Uma API em linguagem C++ para programas com laços paralelos e suporte a multi-CPUs e multi-GPUs

Daniel Di Domenico[1], **João V. F. Lima**[1]

[1]Universidade Federal de Santa Maria (UFSM)
Santa Maria – RS – Brasil

{ddomenico,jvlima}@inf.ufsm.br

Resumo. *Este artigo apresenta uma API C++ de alto nível para a implementação de programas paralelos utilizando laços e reduções. Ele visa suprir a falta de APIs que suportam a construção de aplicações que possam ser processadas simultaneamente em multi-CPUs e multi-GPUs. A hipótese levantada estima que aplicações científicas podem valer-se do processamento heterogêneo em multi-CPUs e multi-GPUs para alcançar um desempenho superior em relação ao uso de apenas um acelerador. Os resultados obtidos a partir de experimentos com mini-aplicações científicas desenvolvidas utilizando a nova API sugerem que o processamento combinando CPUs e GPUs pode trazer ganhos de desempenho.*

Abstract. *This article presents a high-level C++ API to implement parallel programs using loops and reductions. It intends to provide a solution for the gap of APIs that support the developing of applications which can be simultaneously processed by multi-CPUs and multi-GPUs. Our hypothesis estimates that scientific applications can explore heterogeneous processing in multi-CPUs and multi-GPUs to achieve a better performance than exploring just an accelerator. Results obtained from experiments with scientific mini-applications developed applying the new API suggest that combining CPUs and GPUs processing can lead to performance gains.*

1. Introdução

O processamento de alto desempenho (PAD) possui grande importância na computação, principalmente para a execução de aplicações científicas. Neste sentido, o paralelismo é um relevante caminho para alcançá-lo, sendo que sua utilização foi intensificada a partir do prenúncio do fim da Lei de Moore e da criação dos processadores *multicore*. No entanto, a codificação de programas paralelos possui um grau de dificuldade maior em relação à codificação convencional. Apesar de existirem algumas ferramentas que facilitam a 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 para explorar CPUs e GPUs, estudos recentes propuseram novas interfaces e modelos de programação de alto nível. Entre eles, alguns empregaram a linguagem C++, como

o Kokkos [Edwards et al. 2012], que possibilita codificar um programa para ser executado exclusivamente por CPUs ou por uma GPU. Outra interface semelhante ao Kokkos é a C++ AMP [Gregory and Miller 2012], porém ela visa a execução da aplicação paralela somente em um acelerador. Ainda, há o modelo de programação Phalanx [Garland et al. 2012], que permite explorar de forma exclusiva as CPUs ou GPUs de ambientes distribuídos. Neste cenário, percebe-se a não existência de APIs (*Application Programming Interfaces*) de alto nível que permitam a construção de programas que possam ser processados simultaneamente em múltiplas CPUs e múltiplas GPUs. Considerando que já existem ferramentas que possuem a capacidade de explorar o processamento heterogêneo em multi-CPUs e multi-GPUs (como a XKaapi [Gautier et al. 2013], a StarPU [Hugo et al. 2013] e o OmpSs [Duran et al. 2011]), uma API C++ que possua este suporte é um problema de pesquisa que pode ser explorado.

Este trabalho descreve uma API C++ com laços e reduções paralelas para arquiteturas multi-CPUs e multi-GPUs por meio de diferentes *back-ends* paralelos. A API foi utilizada na paralelização de mini-aplicações científicas, sendo que as mesmas foram executadas a fim de demonstrar o desempenho alcançado. Nossa hipótese estima que aplicações científicas podem valer-se do processamento heterogêneo em multi-CPUs e multi-GPUs para alcançar um desempenho superior em relação ao uso de apenas um acelerador.

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 garante 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 C++ apresentada.

O restante deste manuscrito está dividido da seguinte maneira. Na seção 2, são descritos os trabalhos relacionados. Depois, na seção 3, a API implementada é detalhada. A seção 4 apresenta os resultados de desempenho da execução de mini-aplicações científicas implementadas utilizando a nova API. Na sequencia, a seção 5 interpreta e discute os resultados. Por fim, na seção 6 é apresentada a conclusão e os trabalhos futuros.

2. Trabalhos relacionados

Em PAD 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. A Tabela 1 sumariza as características das interfaces abordadas nesta seção.

