geral, houve ganhos a medida que mais recursos foram adicionados ao processamento, sejam eles CPUs, GPUs ou a combinação de ambos. Isto aconteceu de acordo com o previsto na hipótese deste trabalho, apesar da ocorrência de situações onde o desempenho não escalou para todas as *threads*.

Considerando somente CPUs, ocorreram estabilizações nos ganhos entre alguns intervalos de *threads* com os *back-ends* OpenMP e StarPU para as aplicação N-Body e CFD. Acredita-se que este fato aconteceu devido a divisão do trabalho em blocos que resulta em um número de tarefas homogêneas a serem processadas. Caso este número não seja divisível pelo número de *threads* disponíveis, algumas ficam ociosas durante o último ciclo de processamento, fazendo com que o tempo de execução permaneça estável. Por exemplo, a aplicação N-Body gera 48 tarefas. Com 16 *threads* é preciso 3 ciclos de processamento para executar todas elas. Com 18 e 20 este número se repete, porém nestes casos, o último ciclo fica com 4 e 12 *threads* ociosas. Este problema poderia ser solucionado utilizando um tamanho de bloco menor, porém nos experimentos optou-se por aplicar um bloco compatível entre CPU e GPU. Com a aplicação Hotspot, este evento não ocorreu pois ela é modelada na forma de matriz, sendo geradas mais tarefas que diminuem a incidência deste fato. O *back-end* Kaapi utiliza um particionamento adaptativo e por isso o problema também não ocorre. Já com os *back-ends* StarPU+OpenMP e StarPU+Kaapi, todas as *threads* disponíveis cooperam para processar uma mesma tarefa, fato que também justifica a não ocorrência da estabilização.

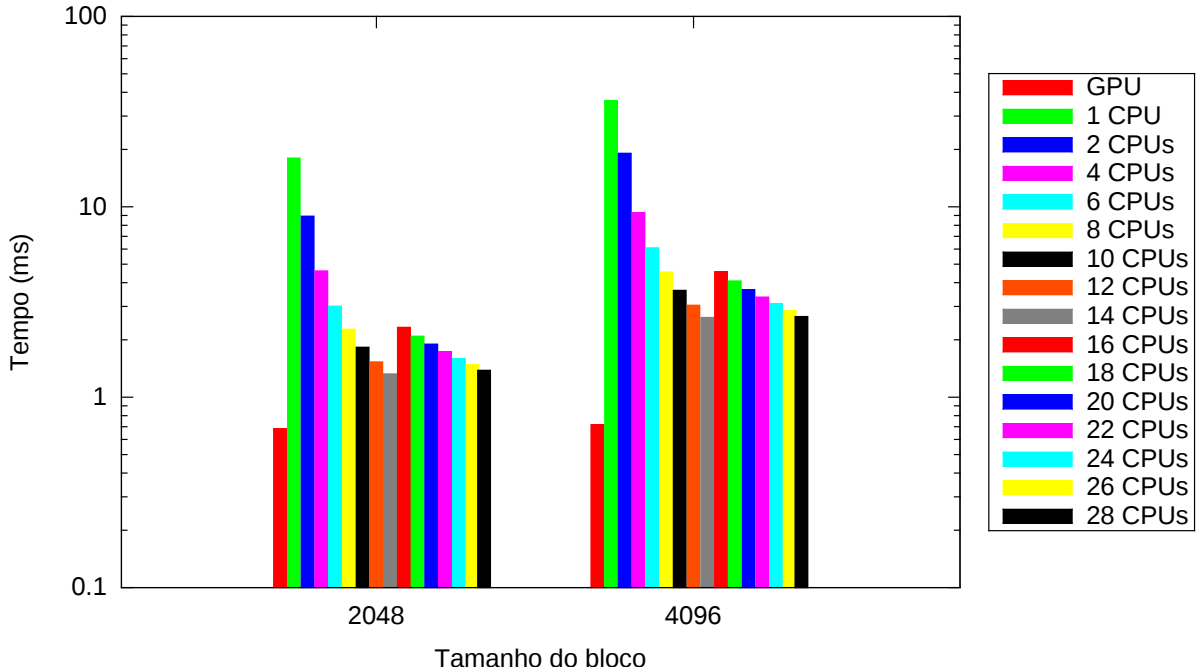
Nas execuções combinando GPUs e CPUs, as quedas que ocorreram próximas ao máximo de *threads* já estavam previsas, pois elas são resultado de uma característica da *runtime* StarPU, que dedica um núcleo do processador para controlar cada GPU do sistema (AUGONNET et al., 2011). Desta maneira, caso estes núcleos sejam utilizados também para processar, pode acontecer uma queda no desempenho, uma vez que o escalonamento das tarefas para as GPUs tende a ser prejudicado.

Também aconteceram quedas de desempenho que acredita-se terem sido ocasionadas em virtude da *runtime* StarPU não possuir uma afinidade de dados para a arquitetura NUMA. Estas quedas resultaram na interrupção dos ganhos de aceleração após o uso de 10 ou 12 *threads*. Exemplos desta situação são as execuções com os *back-ends* StarPU+OpenMP e StarPU+

Kaapi, bem como os experimentos da aplicação CFD com o *back-end* StarPU. O escalonador da *runtime* StarPU prioriza a alocação das *threads* nos processadores do primeiro *socket* (ou nó). Somente quando ele estiver totalmente ocupado (ou seja, com 14 *threads*) é que a *runtime* utiliza os núcleos do segundo nó. Entretanto, toda a memória requerida pelas aplicações é alocada no primeiro *socket*, pois este possui 128 GB de capacidade, valor suficiente para todas elas. De acordo com dados obtidos a partir da ferramenta NUMACLT, a distância para acesso à memória é de 10 quando realizada dentro do mesmo *socket*, passando a ser de 21 quando ocorre no outro nó. Isto indica que há um custo maior nas operações de acesso a memória a partir do uso da décima quinta *thread*. No entanto, é preciso considerar que a *runtime* StarPU dedica núcleos do processador para controlar as GPUs, fazendo com que este número diminua. Por exemplo, caso utilize-se 1 GPU, a décima quarta *thread* já terá de ser alocada no *socket* com acesso mais custoso a memória. Assim, como executou-se os experimentos somente com números pares, nas execuções que acredita-se terem sido afetadas por este problema os ganhos foram mantidos no máximo até 12 *threads*.

Outro indicativo da influência da arquitetura NUMA nas quedas de aceleração são os dados de calibragem das tarefas gerados pela *runtime* StarPU para a aplicação CFD utilizando o *back-end* StarPU+OpenMP. Estes dados estão apresentados no gráfico da Figura 4.13 e representam o tempo que uma tarefa levou para ser executada por cada configuração disponível na plataforma. Conforme é possível observar, as tarefas que são processadas de forma paralela pelas CPUs reduzem seu tempo até 14 *threads*, quando ocorre um aumento com 16 *threads*.

Figura 4.13 – CFD: calibragem das tarefas para a *runtime* StarPU através do *back-end* StarPU+OpenMP. Tamanho: 131.072. Função: `compute_flux()`.



Fonte: Próprio autor.

A situação evidenciada no gráfico da Figura 4.13 ocorreu com a aplicação CFD, sendo

que os experimentos demonstraram ser ela quem aparenta ter maior impacto da arquitetura NUMA, com quedas acontecendo a partir de 10 *threads*. Apesar do mesmo comportamento não aparecer nos dados de calibragem das aplicações N-Body e Hotspot, as quedas de desempenho similares as da aplicação CFD que aconteceram para as versões StarPU+OpenMP e StarPU+Kaapi sugerem a ocorrência do mesmo fato. Além disso, os *back-ends* StarPU+OpenMP e StarPU+Kaapi tendem a ser mais afetados que o StarPU, visto que neles as *threads* cooperam para executar uma mesma tarefa e concorrem pela memória do mesmo bloco. Em relação a hipótese deste trabalho, as perdas relatadas e que acredita-se serem resultantes da arquitetura NUMA prejudicaram o desempenho das execuções combinando GPUs com CPUs, pois as aplicações não apresentaram ganhos para todas as *threads*.

