G-PETo: Um Framework para Troca Direta e Transparente de Dados entre Aplicações Filtro-Fluxo em Arquiteturas Multi-GPUs*

Millas Avelar , Guilherme Andrade , Leonardo Rocha

Universidade Federal de São João del Rei, Brazil

{millas,gandrade,lcrocha}@ufsj.edu.br

Abstract. *In this work we propose, implement and test a generic abstraction for direct and transparent data exchange of filter-stream applications in heterogeneous cluster (interconnected computers) composed by multiples GPUs (Graphics Processing Units) architectures. This abstraction allows all the low-level implementation details related to GPU communication and the control related to the location of the filters between available nodes to be performed transparently to the programmers. This work is consolidated in a framework, which we call G-PETo, and our evaluation results show that it provides an abstraction layer for programmers without compromising the overall application performance.*

Resumo. *Neste trabalho propomos e avaliamos uma abstração genérica para a troca de dados direta e transparente em aplicações filtro-fluxo executadas em cluster (computadores interligados) heterogêneos, compostos por múltiplas placas aceleradoras gráficas (GPUs). Esta abstração permite que todos os detalhes de implementação de baixo nível, relacionados à comunicação entre GPUs e o controle relacionado à localização dos filtros, sejam realizados de forma transparente para os programadores. Este trabalho está consolidado em um framework, o qual denominamos G-PETo, e nossos resultados de avaliação demonstram que ele é capaz de fornecer uma camada de abstração aos programadores sem comprometer o desempenho geral da aplicação.*

1. Introdução

Observamos um crescente uso de arquiteturas heterogêneas, caracterizadas por diferentes tipos de unidades de processamento, tais como múltiplos processadores (arquitetura *multicore*) e placas aceleradoras gráficas (GPUs). Além dos tradicionais ambientes de programação paralela *multicore* [Gabriel et al. 2004], novas bibliotecas e ambientes [Stone et al. 2010, Wienke et al. 2012] que permitem o desenvolvimento de aplicações em unidades de processamento alternativas vêm sendo propostos. Apesar desses avanços, existem cenários em que uma única implementação paralela não é suficiente, como exemplo, aplicações compostas por diferentes passos de processamento, onde cada passo recebe uma determinada entrada e produz uma saída que será usada como entrada para outra etapa. Estas aplicações são tipicamente modeladas usando o paradigma filtro-fluxo [Acharya et al. 1998], cujos passos do processamento são representados por filtros, instanciados em uma ou mais máquinas, e a comunicação de dados é representada

*Esse trabalho foi parcialmente financiado por CNPq, CAPES, FINEP, Fapemig, e INWEB.

por fluxo de dados. A eficiência dessas aplicações é alcançada, principalmente, por uma distribuição paralela adequada desses filtros, explorando de forma coordenada o conjunto de computadores interligados (*cluster*).

Tradicionalmente, em ambientes para programação filtro-fluxo encontrados na literatura, a comunicação é feita principalmente utilizando o padrão *Message Passing Interface (MPI) [Gabriel et al. 2004]*, que pode ser aplicado tanto na comunicação entre filtros instanciados na mesma máquina ou em diferentes máquinas. Quando a execução de uma aplicação é feita em um *cluster* composto por máquinas heterogêneas, em que um filtro particular pode ser processado em uma GPU, por exemplo, a comunicação MPI entre os filtros pode exigir movimentações extras de dados. Neste caso, os dados a serem transmitidos devem ser primeiro transferidos da memória global da GPU para a memória principal do computador e, em seguida, enviados via MPI para o filtro de destino. A fim de evitar estas movimentações excessivas, existem na literatura algumas estratégias que permitem que essa comunicação entre GPUs seja direta, como por exemplo: (i) *GPUDirect RDMA [Shainer et al. 2011]* para transferir dados entre GPUs localizadas em uma mesma máquina; (ii) *Cuda Aware MPI (OpenMPI + GPUDirect RDMA) [Wu et al. 2016]* focada principalmente na comunicação entre GPUs em diferentes máquinas. Apesar dessas tecnologias, para aplicações filtro-fluxo, essa comunicação direta é ainda um grande desafio, uma vez que os filtros podem ser instanciados na mesma máquina ou em máquinas diferentes, tornando o programador responsável por controlar onde cada filtro é instanciado e decidindo qual a estratégia de comunicação a ser usada. Além disso, essa abordagem precisa ser ajustada e reimplementada a cada nova configuração de filtros.