Entre as ferramentas que possuem funcionalidades de processamento heterogêneo em multi-CPUs e em multi-GPUs, podemos citar Kaapi [Gautier et al. 2013], StarPU [Augonnet et al. 2011] e OmpSs [Bueno et al. 2013]. Delas, apenas a Kaapi possui uma interface C++. Como característica comum, as três ferramentas demandam do programador a escrita de duas versões do código, uma para CPU e uma para GPU (em CUDA ou OpenCL). Além disso, elas são baseadas em tarefas assíncronas, recurso que não possui a mesma facilidade de uso se comparado a um laço paralelo.

Tabela 1. Características das interfaces relacionadas.

Característica	Ferramenta/API								
	Kaapi++	StarPU	OmpSS	OpenMP 4.0	Kokkos	C++ AMP	Thrust	Phalanx	HPX
multi-CPUs	✓	✓	✓	✓	✓			✓	✓
GPU	✓	✓	✓	✓	✓	✓	✓	✓	
multi-GPUs	✓	✓	✓					✓	
Proces. CPU+GPU	✓	✓	✓						
Alto nível				✓	✓	✓	✓	✓	✓

O OpenMP, considerado o padrão *de facto* para o desenvolvimento de algoritmos paralelos em memória compartilhada [Adcock et al. 2013, Duran et al. 2009], suporta aceleradores a partir da sua versão 4.0 [OpenMP 2016]. No entanto, este suporte não permite o processamento em multi-GPUs.

No campo das APIs C++, uma das interfaces que mais assemelha-se com a API apresentada neste estudo é a Kokkos [Edwards et al. 2012]. Ela possibilita o processamento de laços e reduções paralelas codificadas através de um objeto C++ tanto em CPU como em GPU sem a necessidade de alterações no fonte do *kernel* da aplicação. Apesar disso, o processamento é realizado apenas nas CPUs ou em uma GPU, visto que a Kokkos também não possui suporte para multi-GPUs.

Seguindo a linha de interfaces com recursos para GPUs, outra que compara-se a Kokkos é a biblioteca C++ AMP (*Accelerated Massive Parallelism*) [Gregory and Miller 2012]. Ela também visa simplificar a implementação de programas paralelos voltados a explorar aceleradores, onde laços podem ser facilmente adaptados para a execução paralela em um acelerador. O mesmo ocorre com a biblioteca Thrust [Thrust 2016]. Ela possui uma implementação da C++ *Standard Library* em CUDA, e desta forma, o paralelismo é alcançado em alto nível, apenas passando objetos C++ (como *containers*) para suas rotinas (como ordenações, reduções e pesquisas). Porém, a Thrust não permite muita flexibilidade no *kernel* da aplicação, visto que as rotinas paralelas estão atreladas às chamadas das funções da *Standard Library*.

Com suporte não só a CPUs e GPUs como também a arquiteturas de memória distribuída, o modelo de programação Phalanx [Garland et al. 2012] é uma biblioteca C++ que possibilita alcançar paralelismo em máquinas heterogêneas (inclusive multi-nó) através do uso de tarefas assíncronas. No entanto, apesar de ser multi-CPU e multi-GPU, o Phalanx não permite o processamento heterogêneo, ou seja, ele realiza o processamento em apenas uma das arquiteturas por vez.

Por fim, destacamos uma API C++ de alto nível que não possui suporte para aceleradores. A HPX (*High Performance ParalleX*) [Heller et al. 2013] 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.

3. API C++

A API C++ apresentada neste trabalho oferece recursos à construção de programas paralelos voltados ao processamento em CPUs e GPUs. Sua principal característica é a execução de iterações de laços e reduções paralelas em CPUs e GPUs. Além disso, ela permite executar uma mesma rotina paralela por meio de diferentes *back-ends*, o que

garante que um programa que será executado em CPU e GPU seja implementado com apenas uma versão de código.

O restante desta seção irá detalhar a nova API. Primeiro, será descrito como o uso de *back-ends* facilita a construção de códigos que podem ser executados em diferentes arquiteturas (seção 3.1). Na sequência, a seção 3.2 mostrará como os dados de um programa são mapeados para a API a fim de serem processados em paralelo. A terceira e última seção (3.3) demostrará como é realizada a execução paralela através de laços ou reduções.

3.1. Execução através de *back-ends*

A API C++ foi modelada de modo a permitir que o processamento paralelo de um programa pudesse ser realizado por meio de diferentes *back-ends* (ou bibliotecas de programação). Porém, durante a codificação da aplicação, este recurso fica transparente ao programador, pois a API garante a portabilidade do código paralelo independente da arquitetura na qual ele será executado. Isso é possível visto que existe uma camada de classes da API que abstrai o acesso do programa paralelo aos *back-ends*, conforme detalhado na Figura 1. A seleção do *back-end* para a execução do programa é feita após a sua implementação, ou seja, em tempo de compilação. Apenas um *back-end* pode ser selecionado por compilação. Internamente, a ligação entre as classes da API e as chamadas específicas do *back-end* selecionado foi implementada com metaprogramação.