Outro fato relevante a ser considerado para o experimento de escalabilidade são as oscilações apresentadas pela aplicação CFD combinando GPUs com 2 e 4 *threads* nos *back-ends* StarPU+OpenMP e StarPU+Kaapi (Figura 4.12 na página 67). Em um comparativo de rastros de execução utilizando 2GPUs+4CPUs e 2GPUs+6CPUs, não foi possível identificar as causas deste comportamento. Contudo, a partir das estatísticas destes rastros, que estão expostas na Tabela 4.1, percebe-se que a razão do maior tempo para a execução com 4 CPUs é oriundo do período que as GPUs ficam inativas (estado *Sleeping*). Com 6 *threads*, as estatísticas apontam que este tempo é 28,71% menor do que com 4, mesmo a execução com 4 *threads* processando 7,42% mais tarefas nas GPUs (estado *Executing*). Como os níveis de transferências de memória mantém-se no mesmo patamar em ambos cenários com diferenças máximas de 6,68% (estados *DriverCopy* e *DriverCopyAsync*), sugere-se que o escalonador faz com que a aplicação fique aguardando o término do processamento das CPUs para submeter novas tarefas às GPUs.

Tabela 4.1 – CFD: estatísticas da *runtime* StarPU para 2GPUs+4CPUs e 2GPUs+6CPUs. Tamanho: 131.072. Bloco: 2.048. Iterações: 20[6].

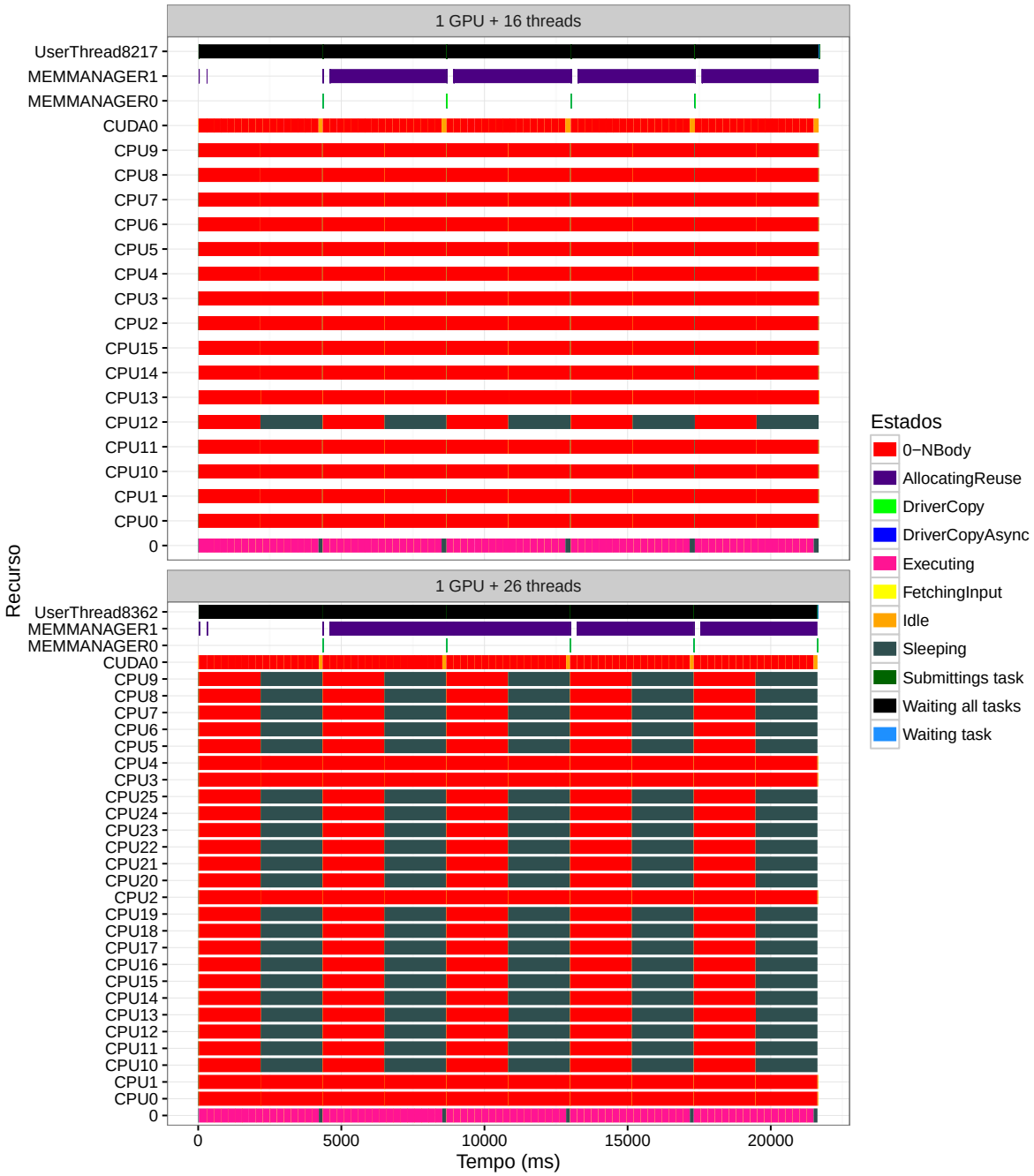
Estado	**Tempo (ms)**		**Contagem**		**Descrição**
	4 *threads*	6 *threads*	4 *threads*	6 *threads*	
DriverCopy	506,93	473,06	29.342	27.278	Cópias síncronas entre dispositivos.
DriverCopyAsync	138,56	129,69	14.671	13.639	Cópias assíncronas entre dispositivos.
Executing	1.476,77	1.426,65	3.805	3.542	GPUs executando tarefas.
Sleeping	3.302,51	2.354,24	1.029	1.201	GPUs inativas.

Fonte: Próprio autor.

A última situação a ser analisada é o saturamento que ocorre com 1 GPU a partir de 16 *threads* para o *back-end* StarPU na aplicação N-Body (gráfico da Figura 4.2). De acordo com os rastros gerados pela *runtime* e apresentados na Figura 4.14, acredita-se que isto aconteça devido a uma escolha do escalonador, que da preferência para executar as tarefas somente na GPU. O gráfico evidencia na execução com 26 *threads* que muitas CPUs ficam inativas (estado *Sleeping*).

[6]Utilizado 20 iterações, pois com 100 o arquivo do rastro ficou muito grande (mais de 100 MB). Isto inviabilizou sua análise.

Figura 4.14 – N-Body: rastros StarPU com 1GPU+16CPUs e 1GPU+26CPUs. Tamanho: 98.304. Bloco: 2.048. Iterações: 5.



Fonte: Próprio autor.

4.3.2 Máxima configuração

Nos experimentos de máxima configuração, os desempenhos somente com GPUs não apresentaram oscilações em nenhuma das aplicações com as variações das entradas. Já com apenas CPUs, os *back-ends* OpenMP e StarPU oscilaram para as aplicações N-Body e CFD. Estima-se que este comportamento seja resultado da estabilização de desempenho que ocorre devido a divisão da entrada em blocos, conforme já explicado na discussão sobre o experimento de escalabilidade (seção 4.3.1).

As execuções combinando GPUs e CPUs através do *back-end* StarPU também apresentaram variações. Acredita-se que o StarPU é mais suscetível a oscilações de desempenho para diferentes tamanhos de entrada do que o StarPU+OpenMP e o StarPU+Kaapi. Isto ocorre pois a versão StarPU executa as tarefas nas CPUs de forma sequencial. Desse modo, cada CPU disponível para o processamento é uma unidade independente, sendo que nem sempre o escalonador da *runtime* decide ou consegue utilizar todas elas. Somado a isso, a diferença nos tempos de processamento das tarefas por uma GPU e por uma CPU é grande, conforme dados apresentados na Tabela 4.2. Este fato faz com que o escalonador opte por preterir as CPUs, principalmente quando a entrada é pequena ou se têm mais de uma GPU disponível. Outro fator que pode ter colaborado para a incidência de variações foram as diferentes entradas que geraram números de tarefas diferentes, modificando a quantidade de paralelismo disponível. Menos ou mais paralelismo tendem a influenciar as decisões do escalonador, que nem sempre consegue deixar duas entradas no mesmo patamar de ganhos.

Tabela 4.2 – Tempos das tarefas das aplicações executadas com a *runtime* StarPU.