Portanto, neste trabalho propomos uma abstração genérica para troca de dados direta entre GPUs em cenários de aplicações filtro-fluxo. Nossa proposta torna transparente para o programador os detalhes de implementação de baixo nível relacionados à comunicação GPU e, além disso, retira do programador a responsabilidade de controlar a localização dos filtros e de escolher qual estratégia de comunicação utilizar. Consolidamos nossa proposta como um framework filtro-fluxo, o qual denominamos *G-PETo* (**G***PU* - **P***arallel and* **E***fficient Data* **T***ransfer* **O***f filter Stream Applications*). Nossa proposta também fornece uma API para auxiliar a implementação dos filtros, bem como estabelece um padrão para representar o fluxo de dados entre os filtros. Em nossa avaliação, primeiro comparamos diferentes estratégias de comunicação entre GPUs, tanto intra-node (GPUs em um mesmo computador) quanto inter-node (GPUs em computadores diferentes), com objetivo de justificar as escolhas feitas para o framework. Em seguida, avaliamos a eficiência da nossa proposta implementando uma aplicação real usando o *G-PETo*. Comparando-a com uma implementação manual, que usa diretamente as tecnologias existentes para movimentar dados entre GPUs, mostramos que o tempo da aplicação real é praticamente o mesmo para ambas as implementações. Este fato demonstra que o *G-PETo*, além de fornecer uma camada de abstração capaz de facilitar o processo de programação, controla automaticamente a instanciação dos filtros e não compromete o desempenho da aplicação como um todo, correspondendo à nossa maior contribuição.

A implementação do framework, as execuções dos experimentos e a avaliação dos resultados foram realizadas pelo aluno Millas Násser, sob a orientação do professor Leonardo Rocha. Esse trabalho contou com a colaboração do aluno de pós-graduação Guilherme Andrade no estudo detalhado das tecnologias de comunicação entre GPUs.

2. Trabalhos Relacionados

Filtro-fluxo [Acharya et al. 1998] é um dos paradigmas mais apropriado para trabalhar, de forma coordenada, com *clusters*, explorando o desempenho individual de cada computador, bem como a cooperação paralela entre eles. Encontramos na literatura várias propostas de ambientes que fornecem ferramentas eficientes capazes de lidar com os diferentes desafios relacionados à implementação de aplicações usando esse paradigma, tais como gerenciamento de instanciação de filtros, comunicação entre filtros, etc [De Souza Ramos et al. 2011, Beynon et al. 2001, Agbaria and Friedman 1999]. Em [De Souza Ramos et al. 2011] é apresentado o *Watershed*, um ambiente filtro-fluxo que fornece uma API C++ para a implementação de filtros e toda a troca de dados deve ser descrita usando arquivos XML (*eXtensible Markup Language*). Proposta similar é apresentada em [Beynon et al. 2001], o *Datacuter*. Ambos são ambientes completos de execução em que os programadores carregam os arquivos de configuração e os filtros implementados e toda a execução é realizada de forma transparente. Por fim, em [Agbaria and Friedman 1999], é proposto um sistema que permite salvar o estado das aplicações em ambientes distribuídos, explorando configurações dinâmicas em um *cluster*. Em nosso trabalho não propomos um ambiente completo como os descrito acima, mas uma abstração genérica para troca direta de dados entre GPUs em cenários filtro-fluxo.

Apesar da paralelização de aplicações em GPU se apresentar como um promissora área de pesquisa [Rocha et al. 2015, Melo et al. 2016], não encontramos na literatura frameworks de execução filtro-fluxo que permitem uso de múltiplas GPUs. Esse fato pode estar relacionado às limitações das tecnologias de compartilhamento de dados entre GPUS existentes: (i) necessidade de implementações baixo nível de comunicação; e (ii) necessidade de complexos controles de instanciação de filtros. Essas tecnologias podem ser divididas em dois conjuntos: (1) estratégias *intra-node*; que são exclusivamente focadas no compartilhamento de dados entre GPUs presentes em um mesmo computador [Cabezas et al. 2015, Tang et al. 2012, Young et al. 2013]; e (2) estratégias *inter-node*, focada em GPUs presentes em diferentes computadores [Shainer et al. 2011, Ammendola et al. 2013, Wu et al. 2016] .