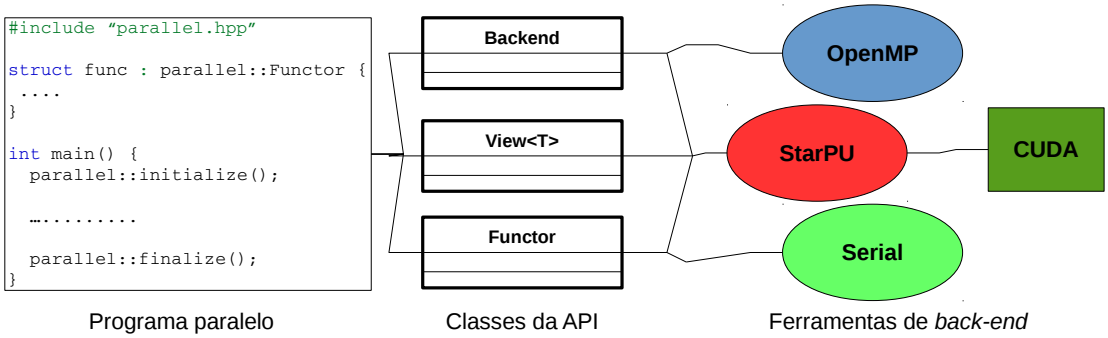

Figura 1. Modelagem da API através de *back-ends*.

O desenvolvimento de um código sem as especifidades de uma determinada arquitetura é uma importante vantagem trazida pelo uso de *back-ends*. Este fato, aliado à execução paralela utilizando laços e reduções, contribuem para que a API seja caracterizada de alto nível.

Atualmente, a API suporta três *back-ends*, sendo dois deles paralelos: StarPU e OpenMP. O *back-end* Serial, que como o nome sugere não é paralelo, pode ser utilizado para realizar o processamento de um programa em blocos (conforme descrito na seção 3.2). 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*.

3.2. Mapeamento dos dados

O processamento paralelo realizado pela API C++ 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*. Ao realizá-lo, o dado será registrado no *back-end* e poderá ser particionado em blocos. É possível mapear para uma *View* estruturas de *arrays* (como `int*` ou `float*`) da linguagem C, que 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. Isto está exemplificado nas Figuras 3 e 4. Na Figura 2 pode-se visualizar um código onde são mapeadas duas estruturas para as *Views* da API.

```
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 parallel::View<float> view_vector(vector, size, block_size, parallel::AccessMode::In);
11
12 //'View' mapeando uma matriz - 2 Dimensões
13 parallel::View<float> view_matrix(matrix, lines, columns, block_size,
14             parallel::PartitionMode::Matrix_Vert_Horiz, //Particionamento da matriz
15             parallel::AccessMode::InOut);
```

Figura 2. Mapeamento de *arrays* para *Views* da API.

Além do modo de mapeamento, 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:

- **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 2, 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 copias de memória para um acelerador (GPU). 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.

3.3. Execução paralela

Um programa paralelo utilizando a API deve ser implementado para executar laços (`parallel_for`) ou reduções paralelas (`parallel_reduce`). Para isso, é preciso mapear a rotina paralela para um *Functor*, uma classe C++ que sobrescreve o seu operador de aplicação (`operator()`) [Stroustrup 2013]. As Figuras 3 e 4 demonstram a declaração de *Functors* no formato requisitado pela API, bem como a sua utilização na chamada das funções paralelas.

```
1 using View = parallel::View<float>;
2
3 //'struct funcAXPY' deve herdar a classe
4 //'parallel::Functor' para ter acesso
5 //aos métodos do back-end
6 struct funcAXPY : parallel::Functor {
7   View x, y; float a;
8
9 //Construtor deve registrar as 'View's
10 //no back-end chamando 'register_data'
11   funcAXPY(View _x, View _y, float _a) :
12     x(_x), y(_y), a(_a) {
13     register_data(x, y);
14   }
15
16 //Operador de cópia necessário para copiar
17 //o objeto 'struct funcAXPY' para a GPU
18   funcAXPY(const funcAXPY& f) :
19     parallel::Functor(f),
20     x(f.x), y(f.y), a(f.a) {
21     clear_data(); register_data(x, y);
22   }
23
24 //Operador de aplicação que executa de
25 //forma paralela o kernel do programa.
26 //'PARALLEL_FUNCTION' é uma macro de
27 //compilação, como '__device__' para CUDA.
28 //'parallel::index<1>' indica 1 dimensão
29 //para o acesso às 'View's
30   PARALLEL_FUNCTION
31   void operator()(parallel::index<1> i) {
32     y(i) = x(i) * a + y(i);
33   };
34 };
35
36 /*** ------ PROGRAMA PRINCIPAL ------ ***/
37 //Chamada do laço paralelo, onde
38 //percorre-se o range<1> (0..N)
39 funcAXPY func(view_x, view_y, 3.41);
40 parallel::range<1> rg(0, N);
41 parallel::parallel_for(rg,
42   view_y.block_range(), func);
43
44 func.remove_data(); //Remoção das 'View's
```