Aplicação	**Tempo GPU**	**Tempo CPU**	**Diferença**
N-Body	245,95 ms	2.079,84 ms	8,45 vezes
Hotspot	9,87 ms	506,33 ms	51,30 vezes
CFD	0,68 ms	18,03 ms	26,51 vezes

Fonte: Próprio autor.

Mesmo diante da ocorrência das variações abordadas no parágrafo anterior, a hipótese levantada neste estudo esperava que os desempenhos combinando GPUs e CPUs obtivessem valores superiores aos com somente GPUs. Como isto não ocorreu quando utilizou-se a configuração máxima para as aplicações Hotspot e CFD, foi necessário realizar o experimento de melhor configuração.

4.3.3 Melhor configuração

Utilizando a melhor configuração de cada aplicação para cada *back-end*, verificou-se que em praticamente todos os casos houve ganhos superiores (ou no mínimo iguais) nas combinações de GPUs e CPUs na comparação com uso de somente GPUs. Diante disso, percebe-se que a combinação de processamento em multi-CPU e multi-GPU pode ser utilizada para superar o desempenho de apenas aceleradores, conforme havia sido previsto na hipótese deste trabalho.

Estima-se que a necessidade de utilizar menos *threads* do que o máximo permitido pela plataforma para alcançar o melhor desempenho tenha acontecido em virtude da falta de afinidade de dados da *runtime* StarPU para a arquitetura NUMA, conforme evidências descritas na discussão do experimento de escalabilidade (tópico 4.3.1). Este fato ocorreu para a aplicação CFD, onde os melhores resultados para 2, 3 e 4 GPUs foram obtidos com o uso de 12 *threads* ou menos. Situação semelhante resultou do uso dos *back-ends* StarPU+OpenMP e StarPU+Kaapi

para a aplicação Hotspot, que atingiram seu melhor desempenho com 12 (para 1 e 2 GPUs) e 10 (para 3 e 4 GPUs) *threads*.

Outro fator que acredita-se ter influenciado na necessidade de redução no número de *threads* são as decisões do escalonador da *runtime* StarPU. Com a aplicação Hotspot e o *back-end* StarPU, alcançou-se o melhor desempenho com uma *thread* a menos do que a configuração máxima para todos os números de GPUs. Por exemplo, com 2 GPUs, ao invés do valor máximo ser obtido com 26 CPUs, ele foi registrado com 25.

Em decorrência disto, foi realizada uma investigação que originou os gráficos da Figura 4.15, onde são apresentados rastros de execução com 2GPUs+26CPUs e 2GPUs+25CPUs. Analisando os dois gráficos, o teste com 26 *threads* leva mais tempo para finalizar a execução do que com 25 em razão dos maiores períodos de ociosidade e inatividade (estados *Idle* e *Sleeping*) entre as iterações. De acordo com as estatísticas do processamento que foram geradas pela *runtime* StarPU e estão expostas na Tabela 4.3, o tempo total de execução das tarefas (estado *0-Hotspot*), das transferências de memória (estados *DriverCopyAsync* e *DriverCopy*) e das execuções de tarefas somente nas GPUs (estado *Executing*) possuem valores semelhantes. Neste sentido, sugere-se que, quando há no ambiente uma CPU a mais, decisões do escalonador fazem com que ocorram períodos maiores de ociosidade e inatividade nos intervalos entre iterações. Com 2 GPUs, isto resultou em um tempo médio de execução 8,80% superior para 26 *threads* (de 7,84 para 8,53 segundos).

Tabela 4.3 – Hotspot: estatísticas da *runtime* StarPU para 2GPUs+26CPUs e 2GPUs+25CPUs. Tamanho: 16.384x16.384. Bloco: 1.024x1.024. Iterações: 5.

Estado	**Tempo (ms)**		**Contagem**		**Descrição**
	25 *threads*	26 *threads*	25 *threads*	26 *threads*	
0-Hotspot	135.023,1	138.829,2	1.280	1.280	Execução da tarefa Hotspot.
DriverCopy	1.008,0	1.024,4	4.140	4.100	Cópias síncronas entre dispositivos.
DriverCopyAsync	2.668,3	2.609,8	2.070	2.050	Cópias assíncronas entre dispositivos.
Executing	10.211,7	10.074,9	1.030	1.020	GPUs executando tarefas.
Idle	5.654,8	8.700,2	1.308	1.307	CPUs ociosas.
Sleeping	76.232,9	121.900,4	162	168	GPUs inativas.

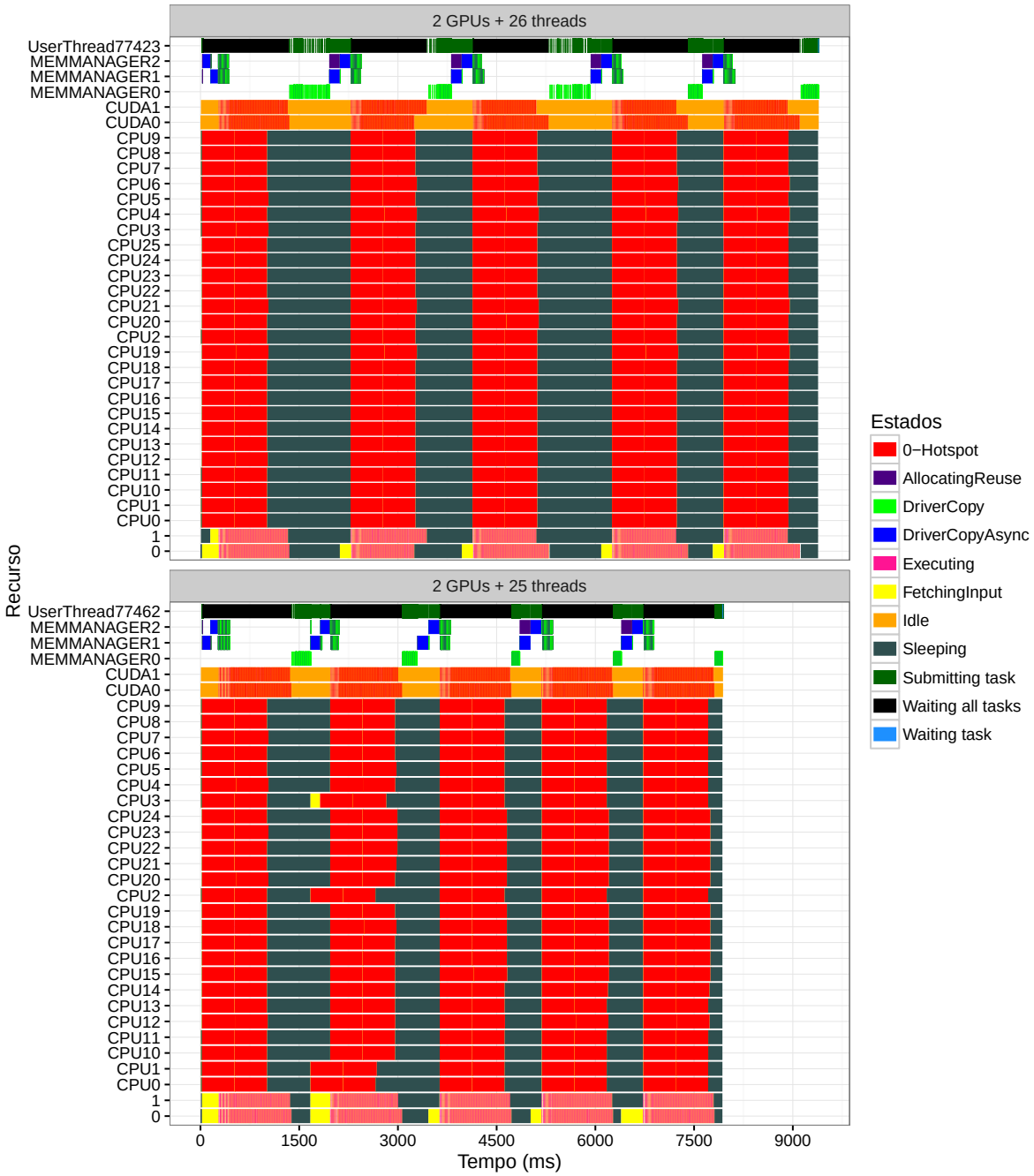
Fonte: Próprio autor.