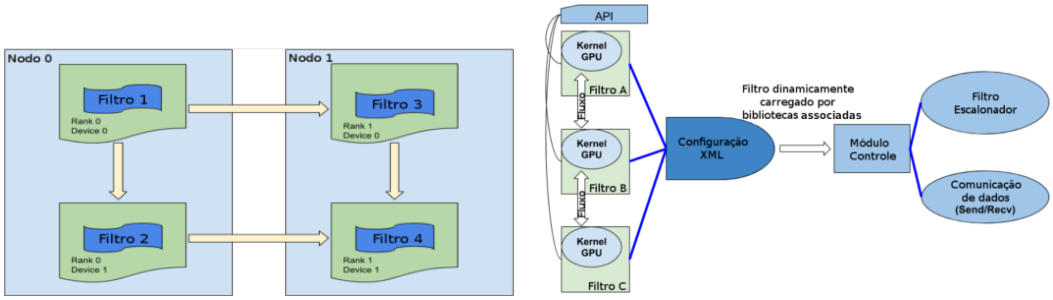
No que diz respeito às estratégias *intra-node*, uma tecnologia comum utilizada é a *Unified Virtual Adressing (UVA)*, que permite que diferentes unidades de processamento (i.e. CPU e GPU) acessem o mesmo espaço de endereço de memória, reduzindo o total de cópias de memória. Em [Tang et al. 2012] é apresentada uma otimização na qual os dados que estão em diferentes espaços de endereço são encapsulados em um único objeto. Em [Young et al. 2013], os autores apresentam um novo sistema para realocar memória em clusters baseado em um conceito de único espaço de endereço. *GPUDirect RDMA [Shainer et al. 2011]* é uma implementação desse conceito que permite a troca de dados entre GPU diretamente pelo barramento *PCI Express*. Em [Cabezas et al. 2015] os autores apresentam um estudo detalhado do desempenho desta tecnologia. As estratégias *inter-node* [Ammendola et al. 2013, Wu et al. 2016] são baseadas em uma combinação da tecnologia *GPUDirect RDMA* e implementações do padrão MPI [Gabriel et al. 2004], (i.e. *Cuda-aware MPI*). Nessas abordagens, em vez de utilizar o barramento *PCI Express* para troca de dados entre GPUs, a interface de rede é utilizada e o processo de comunicação entre diferentes computadores é realizado pelo *OpenMPI*. Conforme veremos mais adiante, algumas dessas estratégias foram avaliadas em nosso trabalho no intuito de definir aquelas a serem utilizadas por nosso framework.

3. Framework

Nesta seção detalhamos nossa proposta de uma abstração para troca direta de dados entre GPUs em cenários filtro-fluxo consolidada no *framework G-PETo*.

3.1. Visão geral

Na figura 1(a) apresentamos a aplicação que será utilizada como um exemplo base para discussão nesta seção. Esta aplicação contém quatro filtros. O primeiro filtro carrega os dados e realiza o processamento inicial. Depois, os dados são enviados para os filtros 2 e 3, onde cada filtro realiza um processamento independente nesse mesmo conjunto de dados de entrada. Neste ponto, estes filtros podem ser executados em paralelo, uma vez que o processamento de dados é independente. Além disso, a troca de dados realizada pelos filtros 1 e 2 é local, enquanto que a troca de dados entre os filtros 1 e 3 depende de transferências na rede. Ao final do processamento dos filtros 2 e 3, cada um envia seus dados para o último filtro, que realiza a operação final. Este último filtro depende dos dados de saída de mais de um filtro (2 e 3) para realizar seu processamento.



(a) Exemplo de Aplicação Filtro-Fluxo (b) Arquitetura do *G-PETo*

Figura 1. Visão Geral do G-PETo

O *G-PETo* é separado entre duas partes principais. Primeiro, temos uma API que fornece uma biblioteca de funções pré-definidas para prover a comunicação entre filtros, bem como um arquivo de configuração XML para definir os ambientes de execução (i.e. os filtros, o diretório onde os filtros estão armazenados, os endereços IP das máquinas disponíveis e o número de GPUs disponíveis em cada uma dessas máquinas). Toda essa configuração é usada pelo mecanismo de funcionamento (que será referenciado como *engine*) do *G-PETo*, que é responsável por compilar, alocar e executar os filtros da aplicação. Depois da definição do código dos filtros (funções independentes contendo um ou mais *kernels* de GPU) e da configuração do ambiente, o *engine* compila cada filtro como uma biblioteca dinâmica, e então executa a aplicação em um ambiente específico. O *engine* do framework é responsável por cuidar do escalonamento de tarefas e da comunicação. Na figura 1(b), ilustramos a arquitetura do *G-PETo*.