Figura 3. `parallel_for`.

```
1 //Alias para o tipo da View
2 using View = parallel::View<float>;
3
4 //Declaração do Functor
5 struct funcRed : parallel::Functor {
6   View mat;
7
8 //Construtor e operador de cópia
9   funcRed(View _mat) : mat(_mat) {
10     register_data(mat);
11   }
12
13   funcRed(const funcRed& f) :
14     parallel::Functor(f), mat(f.mat) {
15     clear_data(); register_data(mat);
16   }
17
18 //'parallel::index<2>' indica 2 dimensões
19 //para o acesso às 'View's
20   PARALLEL_FUNCTION
21   void operator()(parallel::index<2> i) {
22
23 //'parallel::atomic_add' para proteger o
24 //acesso pela GPU ao ponteiro da variável
25 //de redução 'reduction_var<float>()'
26     parallel::atomic_add(
27       reduction_var<float>(), Mat(i));
28   };
29 };
30
31 /*** ------ PROGRAMA PRINCIPAL ------ ***/
32 float red; //Variável de redução
33
34 //Percorre-se os range<2> (0..N) e (0..N)
35 //para as linhas e colunas, reduzindo a
36 //variável 'red' usando 'ReduxMode::Sum'
37 funcRed func(view_mat);
38 parallel::range<2> rg(0, N, 0, N);
39 parallel::parallel_reduce(rg_mat,
40   MAT.block_range(), func_mat, red,
41   parallel::ReduxMode::Sum);
42
43 //Remoção das 'View's do back-end
44 func_mat.remove_data();
```

Figura 4. `parallel_reduce`.

Na Figura 3, 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 2, bem como sua passagem para o *Functor* na linha 39, o laço paralelo é disparado na linha 41. Nele, será executada a chamada do `operator()` do `funcAXPY` para cada índice contido no intervalo do *range* definido na linha 40. 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.

A Figura 4 apresenta um programa que executa uma redução a partir dos valores de uma matriz. Seu código e forma de execução são semelhantes aos detalhados na Figura 3, porém há dois itens que precisam ser destacados. Primeiro, a variável de redução deve ser passada para a função `parallel_reduce` (linha 39) 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 27), sendo `T` seu tipo.

4. Resultados

Nesta seção são apresentados os resultado obtidos com experimentos utilizando a nova API C++. Estes experimentos visaram utilizar mini-aplicações científicas, onde seus dados foram mapeados para as *Views* da API de diferentes maneiras. Com matrizes, foi selecionado o programa Hotspot. Já com vetores, foram escolhidas as aplicações LJ-Forces e N-Body. Os resultados contemplam os dois *back-ends* paralelos da API (OpenMP e StarPU), sendo que para o StarPU foram aplicados testes com somente CPUs e combinando CPUs+GPUs.

Além dos *back-ends* da API, os resultados também possuem dados de experimentos com a *runtime* Kaapi. Para obtê-los, utilizou-se como base os programas gerados a partir do *back-end* OpenMP, apenas substituindo-se as chamadas de *runtime* para a LIBKOMP [Broquedis et al. 2012]. As configurações do Kaapi para as execuções foram as mesmas utilizadas no trabalho de [Virouleau et al. 2016].