4.3.4 Escalabilidade por bloco (StarPU)

O último experimento a ter seus resultados discutidos é o de escalabilidade com diferentes números de blocos, onde verificou-se o desempenho das execuções com a *runtime* StarPU a partir de tarefas com o dobro do tamanho das utilizadas nos testes de escalabilidade.

Esperava-se que com um grão maior o desempenho das GPUs melhorasse em prejuízo ao desempenho das CPUs. Com o *back-end* StarPU, o uso do bloco maior não superou o bloco menor em nenhuma aplicação nos cenários envolvendo apenas CPUs. Já as execuções envolvendo GPUs alcançaram acelerações superiores para as aplicações N-Body e CFD. Com

Figura 4.15 – Hotspot: rastros StarPU com 2GPUs+26CPUs e 2GPUs+25CPUs. Tamanho: 16.384x16.384. Bloco: 1.024x1.024. Iterações: 5.



Fonte: Próprio autor.

a Hotspot isto não ocorreu, fato que indica uma saturação das GPUs, ou seja, o bloco menor já estava explorando-as em sua capacidade máxima. Para a aplicação N-Body, sugere-se que o grão maior das tarefas fez com que o escalonador decidisse executá-las apenas nas GPUs, resultando assim em ganhos estáveis mesmo com o aumento do número de *threads*. Acredita-se que isso ocorra em virtude do aumento da diferença do tempo entre GPU e CPU na execução de uma única tarefa, pois os dados de calibragem da *runtime* StarPU indicam que ela passa de 8,45 para 16,07 vezes com blocos de 2.048 e 4.096, respectivamente.

Outro comportamento esperado neste experimento seria a melhora no desempenho dos *back-ends* StarPU+OpenMP e StarPU+Kaapi para as execuções com o bloco maior. A versão StarPU+Kaapi obteve resultados melhores do que os demais *back-ends* para as aplicações N-Body e Hotspot (exceto com 4 GPUs). Estima-se que isso tenha ocorrido em virtude das otimizações que o Kaapi possui para arquiteturas NUMA. Já o *back-end* StarPU+OpenMP conseguiu superar a variante StarPU utilizando o bloco maior apenas para a aplicação N-Body.

Os *back-ends* StarPU+OpenMP e StarPU+Kaapi não obtiveram ganhos em comparação ao StarPU utilizando o bloco maior para a aplicação CFD. Sugere-se que a razão deste fato provém deles não terem conseguido aproveitar a maior granularidade para tirar proveito das tarefas paralelas nas CPUs. Supõe-se que isto tenha relação com a carga de processamento das tarefas da aplicação, que é menor se comparada às demais. A título de comparação, com bloco de 2.048 uma tarefa da aplicação N-Body leva 361,7 (GPU) e 115,35 (CPU) vezes mais tempo para ser executada do que a tarefa `compute_flux()`, a maior que a CFD possui. Portanto, a falta de carga de trabalho é a provável causa do baixo rendimento das versões StarPU+OpenMP e StarPU+Kaapi com o bloco dobrado.

4.4 CONCLUSÃO

O presente capítulo apresentou os resultados experimentais de três mini-aplicações científicas que foram implementadas através da HPSM. O primeiro experimento realizado visou verificar o sobrecusto adicionado pela API no desempenho de um programa, sendo constatado que a HPSM pode agregar custo e que ele varia de acordo com a aplicação, podendo chegar a ser de até 16,4%.

Os experimentos também visaram comprovar a hipótese estabelecida para este estudo, onde estimou-se que aplicações científicas podem valer-se do processamento heterogêneo em multi-CPU e multi-GPU para alcançar um desempenho superior em relação ao uso de apenas aceleradores. Os resultados destes experimentos confirmaram a hipótese, pois as aplicações N-Body, Hotspot e CFD, além de alcançarem ganhos ao utilizar somente CPUs e somente GPUs, também superaram o desempenho obtido por somente aceleradores (GPUs) através da combinação de multi-CPU e multi-GPU. No entanto, ocorreram situações onde a configuração máxima da plataforma não obteve o melhor desempenho, sendo este alcançado utilizando menos recursos. Desta forma, os ganhos obtidos, apesar de confirmarem a hipótese, poderiam ter sido melhores, visto que as aplicações não apresentaram escalabilidade para todas as *threads*.

Acredita-se que contribuiu para as aplicações não terem escalado para todas as *threads* o fato da *runtime* StarPU não possuir uma afinidade de dados para a arquitetura NUMA. Isto ocorre pois ela não aloca estes dados na memória do mesmo nó onde está sendo executada a *thread* que irá acessá-los. Assim, como as aplicações empregadas nos experimentos utilizam somente a memória do primeiro nó, o custo de acesso a ela pelas *threads* do segundo nó é maior,

resultando em quedas na aceleração. Estima-se que caso a ferramenta possuísse o recurso de afinidade de dados com as *threads*, as perdas originadas pela arquitetura seriam menores, pois segundo Virouleau et al. (2016) e He, Chen e Tang (2016), a localidade dos dados é um dos fatores que colabora para se alcançar desempenho em máquinas NUMA. Somado a isso, outro fator que pode ter influenciado nos resultados são as decisões do escalonador da *runtime* StarPU, como por exemplo, a não utilização de todas as CPUs disponíveis para priorizar as GPUs. Estas decisões do escalonador também podem ter ocasionado períodos de ociosidade e inatividade nos intervalos entre iterações quando fez-se uso da configuração máxima da plataforma, onde sugere-se que houve influência da falta de afinidade de dados para arquitetura NUMA.

A HPSM facilita a codificação de programas para o processamento CPU+GPU através de laços paralelos. Para tal, é necessário dividir os dados em blocos, sendo este outro fator que acredita-se ter impactado nos resultados. As três aplicações utilizadas requerem mudanças na forma de particionamento dos dados mapeados como *Views* após cada iteração. Isto cria a necessidade de sincronizar as memórias acessadas pelas CPUs e GPUs, visto que a alteração do particionamento pode ocasionar transferências de dados, o que adiciona custo ao processamento. Neste sentido, cabe avaliar se a HPSM compensa o sobrecusto trazido ao programa, bem como se ela é adequada ao processamento paralelo que deseja-se alcançar.

5 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou a HPSM, uma API de alto nível em linguagem C++ voltada a facilitar e tornar mais eficiente a implementação de programas paralelos para serem executados em multi-CPU e multi-GPU. Os programas desenvolvidos a partir da API são executados através de ferramentas de *back-end*, sendo suportadas Serial, OpenMP e StarPU. No decorrer deste estudo, detalhou-se as funcionalidades e a estrutura da HPSM, contemplando suas classes e rotinas. Além disso, avaliou-se o esforço de programação necessário para implementar um programa paralelo através da API em comparação às interfaces StarPU e OpenMP, o que indicou que seu uso pode reduzir em mais de 50% o tamanho de um programa voltado a execução em multi-CPU e multi-GPU através da *runtime* StarPU. No entanto, houve aumento do esforço ao empregar a HPSM para a construção de uma aplicação *multicore*, visto que em comparação a um código OpenMP o fonte com a API ficou 2,5 vezes maior.

Outros trabalhos já propuseram interfaces de programação paralela de alto nível visando facilitar o desenvolvimento de aplicações para explorar aceleradores, como GPUs. O objetivo deste estudo, apesar de caminhar no mesmo sentido, difere-se dos demais pois foca em uma interface que permite o desenvolvimento de programas direcionados ao processamento heterogêneo e simultâneo por meio de laços e reduções paralelas em multi-CPU e multi-GPU. Os demais trabalhos que suportam multi-CPU e multi-GPU requerem o uso de um paralelismo explícito através de tarefas assíncronas, enquanto que este permite o paralelismo implícito por meio de laços paralelos. Já os outros trabalhos baseados em laços paralelos não suportam o processamento CPU+GPU.

A HPSM foi utilizada para implementar as mini-aplicações científicas N-Body, Hotspot e CFD. Elas serviram de base para experimentos em uma plataforma NUMA composta por 2 CPUs de 14 núcleos (totalizando 28 *cores*) e 4 GPUs com o intuito de validar a API. Após a execução de quatro tipos de experimentos para cada aplicação, constatou-se que todas obtiveram ganhos em cenários com somente CPUs, somente GPUs e CPUs+GPUs. Mesmo assim, é preciso destacar que o uso da nova API pode trazer impacto no desempenho do programa, sendo ele variável de acordo com a aplicação. Os experimentos demonstraram que o sobrecusto pode chegar até 14,0%.