3.2. API do *G-PETo*

3.2.1. Definindo os Filtros

Para usar o *G-PETo*, o programador precisa implementar cada filtro em C++, salvando cada um deles em arquivos separados. O processamento dos dados é independente: cada filtro recebe seus dados, realiza o processamento necessário e envia o resultado para o próximo filtro. A dependência entre cada filtro, se necessária, é criada pelas tarefas de recebimento e envio dos dados, que são realizadas pelas funções fornecidas pela

API do framework. Na Listagem 1 apresentamos as assinaturas das funções para envio (*sendMessage*) e recebimento (*recvMessage*) de dados. A assinatura dessas funções é similar, recebendo como parâmetros o nome de um filtro de origem, o nome de um filtro de destino, o tipo de dado a ser enviado, o dado a ser transmitido (*buffer*) e seu tamanho. O processo de transmissão deve ser feito em pares: a mensagem enviada por um filtro precisa ser recebida por outro. Baseado nessa informação, bem como na configuração de ambiente previamente feita, o *engine* reconhece onde os filtros e dados estão localizados e realiza as transferências quando necessário.

```
// Filter sending data
sendMessage(char* src, char* dst, MPL_DATATYPE, void* buffer, size_t size);
// Filter receiving data
receiveMessage(char* src, char* dst, MPL_DATATYPE, void* buffer, size_t size);
```

Listagem 1. Funções de envio e recebimento de mensagens com a API proposta.

A Listagem 2 fornece um exemplo de um filtro implementado utilizando a API. A função *myFilter* é responsável por receber os dados de entrada. A função então invoca o *kernel* da GPU para o processamento dos dados. No final, os dados resultantes são enviados para o próximo filtro. A etapa de comunicação é opcional.

```
#include "comm/comm.h"
__global__ void kernel(){
    //Kernel code to be executed on GPU
}
extern "C" void myFilter(){
    //Receive the input data from the previous filter (if exists)
    recvMessage();
    //Process the data
    kernel<<<>>>();
    sendMessage();
}
```

Listagem 2. Filtro implementado usando a API. Note que os envios e recebimentos são opcionais, dependendo da fluxo de dados.

3.2.2. Processo de Comunicação

A principal contribuição do *G-PETo* é a independência da localização espacial dos dados. As funções fornecidas pela API reconhecem, automaticamente se os dados estão na GPU ou na CPU, como também se estão no mesmo nodo ou não, e então realiza a transferência dos dados. Além disso, elas atuam como barreiras entre as execuções dos filtros, cujas entradas são saídas de filtros anteriores. No exemplo de base, enquanto a transferência de dados entre os filtros 1 e 2 pode ocorrer usando uma tecnologia *intra-node* (ou seja, *GPUDirect RDMA*), a comunicação entre os filtros 1 e 3 precisa ser realizada usando uma estratégia por meio da rede (ou seja, *Cuda-aware MPI*).

Para realizar a comunicação entre GPUs no mesmo nodo, existem, basicamente, duas estratégias. A primeira delas é utilizando os recursos disponíveis na API do *CUDA*, realizando as cópias de dados da primeira GPU para a memória principal da CPU e, em seguida, realizando a cópia dos dados para o endereço de memória da segunda GPU, aqui denominada de *Standard Communication through CPU Memory*. A segunda delas é baseada na tecnologia *GPUDirect RDMA*, introduzida no CUDA 5.0, que permite acesso direto entre a GPU e um terceiro dispositivo, por exemplo, outra GPU, interface de rede ou dispositivos de armazenamento, por meio do barramento *PCI-Express*.atuais.= Placas de vídeo com CUDA 5.0 já não são hardware tão recente assim.