A plataforma onde os experimentos foram executados é uma máquina Dell PowerEdge T630 equipada com dois processadores Intel Xeon E5-2697 v3 de 2,60 GHz com 14 núcleos (total de 28 *cores*) e três 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 sistema operacional é Debian GNU/Linux versão jessie-x64-std, compilador g++ versão 5.4.0, CUDA versão 7.5, StarPU revisão 18390 do repositório SVN[1] e X-Kaapi no *branch public/europar2016* do repositório Git[2]. Para as execuções CPUs+GPUs, foi aplicado o escalonador DMDA da *runtime* StarPU. Ele é similar ao escalonador HEFT (*Heterogenous Earliest First Time*), porém considera o tempo das transferências de dados.

A Figura 5 mostra o *speedup* das três aplicações em relação às suas versões sequenciais. Os resultados experimentais de desempenho são uma média de no mínimo 30 execuções com intervalo de confiança de 95% representado por meio de uma linha vertical escura em cada média.

4.1. Hotspot

A primeira aplicação utilizada foi a Hotspot, uma ferramenta de simulação térmica empregada para estimar a temperatura de um processador baseando-se na sua arquitetura e

[1] svn://scm.gforge.inria.fr/svn/starpu/trunk

[2] https://scm.gforge.inria.fr/anonscm/git/kaapi/xkaapi.git

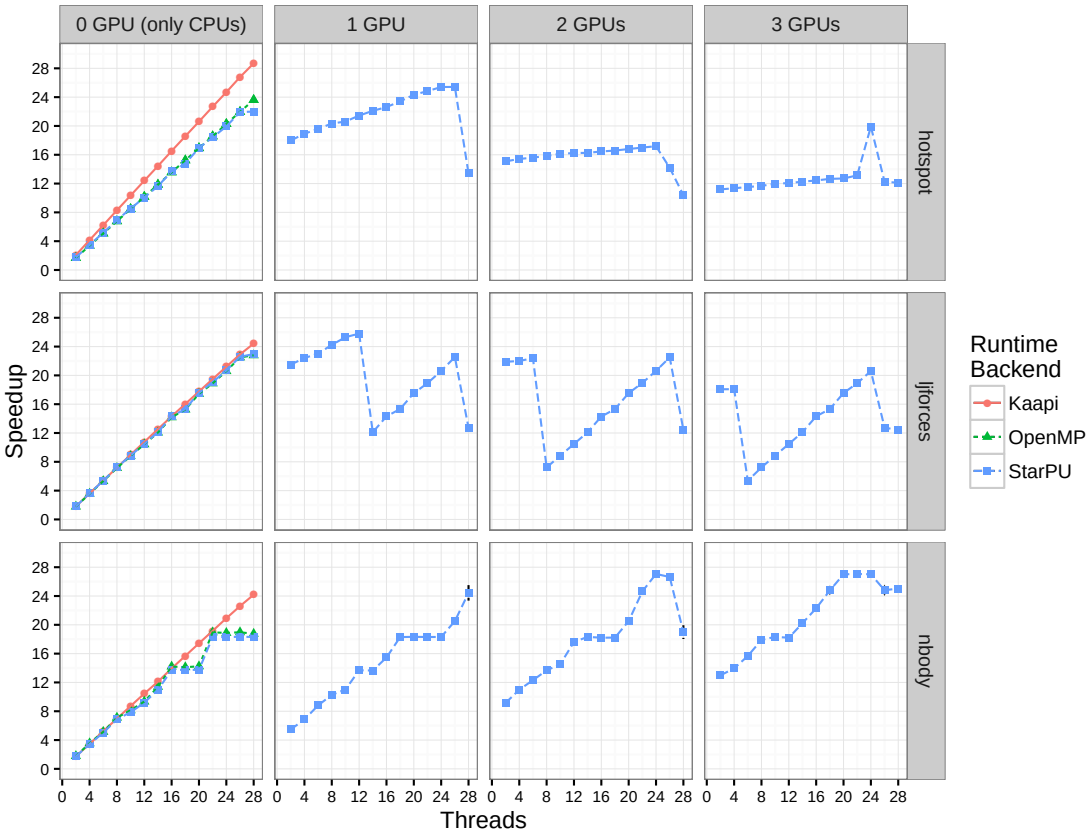

Figura 5. Resultados - *speedup* das aplicações.

em medições de energia (também simuladas) [Che et al. 2009]. Para estes experimentos, adaptou-se a versão disponível na *suite* Rodinia[3], sendo utilizados como parâmetros de entrada uma matriz (representando o processador) de ordem 16.384, blocos de ordem 1.024 e 20 iterações.