De acordo com os objetivos definidos para o trabalho, pode-se afirmar que eles foram alcançados, pois implementou-se um modelo de programação que foi aplicado e validado através de experimentos, bem como teve o esforço necessário para utilizá-lo comparado com outras abordagens já existentes. Os experimentos executados comprovaram a hipótese do estudo de utilizar a combinação de multi-CPU e multi-GPU para superar o desempenho obtido com apenas aceleradores (GPUs neste caso). Apesar disto, ocorreram casos onde a configuração máxima da plataforma não demonstrou o melhor desempenho, sendo este alcançado utilizando menos recursos. Estima-se que contribuíram para isto fatores como o *back-end* StarPU não possuir

uma afinidade de dados para a arquitetura NUMA, além de decisões do escalonador da *runtime* StarPU, que também podem ser resultantes desta falta de afinidade. Portanto, o desempenho alcançado poderia ter sido melhor, uma vez que as aplicações não apresentaram escalabilidade para todas as *threads*.

Como trabalhos futuros, elencamos:

- Implementar novas aplicações com a HPSM que demandem o uso de reduções, a fim de verificar o quanto elas são eficientes em execuções com multi-CPU e multi-GPU;
- Continuar desenvolvendo a HPSM, adicionando a ela novos recursos. Um destes recursos seria o de tarefas assíncronas, o que ampliaria o leque de aplicações onde a API poderia ser utilizada. Além disso, poderiam ser adicionados mais *back-ends* visando a execução em aceleradores, como por exemplo, utilizando as ferramentas XKaapi e CUDA;
- Comparar o desempenho de aplicações implementadas a partir da HPSM com as codificadas diretamente com as ferramentas de *back-end*. Deste modo, seria possível calcular seu sobrecusto de uma maneira mais precisa, visto que seriam comparados dois programas paralelos.

REFERÊNCIAS BIBLIOGRÁFICAS

ADCOCK, A. B. et al. Evaluating OpenMP Tasking at Scale for the Computation of Graph Hyperbolicity. In: RENDELL, A. P.; CHAPMAN, B. M.; MüLLER, M. S. (Ed.). **Proceedings of the OpenMP in the Era of Low Power Devices and Accelerators**. Canberra, ACT, Australia: Springer Berlin Heidelberg, 2013. (9th International Workshop on OpenMP, IWOMP 2013), p. 71–83. ISBN 978-3-642-40697-3.

ANDERSCH, M.; CHI, C. C.; JUURLINK, B. Using openmp superscalar for parallelization of embedded and consumer applications. In: **Proceedings of the International Conference on Embedded Computer Systems**. Samos, Greece: [s.n.], 2012. p. 23–32.

AUGONNET, C.; THIBAULT, S.; NAMYST, R. **StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines**. [S.l.], 2010. 33 p. Disponível em: <https://hal.inria.fr/inria-00467677>.

AUGONNET, C. et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. **Concurrency and Computation: Practice and Experience**, John Wiley and Sons, v. 23, n. 2, p. 187–198, 2011.

AYGUADÉ, E. et al. A Proposal for Task Parallelism in OpenMP. In: CHAPMAN, B. et al. (Ed.). **Proc. of the A Practical Programming Model for the Multi-Core Era (3rd International Workshop on OpenMP, IWOMP 2007)**. Beijing, China: Springer Berlin Heidelberg, 2008. p. 1–12. ISBN 978-3-540-69303-1.

BROQUEDIS, F.; GAUTIER, T.; DANJEAN, V. libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms. In: **Proc. of the OpenMP in a Heterogeneous World - 8th IWOMP**. Rome, Italy: [s.n.], 2012. p. 102–115.

BSC. **OmpSs Specification**. 2015. Acesso em: 13 abr 2016. Disponível em: <http://pm.bsc.es/ompss-docs/specs/OmpSsSpecification.pdf>.

BUENO, J. et al. Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces. In: **Proceedings of the 27th International Conference on Supercomputing**. Eugene, Oregon, USA: ACM, 2013. (ICS '13), p. 359–368. ISBN 978-1-4503-2130-3.

CHAPMAN, B.; JOST, G.; PAS, R. v. d. **Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)**. Cambridge, MA, USA: The MIT Press, 2007. ISBN 0262533022, 9780262533027.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: **Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on**. [S.l.: s.n.], 2009. p. 44–54.

____. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In: **Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)**. Washington, DC, USA: IEEE Computer Society, 2010. (IISWC '10), p. 1–11. ISBN 978-1-4244-9297-8.

CORRIGAN, A. et al. Running Unstructured Grid CFD Solvers on Modern Graphics Hardware. In: **19th AIAA Computational Fluid Dynamics Conference**. [S.l.: s.n.], 2009.

DONGARRA, J. et al. (Ed.). **Sourcebook of Parallel Computing**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 1-55860-871-0.

DURAN, A. et al. Ompss: a proposal for programming heterogeneous multi-core architectures. **Parallel Processing Letters**, v. 21, n. 2, p. 173–193, 2011.

____. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In: **International Conference on Parallel Processing, 2009. ICPP '09**. [S.l.: s.n.], 2009. p. 124–131. ISSN 0190-3918.

EDWARDS, H. C. et al. Manycore performance-portability: Kokkos multidimensional array library. **Scientific Programming**, v. 20, n. 2, p. 89–114, 2012.

FENG, W. chun; MANOCHA, D. High-performance computing using accelerators. **Parallel Computing**, v. 33, n. 10–11, p. 645 – 647, 2007. High-Performance Computing Using Accelerators.

FENTON, N. E.; BIEMAN, J. **Software Metrics: A Rigorous and Practical Approach**. 3. ed. [S.l.]: CRC Press, 2014. (Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series).

GARLAND, M.; KUDLUR, M.; ZHENG, Y. Designing a Unified Programming Model for Heterogeneous Machines. In: **SC '12: Proc. Conference on High Performance Computing Networking, Storage and Analysis**. [S.l.: s.n.], 2012.

GAUTIER, T. et al. X-kaapi: A multi paradigm runtime for multicore architectures. In: **Proceedings of the 42nd International Conference on Parallel Processing**. Washington, DC, USA: IEEE Computer Society, 2013. (ICPP '13), p. 728–735. ISBN 978-0-7695-5117-3.

____. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In: **Proc. of the IEEE 27th IPDPS**. [S.l.: s.n.], 2013. p. 1299–1308.

GREGORY, K.; MILLER, A. **C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®**. [S.l.]: Microsoft Press, 2012. (Developer Reference). ISBN 9780735668195.

HE, J.; CHEN, W.; TANG, Z. NestedMP: Enabling cache-aware thread mapping for nested parallel shared memory applications. **Parallel Computing**, v. 51, p. 56 – 66, 2016. ISSN 0167-8191.

HELLER, T.; KAISER, H.; IGLBERGER, K. Application of the ParalleX Execution Model to Stencil-based Problems. **Comput. Sci.**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 28, n. 2-3, p. 253–261, maio 2013. ISSN 1865-2034.

HUGO, A.-E. et al. Composing multiple starpu applications over heterogeneous machines: A supervised approach. In: **Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)**. [S.l.: s.n.], 2013. p. 1050–1059.

INRIA. **StarPU Handbook**. 2016. Acesso em: 05 oct 2016. Disponível em: <http://starpu.gforge.inria.fr/doc/starpu.pdf>.

JEFFERS, J.; REINDERS, J. **Intel Xeon Phi Coprocessor High-performance Programming**. Waltham, MA, USA: Elsevier, 2013. (Morgan Kaufmann). ISBN 9780124104143.

JONES, C. **Software Engineering Best Practices**. 1. ed. New York, NY, USA: McGraw-Hill, Inc., 2010. ISBN 007162161X, 9780071621618.

JUNIOR, P. J. **Introdução ao C++**. 1st. ed. São Paulo: Futura, 2003. ISBN 8574131725.

KAISER, H. **C++ and the Heterogeneous Challenge**. 2016. Online. Acesso em: 09 set 2016. Disponível em: <http://stellar.cct.lsu.edu/2016/02/c-and-the-heterogeneous-challenge/>.