A parte superior da Figura 5 mostra o *speedup* obtido. É possível observar que o uso de mais GPUs não melhorou o desempenho como esperado, pois os ganhos ao empregar-se 2 e 3 GPUs foram menores do que os obtidos por apenas 1 GPU. Considerando-se apenas 1 GPU, houve um ganho crescente a medida que o número de *threads* foi aumentado, exceto para 28 *threads*, onde ocorreu uma queda de 47,3% em relação a execução com 26 *threads*.

Para os resultados com somente CPUs, destaca-se o desempenho alcançado pelo Kaapi. Ele foi escalável a exemplo do que ocorre com OpenMP e StarPU, porém foi superior a eles para todos os números de *threads*. A maior diferença foi obtida com 28, onde o *speedup* obtido superou o OpenMP em 21,5% e o do StarPU em 30,5%. Com 26 *threads*, o Kaapi superou inclusive o StarPU com o uso de 1 GPU em 5,4%.

4.2. LJ-Forces

O LJ-Forces é um *kernel* de dinâmica molecular que calcula forças sobre átomos a partir do método de Lennard-Jones (LJ). Através de laços que percorrem os átomos, as forças

[3]http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators

são calculadas sobre pares que estão próximos, ou seja, a uma distância onde são considerados vizinhos [Edwards et al. 2014]. A versão utilizada neste trabalho foi adaptada da disponível no repositório da API Kokkos[4]. A configuração de entrada utilizada nos experimentos foi de 6.912.000 átomos divididos em blocos de 27.000, sendo o cálculo realizado por 10 iterações.

Os resultados do *kernel* LJ-Forces apresentados na parte central da Figura 5 indicam que, assim como para o Hotspot (seção 4.1), o uso das GPUs da plataforma não resultou nos ganhos esperados. Aqui, este fato ocorre inclusive no cenário com 1 GPU, onde a partir de 14 *threads* as tarefas passam a ser executadas apenas por CPUs, resultando numa redução considerável do *speedup* de 12 para 14 *threads* (53,1%).

A Figura 6 mostra os rastros de execução com StarPU para 12 e 14 *threads*. Em sua parte superior (12 *threads*) as tarefas foram executadas pelas CPUs e GPU, enquanto na parte inferior (14 *threads*) a execução ocorreu somente em CPUs. A execução das tarefas apenas pelas CPUs também aconteceu ao explorar 2 e 3 GPUs, porém este evento inicia com 6 e 4 *threads*, respectivamente.

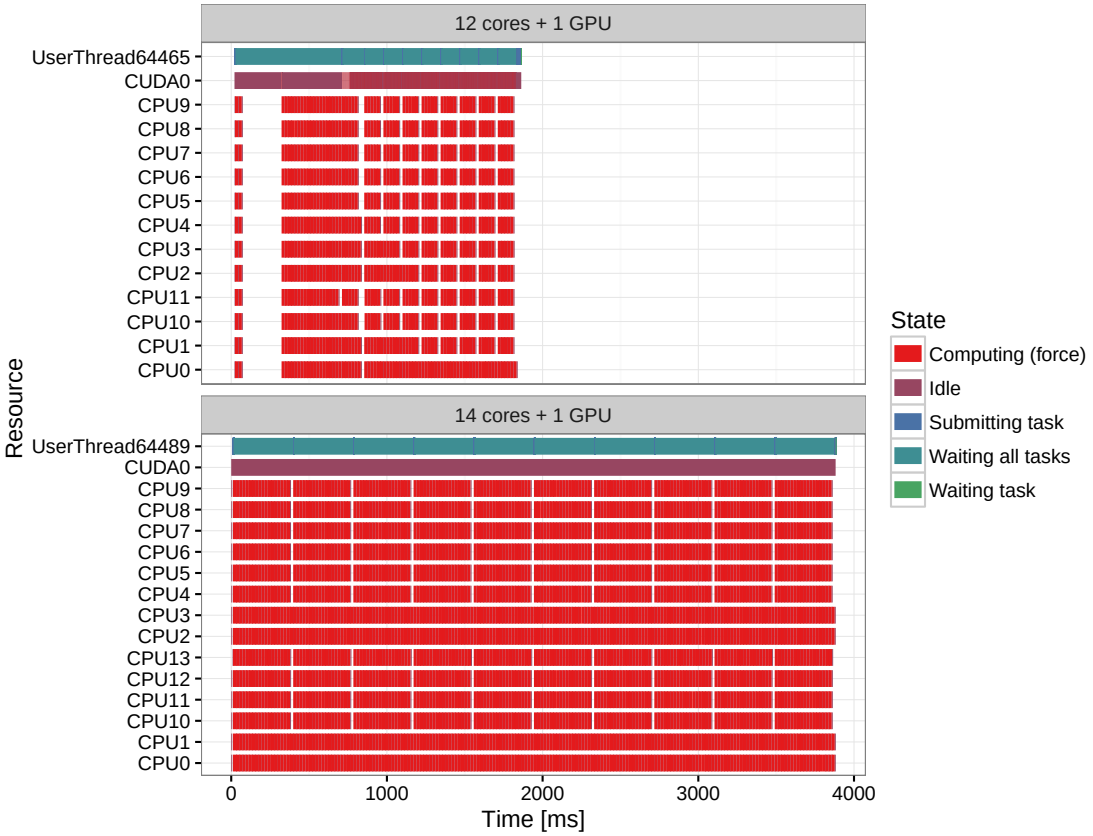

Figura 6. Rastro StarPU do LJ-Forces - CPUs+1GPU com 12 e 14 *threads*.

Os *back-ends* para CPUs apresentaram *speedups* próximos ao ideal. Os resultados com OpenMP e StarPU foram semelhantes, ao passo que o resultado com Kaapi foi superior aos outros dois *back-ends* a medida que o número de *threads* foi aumentado.

[4]Kokkos: https://github.com/kokkos/kokkos

4.3. N-Body

A simulação de N-Body calcula a evolução de um sistema de corpos que interagem entre si. Esta 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[5]. Como entrada, foram empregadas 65.536 partículas, bloco de 1.024 e 5 iterações.

O melhor desempenho foi obtido com 3 GPUs e 22 *threads* (*speedup* de 27,1x), apesar de as execuções com 20 e 24 mostrarem dados minimamente inferiores (*speedup* de 27,0x). Destacam-se também os desempenhos obtidos com 2 GPUs para 24 e 26 *threads* (*speedups* de 27,0x e 26,6x, respectivamente). A partir de 26 *threads*, o *speedup* diminuiu com 3 GPUs (8,7% para 26 e 7,6% para 28 em relação a 24 *threads*). O mesmo sucedeu-se com 28 *threads* mais 2 GPUs, onde houve perda de 30,0% em relação a 26 *threads*.

Considerando apenas o uso de CPUs, foi possível observar um saturamento nos ganhos a partir de 22 *threads* para as execuções com OpenMP e StarPU, o que acabou resultando na estabilização dos *speedups* máximos obtidos ao utilizar GPUs. No entanto, isso não aconteceu nas execuções utilizando a *runtime* Kaapi, que seguiu apresentando ganhos de forma linear.

5. Discussão

A hipótese levantada confirmou-se parcialmente, visto que o processamento em multi-CPUs e multi-GPUs trouxe ganhos apenas para a aplicação N-Body. Nos programas Hotspot e LJ-Forces, a melhoria no desempenho resultante da combinação de arquiteturas heterogêneas ocorreu apenas na combinação de CPUs com 1 GPU devido a limitações de escalonamento do *back-end* StarPU.

Os experimentos com o *back-end* StarPU indicam que o processamento simultâneo em CPUs e GPUs não ocorreu de acordo com a hipótese deste trabalho para as aplicações Hotspot e LJ-Foces. De acordo com o relatado na seção 4.2 e ilustrado pela Figura 6, o escalonamento DMDA do StarPU parece ter limitações em arquiteturas NUMA. As decisões gulosas de escalonamento do algoritmo DMDA objetivam o menor tempo de término localmente, mas mostraram-se ineficientes globalmente nos experimentos. Sugere-se que esse fato justifique a queda no *speedup* alcançado pelo LJ-Forces explorando 1 GPU a partir de 14 *threads*, bem como a perda de desempenho para o Hotspot e Lj-Forces com multi-GPUs.

Além disso, constatou-se a diminuição do *speedup* a partir do uso de 26 *threads* quando o processamento é realizado nas GPUs. A *runtime* StarPU dedica um núcleo do processador para controlar cada GPU do sistema [Augonnet et al. 2011]. Neste sentido, caso eles sejam utilizados também para processar, pode acontecer uma queda no desempenho, pois o escalonamento das tarefas para as GPUs tende a ser prejudicado.