KIRK, D. B.; HWU, W.-m. W. **Programming Massively Parallel Processors: A Hands-on Approach**. 2nd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

LIAO, C. et al. OpenUH: An optimizing, portable OpenMP compiler. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 19, n. 18, p. 2317–2332, 2007.

LIMA, J. V. F. **A Runtime System for Data-Flow Task Programming on Multicore Architectures with Accelerators**. maio 2014. Tese (Theses) — Grenoble University; UFRGS, maio 2014. Disponível em: <https://tel.archives-ouvertes.fr/tel-01151787>.

LIMA, J. V. F. et al. Exploiting concurrent gpu operations for efficient work stealing on multi-gpus. In: **Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing**. Washington, DC, USA: IEEE Computer Society, 2012. (SBAC-PAD '12), p. 75–82. ISBN 978-0-7695-4907-1.

MCCOOL, M.; REINDERS, J.; ROBISON, A. **Structured Parallel Programming: Patterns for Efficient Computation**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN 9780123914439, 9780124159938.

MENTEC, F. L.; DANJEAN, V.; GAUTIER, T. **X-Kaapi C programming interface**. [S.l.], 2011. 18 p. Disponível em: <https://hal.inria.fr/hal-00647474>.

MEYERS, S. **Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2014. ISBN 1491903996, 9781491903995.

MISRA, G. et al. Evaluation of rodinia codes on intel xeon phi. In: **Proceedings of the 4th International Conference on Intelligent Systems Modelling Simulation (ISMS)**. [S.l.: s.n.], 2013. p. 415–419. ISSN 2166-0662.

NIELSEN, F. **Introduction to HPC with MPI for Data Science**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2016. ISBN 3319219022, 9783319219028.

OpenMP. **OpenMP Application Program Interface Version 4.5**. 2016. Acesso em: 19 jul 2016. Disponível em: <http://www.openmp.org/mp-documents/openmp-4.5.pdf>.

OZEN, G.; AYGUADÉ, E.; LABARTA, J. On the roles of the programmer, the compiler and the runtime system when programming accelerators in openmp. In: **Proceedings of the Using and Improving OpenMP for Devices, Tasks, and More**. Salvador, Brazil: Springer Berlin Heidelberg, 2014. (10th International Workshop on OpenMP, IWOMP 2014), p. 215–229.

PACHECO, P. **An Introduction to Parallel Programming**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 9780123742605.

RÜNGER, G.; RAUBER, T. **Parallel Programming: for Multicore and Cluster Systems**. 2nd. ed. [S.l.]: Springer, 2013. ISBN 978-3-642-37800-3.

SCHILDT, H. **C++: The Complete Reference**. 3rd. ed. Berkeley, CA, USA: Osborne/McGraw-Hill, 1998. ISBN 0078824761.

STOTZER, E. **OpenMP Accelerator Model: OpenMP for Heterogeneous Computing**. 2014. Tutorial apresentado no IWOMP 2014 (International Workshop on OpenMP). Acesso em: 16 nov 2015. Disponível em: <http://portais.fieb.org.br/senai/iwomp2014/presentations/tutorial_accelerator_model.pdf>.

STROUSTRUP, B. **The C++ Programming Language**. 4th. ed. [S.l.]: Addison-Wesley Professional, 2013. ISBN 0321563840, 9780321563842.

THRUST. 2016. Acesso em: 21 mai 2016. Disponível em: <http://thrust.github.io/>.

VIROULEAU, P. et al. Using data dependencies to improve task-based scheduling strategies on NUMA architectures. In: **Euro-Par 2016**. Grenoble, France: [s.n.], 2016. (Euro-Par 2016).

WYNTERS, E. Fast and Easy Parallel Processing on GPUs Using C++ AMP. **J. Comput. Sci. Coll.**, Consortium for Computing Sciences in Colleges, USA, v. 31, n. 6, p. 27–33, jun. 2016. ISSN 1937-4771.

APÊNDICE A – LINGUAGEM C++

O objetivo do presente trabalho foi implementado por meio da linguagem de programação C++. De acordo com Bjarne Stroustrup, que desenvolveu as especificações iniciais e lançou a primeira versão da linguagem em 1979, ela foi criada para que programadores pudessem escrever bons programas de computador de modo simples e de forma agradável. C++ foi especificada baseada na linguagem C, sendo que esta última foi mantida como um subconjunto da nova. Além disso, novos recursos foram adicionados à C++ baseados em outras linguagens já existentes. O conceito de classes e orientação a objetos foi tomado da Simula67, enquanto que a sobrecarga de operadores foi trazido da Algol68 (STROUSTRUP, 2013).

Após a primeira implementação da linguagem feita por Stroustrup, que foi inicialmente batizada como "C com classes" (o termo C++ surgiu apenas em 1983), o C++ foi ganhando algumas novas funcionalidades. Elas foram incluídas exatamente para atender o propósito que levou a sua criação, que é permitir que os programadores possam compreender e gerenciar grandes aplicações, fato que é extremamente difícil utilizando a linguagem C. Destaca-se entre estas novas funcionalidades adicionadas ao C++ a STL (*Standard Template Library*), desenvolvida por Alexander Stepanov. A STL incorporou um grande conjunto de rotinas genéricas que facilitaram a manipulação de dados e, por ser bastante grande, expandiu consideravelmente o escopo original do C++. Grande parte da STL foi incluída na padronização da linguagem, sendo sua primeira versão finalizada em novembro de 1997 em conjunto pela ANSI e ISO (SCHILDT, 1998).

A linguagem C++ possui algumas característica principais. Além de ser orientada a objetos, ela tem uma estrutura simples para a construção de programas elegantes e organizados, permitindo também a divisão do mesmo em múltiplos módulos. O desempenho é outro fator de destaque, pois programas C++ podem atingir uma performance comparável com as linguagens de máquina. Isso é alcançado também com portabilidade, visto que uma mesma aplicação C++ é compatível com outras plataformas, requerendo para isso apenas uma nova compilação do código. Por fim, C++ é ainda bastante abstrata e versátil, pois permite que seus programas sejam implementados utilizando diversos recursos de programação para serem aplicados em diversas finalidades, como *softwares* embarcados, sistemas operacionais, bancos de dados, interfaces gráficas e jogos (JUNIOR, 2003).

Nos seções a seguir, são detalhados dois recursos que foram aplicados na implementação da HPSM e que contribuem para tornar a linguagem C++ robusta, flexível e poderosa.

A.1 – *TEMPLATES*

Os *templates* permitem definir funções e classes genéricas, onde o tipo de dados sobre o qual elas vão operar é especificado como um parâmetro. Desta maneira, é possível utilizar a mesma função ou classe com diversos tipos de dados sem que seja necessário replicar códigos em versões específicas para cada tipo (SCHILDT, 1998).

Além de garantirem uma maior flexibilidade em relação a tipagem de classes e funções, os *templates* também são a base para o uso de metaprogramação, que é definida por Stroustrup (2013) como uma programação voltada a manipular entidades (classes e funções). Ainda segundo Stroustrup (2013), a utilização da metaprogramação garante, além de melhorias na segurança do sistema de tipos, ganhos de desempenho durante a execução de uma rotina, visto que suas instruções são avaliadas em tempo de compilação. Considerando que o contexto deste trabalho é a área de alto desempenho, esta vantagem justifica a sua aplicação.

A.2 – *FUNCTORS*

Functors, também chamados Objetos de Função, são definidos como classes que sobrescrevem o operador de aplicação (`operator()`). Através destes objetos pode-se definir funções que podem ser passadas como parâmetro para outras rotinas (STROUSTRUP, 2013; SCHILDT, 1998). Deste modo, não é necessário utilizar um ponteiro para função como nos programas C.

A partir do padrão C++11, com a introdução das expressões *lambda*, os *Functors* passaram a possuir um concorrente com funcionalidades semelhantes. Apesar de permitir um novo modo de definir funções, tudo o que é implementado com *lambdas* pode também ser codificado usando os *functors*. No entanto, a vantagem dos *lambdas* é que eles podem ser declarados com muito menos código, tornando seu uso mais conveniente (MEYERS, 2014).

Para a implementação da HPSM, utilizou-se *Functors* por dois motivos. Primeiro, a HPSM requereu o uso de atributos e heranças para o objeto de função, e como os *lambdas* são apenas expressões, eles não permitem o uso destes recursos. Mesmo assim, não seria possível utilizar *lambdas*, visto que a versão 7.5 da biblioteca CUDA não suporta *lambdas* que executem em CPU e GPU.