Em relação ao processamento somente com CPUs, os resultados ficaram dentro do previsto. O *back-end* Kaapi apresentou resultados superiores aos *back-ends* OpenMP e StarPU nas três aplicações devido a técnicas para explorar arquiteturas NUMA im-

[5]https://pm.bsc.es/projects/bar/wiki/Applications

plementadas na *runtime* [Virouleau et al. 2016]. A perda de escalabilidade com *back-ends* OpenMP e StarPU para 28 *threads* justifica-se por suas respectivas limitações. A implementação OpenMP do g++ escalona tarefas por meio de uma lista centralizada com acesso concorrente, o que pode resultar em contenção de acesso. No caso do *back-end* StarPU, as decisões de escalonamento não consideram características de arquiteturas NUMA.

6. Conclusão

Este trabalho apresentou uma API C++ para a implementação de programas voltados ao processamento de laços e reduções paralelas. Ele visa suprir a falta de APIs de alto nível que facilitem a construção de aplicações que possam ser processadas simultaneamente em multi-CPUs e multi-GPUs. A hipótese levantada sugere que o processamento em ambas arquiteturas pode resultar em melhoria do desempenho, principalmente para aplicações científicas.

A API C++ proposta permitiu a codificação em alto nível de mini-aplicações científicas para a execução simultânea em uma plataforma composta por multi-CPUs e multi-GPUs. Os resultados obtidos a partir destas aplicações confirmaram a hipótese parcialmente, visto que os ganhos com multi-GPUs não ocorrerem em todos os experimentos executados devido a limitações de escalonamento.

Como trabalhos futuros pretende-se otimizar as rotinas da API voltadas à execução em GPUs com mecanismos de escalonamento de tarefas. Além disso, pretende-se adicionar suporte ao uso simultâneo de dois *back-ends*, um para CPUs e outro para GPUs.

Agradecimentos

Agradecemos o suporte da NVIDIA com a doação de uma GPU NVIDIA GTX Titan X utilizada nos experimentos preliminares deste trabalho. Também gostaríamos de agradecer a agência de fomento CAPES pelo apoio recebido através de bolsa de estudo.

Referências

Adcock, A. B., Sullivan, B. D., Hernandez, O. R., and Mahoney, M. W. (2013). Evaluating OpenMP Tasking at Scale for the Computation of Graph Hyperbolicity. In *Proceedings of the OpenMP in the Era of Low Power Devices and Accelerators*, 9th International Workshop on OpenMP, IWOMP 2013, pages 71–83, Canberra, ACT, Australia. Springer Berlin Heidelberg.

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198.

Broquedis, F., Gautier, T., and Danjean, V. (2012). 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*, pages 102–115, Rome, Italy.

Bueno, J., Martorell, X., Badia, R. M., Ayguadé, E., and Labarta, J. (2013). Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces. In *Proceedings of the 27th International Conference on Supercomputing*, ICS '13, pages 359–368, Eugene, Oregon, USA. ACM.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S. H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54.

Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(2):173–193.

Duran, A., Teruel, X., Ferrer, R., Martorell, X., and Ayguade, E. (2009). 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*, pages 124–131.

Edwards, H. C., Sunderland, D., Porter, V., Amsler, C., and Mish, S. (2012). Manycore performance-portability: Kokkos multidimensional array library. *Scientific Programming*, 20(2):89–114.

Edwards, H. C., Trott, C. R., and Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

Garland, M., Kudlur, M., and Zheng, Y. (2012). Designing a Unified Programming Model for Heterogeneous Machines. In *SC '12: Proc. Conference on High Performance Computing Networking, Storage and Analysis*.

Gautier, T., Lima, J. V. F., Maillard, N., and Raffin, B. (2013). XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 1299–1308, Washington, DC, USA. IEEE Computer Society.

Gregory, K. and Miller, A. (2012). *C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®*. Developer Reference. Microsoft Press.

Heller, T., Kaiser, H., and Iglberger, K. (2013). Application of the ParalleX Execution Model to Stencil-based Problems. *Comput. Sci.*, 28(2-3):253–261.

Hugo, A.-E., Guermouche, A., Wacrenier, P.-A., and Namyst, R. (2013). 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)*, pages 1050–1059.

OpenMP (2016). OpenMP Application Program Interface Version 4.5. http://www.openmp.org/mp-documents/openmp-4.5.pdf. Acesso em: 19 jul 2016.

Stroustrup, B. (2013). *The C++ Programming Language*. Addison-Wesley Professional, 4th edition.

Thrust (2016). http://thrust.github.io/ Acesso em: 21 mai 2016.

Virouleau, P., Broquedis, F., Gautier, T., and Rastello, F. (2016). Using data dependencies to improve task-based scheduling strategies on NUMA architectures. In *Euro-Par 2016*, Euro-Par 2016, Grenoble, France.