APÊNDICE B – FONTES DAS APLICAÇÕES IMPLEMENTADAS COM A HPSM

Neste apêndice estão detalhados os principais trechos dos códigos fontes das mini-aplicações científicas implementadas utilizando a HPSM. Estas aplicações foram empregadas nos experimentos que originaram os resultados deste trabalho (capítulo 4). O fonte completo das aplicações pode ser obtido no repositório Git[1].

B.1 – N-BODY

A simulação de N-Body calcula a evolução de um sistema de partículas (ou corpos) que interagem entre si. Estas interações ocorrem em razão da influência que um corpo exerce sobre os outros, considerando para isso variáveis como distância e aceleração. Ela trabalha com dois vetores que contém os dados e as coordenadas X, Y e Z de cada partícula do ambiente. Conforme apresentado na Figura B.1, um deles é utilizado como entrada e outro como saída. No seu *kernel*, que no fonte está mapeado para o *Functor* `funcNBody`, a saída é calculada a partir da entrada. Neste sentido, após o processamento do laço paralelo deve ocorrer uma inversão dos *arrays*, pois a saída da iteração atual passará a ser a entrada da próxima. No fonte, esta inversão é realizada entre as linhas 16 e 18.

Figura B.1 – Fonte da aplicação N-Body paralelizado com a HPSM.

```
1 hpsm::range<1> rg(number_of_particles);
2
3 /* Para cada iteração... */
4 for(int timestep = 1; timestep <= number_of_timesteps; timestep++) {
5   hpsm::View<Particle> part_in(particle_array, number_of_particles,
6           number_of_particles, hpsm::AccessMode::In);
7   hpsm::View<Particle> part_out(particle_array2, number_of_particles,
8           block_size, hpsm::AccessMode::Out);
9
10  funcNBody func(part_in, part_out, number_of_particles, time_interval);
11  hpsm::parallel_for(rg, part_out.block_range(), func);
12
13  func.remove_data();
14
15  /* Inverte os arrays de partículas */
16  Particle * tmp  = particle_array;
17  particle_array  = particle_array2;
18  particle_array2 = tmp;
19 }
```

Fonte: Próprio autor.

[1] Git HPSM: <https://github.com/danidomenico/hpsm>

Outro fator importante na implementação do N-Body é o particionamento das *Views* da API (linhas 5 a 8 da Figura B.1). O vetor de entrada não é particionado visto que o cálculo de deslocamento das partículas requer acesso a todo o ambiente. Isso ocorre pois toda partícula pode exercer influência sobre as demais, fato que precisa ser considerado para avaliar a sua próxima posição. Levando em conta que há uma inversão dos vetores, a cada iteração o particionamento dos dados também é alterado.

B.2 – HOTSPOT

A aplicação Hotspot é uma ferramenta de simulação térmica empregada para estimar a temperatura de um processador baseando-se na sua arquitetura e em medições de energia (também simuladas) (CHE et al., 2009). A entrada da simulação é uma matriz que representa as dimensões do processador. O cálculo térmico é realizado a partir de duas matrizes que contém as temperaturas e uma matriz que possui as simulações de energia. No código fonte exposto na Figura B.2, as três matrizes são mapeadas como *Views* para a HPSM entre as linhas 5 e 13.

Figura B.2 – Fonte da aplicação Hotspot paralelizado com a HPSM.

```
1 hpsm::range<2> rg(row, col);
2
3 /* Para cada iteração... */
4 for(int i = 0; i < num_iterations ; i++) {
5   hpsm::View<FLOAT> v_power(power, row, col, bl_size,
6           hpsm::PartitionMode::Matrix_Vert_Horiz,
7           hpsm::AccessMode::In);
8   hpsm::View<FLOAT> v_result(result, row, col, bl_size,
9           hpsm::PartitionMode::Matrix_Vert_Horiz,
10          hpsm::AccessMode::Out);
11  hpsm::View<FLOAT> v_temp(temp, row, col, row,
12          hpsm::PartitionMode::Matrix_Vert_Horiz,
13          hpsm::AccessMode::In);
14
15  funcHotspot func(v_result, v_temp, v_power, Cap_1, Rx_1, Ry_1, Rz_1);
16  hpsm::parallel_for(rg, v_result.block_range(), func);
17
18  func.remove_data();
19
20  /* Inverte as matrizes de temperatura */
21  FLOAT* tmp = temp;
22  temp       = result;
23  result     = tmp;
24 }
```

Fonte: Próprio autor.

Seguindo a análise do fonte, o resultado aferido a partir das matrizes `temp` e `power` pela rotina mapeada no `funcHotspot` é armazenado na matriz `result`. Após o término do

processamento do laço é preciso inverter as matrizes de temperatura (linhas 21 a 23 da Figura B.2), pois as temperaturas registradas na iteração atual passarão a ser a entrada da próxima. Por fim, é importante ressaltar que a matriz `temp` quando mapeada para uma *View* não é particionada em blocos, visto que o cálculo térmico considera as bordas da posição que está tendo sua temperatura mensurada. Caso o particionamento em blocos fosse realizado, não seria possível garantir que a borda necessária estaria no bloco que está sendo processado. Assim, a inversão das matrizes resulta também em mudança na forma de particionamento dos dados.

B.3 – CFD

A aplicação CFD é um algoritmo para solucionar um problema de dinâmica de fluidos onde aplicam-se sobre um fluido compressível as equações de Euler considerando um ambiente de três dimensões (CHE et al., 2010). A versão implementada com a HPSM é baseada em um vetor de variáveis que representam os valores de conservação de massa, momento e energia dos elementos. O fonte da Figura B.3 demonstra em sua primeira parte (linhas 2 a 16) a rotina onde são executadas as iterações. Nela, são chamadas quatro funções, sendo que todas elas serão processadas através da API. A responsável por aplicar as equações de Euler é a `compute_flux`. Sua implementação está disponível entre as linhas 19 e 39.

A função `compute_flux` faz o mapeamento de cinco vetores para a HPSM em forma de *Views*. É importante ressaltar que o vetor com as variáveis (`variables`) não é particionado em blocos, pois a mensuração do fluxo requer acesso às variáveis dos elementos vizinhos (armazenados em `elements_surrounding_elements`). Desse modo, não é possível haver particionamento, visto que os elementos vizinhos podem não estar no mesmo bloco. Nas demais rotinas, todas a Views mapeadas para a API são divididas em blocos, pois não há a necessidade de acessar dados que podem estar em outras partições. Neste sentido, o vetor `variables` tem seu particionamento modificado de acordo com a rotina que será executada.

O último detalhe que precisa ser abordado em relação a aplicação CFD é a função `copy` (linha 3 da Figura B.3). Ela é chamada no início de cada iteração com o objetivo de replicar o vetor `variables`. Isso é necessário pois os valores da iteração anterior são necessário para o cálculo dos valores da iteração a ser executada. Dessa forma, eles precisam estar armazenados para serem utilizados pela chamada da rotina `time_step` na linha 13.

Figura B.3 – Fonte da aplicação CFD paralelizado com a HPSM.

```
/* Para cada iteração... */
for(int i = 0; i < iterations; i++) {
  copy(old_variables, variables, nelr * NVAR, block_size * NVAR);

  compute_step_factor(nelr, block_size, variables, areas, step_factors);

  for(int j = 0; j < RK; j++) {
    compute_flux(nelr, block_size, elements_surrounding_elements,
        normals, variables, fluxes, step_factors, ff_variable,
        ff_flux_contrib_momentum_x, ff_flux_contrib_momentum_y,
        ff_flux_contrib_momentum_z, ff_flux_contrib_density_energy);

    time_step(j, nelr, block_size, old_variables, variables,
        step_factors, fluxes);
  }
}

/* Fonte da função compute_flux */
void compute_flux(/* Paramêtros... */) {
  hpsm::View<int> v_ele_sur_ele(elements_surrounding_elements,
        nelr*NNB, block_size*NNB, hpsm::AccessMode::In);
  hpsm::View<float> v_normals(normals, nelr*NNB*NDIM,
        block_size*NNB*NDIM, hpsm::AccessMode::In);
  hpsm::View<float> v_variables(variables, nelr*NVAR,
        nelr*NVAR, hpsm::AccessMode::In);
  hpsm::View<float> v_fluxes(fluxes, nelr*NVAR,
        block_size*NVAR, hpsm::AccessMode::Out);
  hpsm::View<float> v_step_factors(step_factors, nelr,
        block_size, hpsm::AccessMode::In);
  hpsm::range<1> rg(nelr);

  funcComputeFlux func(v_ele_sur_ele, v_normals, v_variables, v_fluxes,
        v_step_factors, ff_variable, ff_flux_contrib_momentum_x,
        ff_flux_contrib_momentum_y, ff_flux_contrib_momentum_z,
        ff_flux_contrib_density_energy);
  hpsm::parallel_for(rg, v_step_factors.block_range(), func);

  func.remove_data();
}
```

Fonte: Próprio autor.

ANEXO A – FONTES DE PROGRAMAS COM O PROBLEMA AXPY

Este anexo apresenta os códigos dos programas paralelos implementados através das interfaces de programação OpenMP, HPSM e StarPU para solucionar o problema AXPY.

Figura A.1 – Fonte da aplicação AXPY utilizando OpenMP.

```
1 #include <omp.h>
2 #include "utils.hpp"
3
4 int main(int argc, char **argv) {
5   int size = 16384;
6   float *x, *y, a = 3.41f;
7   x = new float[size];
8   y = new float[size];
9   fillArray(x, size); /* Chamadas definidas em utils.hpp */
10  fillArray(y, size);
11
12  #pragma omp parallel for
13  for(int i=0; i<size; i++)
14    y[i] = x[i] * a + y[i];
15
16  delete[] x; delete[] y;
17  return 0;
18 }
```

Fonte: Próprio autor

Figura A.2 – Fonte da aplicação AXPY utilizando HPSM.

```
#include "hpsm.hpp"
#include "utils.hpp"

using View = hpsm::View<float>;
struct funcAXPY : hpsm::Functor {
  View x, y; float a;

  funcAXPY(View _x, View _y, float _a) : x(_x), y(_y), a(_a) {
    register_data(x, y);
  }

  funcAXPY(const funcAXPY& other) : hpsm::Functor(other),
    x(other.x), y(other.y), a(other.a) {
    clear_data();
    register_data(x, y);
  }

  PARALLEL_FUNCTION
  void operator()(hpsm::index<1> idx) { //Kernel CPU e GPU
    y(idx) = x(idx) * a + y(idx);
  };
};

int main(int argc, char **argv) {
  int size = 16384;
  int block_size = 512;
  float *x, *y, a = 3.41f;
  x = new float[size];
  y = new float[size];
  fillArray(x, size); /* Chamadas definidas em utils.hpp */
  fillArray(y, size);

  /* Inicializa a API e registra os dados */
  hpsm::initialize();
  View view_x(x, size, block_size, hpsm::AccessMode::In);
  View view_y(y, size, block_size, hpsm::AccessMode::InOut);

  /* Declara o functor com o kernel AXPY */
  funcAXPY func(view_x, view_y, a);
  hpsm::range<1> rg(0, size);

  /* Executa o laço paralelo */
  hpsm::parallel_for(rg, view_y.block_range(), func);

  /* Remove os dados e finaliza a API */
  func.remove_data();
  hpsm::finalize();

  delete[] x; delete[] y;
  return 0;
}
```

Fonte: Próprio autor.

Figura A.3 – Fonte da aplicação AXPY utilizando StarPU - Parte 1.

```
1 #include "utils.hpp"
2 #include <starpu.h>
3 #include <cublas.h>
4
5 void axpy_cpu(void *descr[], void *arg) { //Kernel CPU
6   float a        = *((float *)arg);
7   unsigned bs    = STARPU_VECTOR_GET_NX(descr[0]);
8   float *block_x = (float *)STARPU_VECTOR_GET_PTR(descr[0]);
9   float *block_y = (float *)STARPU_VECTOR_GET_PTR(descr[1]);
10
11  for(int i=0; i<bs; i++)
12    block_y[i] = block_x[i] * a + block_y[i];
13 }
14
15 void axpy_gpu(void *descr[], void *arg) { //Kernel GPU
16   float a        = *((float *)arg);
17   unsigned bs    = STARPU_VECTOR_GET_NX(descr[0]);
18   float *block_x = (float *) STARPU_VECTOR_GET_PTR(descr[0]);
19   float *block_y = (float *) STARPU_VECTOR_GET_PTR(descr[1]);
20
21  cublasSaxpy((int)bs, a, block_x, 1, block_y, 1); //Chamada CUBLAS
22 }
23
24 void configure_model(starpu_perfmodel* perf_model) {
25   std::memset(perf_model, 0, sizeof(starpu_perfmodel));
26   const char* model_name = "axpy";
27   perf_model->symbol     = const_cast<char*>(model_name);
28   perf_model->type       = STARPU_HISTORY_BASED;
29 }
30
31 void configure_codelet(starpu_codelet* codelet,
32                        starpu_perfmodel* perf_model) {
33   starpu_codelet_init(codelet);
34   codelet->cpu_funcs[0]      = axpy_cpu;
35   const char* func_name      = "axpy_cpu";
36   codelet->cpu_funcs_name[0] = (char*) func_name;
37   codelet->cuda_funcs[0]     = axpy_gpu;
38   codelet->cuda_flags[0]     = STARPU_CUDA_ASYNC;
39
40   codelet->nbuffers          = 2;
41   codelet->modes[0]          = STARPU_R;
42   codelet->modes[1]          = STARPU_RW;
43   const char* codelet_name   = "axpy";
44   codelet->name              = codelet_name;
45   codelet->model             = perf_model;
46 }
47
48 int main(int argc, char **argv) {
49   int size            = 16384;
50   int block_size      = 512;
51   unsigned num_blocks = size/block_size;
52   float *x, *y, a = 3.41f;
53   x = new float[size];
54   y = new float[size];
55   fillArray(x, size); /* Chamadas definidas em utils.hpp */
56   fillArray(y, size);
```

Fonte: Adaptado dos exemplos da *runtime* StarPU.

Figura A.4 – Fonte da aplicação AXPY utilizando StarPU - Parte 2.

```
57   starpu_data_handle_t handle_y, handle_x;
58   /* Inicializa StarPU */
59   if (ret == starpu_init(NULL))
60     return 77;
61   starpu_cublas_init();
62
63   /* Registra os dados para a StarPU */
64   starpu_vector_data_register(&handle_x, STARPU_MAIN_RAM,
65                               (uintptr_t)x, size, sizeof(float));
66   starpu_vector_data_register(&handle_y, STARPU_MAIN_RAM,
67                               (uintptr_t)y, size, sizeof(float));
68
69   /* Particiona os vetore em blocos */
70   struct starpu_data_filter block_filter = {
71     .filter_func = starpu_vector_filter_block,
72     .nchildren = num_blocks
73   };
74   starpu_data_partition(handle_x, &block_filter);
75   starpu_data_partition(handle_y, &block_filter);
76
77   /* Configura model e codelet */
78   starpu_perfmodel model;
79   configure_model(&model);
80
81   starpu_codelet axpy_cl;
82   configure_codelet(&axpy_cl, &model);
83
84   /* Submissão das tarefas */
85   for(unsigned b = 0; b < num_blocks; b++) {
86     struct starpu_task *task = starpu_task_create();
87     task->cl          = &axpy_cl;
88     task->cl_arg      = &a;
89     task->cl_arg_size = sizeof(a);
90     task->tag_id      = b;
91
92     task->handles[0] = starpu_data_get_sub_data(handle_x, 1, b);
93     task->handles[1] = starpu_data_get_sub_data(handle_y, 1, b);
94
95     if(starpu_task_submit(task) == -ENODEV)
96       break;
97   }
98
99   /* Aguarda o retorno das tarefas e desregistra os dados da StarPU */
100  starpu_task_wait_for_all();
101  starpu_data_unpartition(handle_x, STARPU_MAIN_RAM);
102  starpu_data_unpartition(handle_y, STARPU_MAIN_RAM);
103  starpu_data_unregister(handle_x);
104  starpu_data_unregister(handle_y);
105
106  /* Finaliza StarPU */
107  starpu_shutdown();
108
109  delete[] x; delete[] y;
110  return 0;
111 }
```

Fonte: Adaptado dos exemplos da *runtime* StarPU.