A comunicação inter-node, por sua vez, pode ser realizada de duas maneiras. Na primeira, aqui denominada de *MPI Standard communication*, é feita uma cópia dos dados

da primeira GPU para a memória principal da CPU para, em seguida, enviar os mesmos para a segunda máquina, utilizando MPI. Esses dados são copiados para a memória principal da máquina destino para, finalmente, serem copiados na memória da segunda GPU. A segunda é utilizar a tecnologia *CUDA-aware MPI*, em que o MPI pode reconhecer as GPUs e realizar as cópias de forma eficiente utilizando o *GPUDirect RDMA*. Dessa forma, é possível ativar a cópia de dados entre GPUs de diferentes máquinas sem passar pelo controlador da memória principal. Os dados deixam uma GPU diretamente para a interface de rede, são enviados para a máquina destino e, em seguida, copiados diretamente para a GPU de destino. Essa estratégia também pode ser usada para a transferência de dados em um mesmo nodo, desde que o *OpenMPI* (implementação MPI usada neste trabalho) possa também utilizar o barramento de dados em vez de utilizar a interface de rede.

3.2.3. Configuração de Ambiente

Finalmente, a configuração do ambiente de execução que será usada pela aplicação também precisa ser passada para o *engine*. Para isso, usamos um arquivo XML contendo a informação necessária pelo *engine*. A Listagem 3 mostra um exemplo de configuração de ambiente. Essa configuração, apesar de simples, precisa conter informação sobre cada filtro, o diretório de instalação do *G-PETo* e as máquinas a serem usadas.

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <modules>
        <file name="myFilter.c"/>
        <func name="myFilter"/>
        <includes>
            <inc>/usr/include</inc>
        </includes>
        <libraries>
            <libPath>/usr/lib</libPath>
            <lib>m</lib>
        </libraries>
    </modules>
    <home>
        <directory dir="/opt/framework/" />
    </home>
    <machine>
        <ipmachine ip="127.0.0.1" ngpus="4" />
    </machine>
</config>
```

Listagem 3. Arquivo de configuração

A configuração de filtro está dentro do campo <modules>. O sub-campo <file> define o arquivo que contém o código fonte do filtro. O sub-campo <func> define o nome da função que implementa o filtro. Os subcampos <include> e <libraries> indicam pastas e bibliotecas para serem incluídas para a compilação correta do filtro como uma biblioteca dinâmica. O campo <home> indica a pasta onde está o framework. O campo <machine> provê a informação sobre o ambiente de execução. O sub-campo <ipmachine> contém o IP de cada máquina a ser usada, assim como o número de GPUs disponíveis na mesma. Baseado neste arquivo XML de configuração, o *engine* pode realizar as tarefas necessárias para execução correta da aplicação.

3.3. Engine do Framework

O *engine* é responsável por instanciar os filtros nas máquinas disponíveis, além de realizar toda a comunicação prevista na aplicação. Recebe como entrada os códigos fontes dos filtros e o arquivo XML de configuração. A primeira etapa é compilar os filtros como bibliotecas dinâmicas, permitindo que cada filtro seja carregado dinamicamente

durante a execução. O próximo passo é configurar o processo de execução utilizando os parâmetros do arquivo de configuração. Para as duas formas de comunicação (*intra-node* e *inter-node*), adotamos o *CUDA-aware MPI*. Dessa forma, o número de filtros definido no arquivo será usado para criar os processos MPI com diferentes *MPI ranks*. Nesse momento, cada filtro recebe um identificador, que será usado para o gerenciamento da comunicação. Em seguida, cada uma das GPUs disponíveis nas máquinas também recebe uma identificação. Com essa informação, cada processo é associado com uma única GPU, usando a função *setCudaDevice* presente na biblioteca do CUDA.

Neste estágio, começa a execução da aplicação, com o *engine* controlando a comunicação. Como cada filtro possui um nome especificado no arquivo de configuração, além de um *rank* único associado ao seu processo MPI, a tradução das funções da API em chamadas MPI é feita diretamente. Cada filtro é executado isoladamente em um processo MPI, com esse fluxo de execução modelado por meio de barreiras entre cópia de dados, definido pelas chamadas da API do *G-PETo*. A figura 1(a) exemplifica o mapeamento dos níveis dos processos e suas respectivas GPU. O filtro 1 está sendo executado no computador 0, possui *rank* 0, e o ID desta GPU também é 0. Os filtros executam no mesmo tempo em que a aplicação começa e ficam em espera enquanto os dados não estão disponíveis.

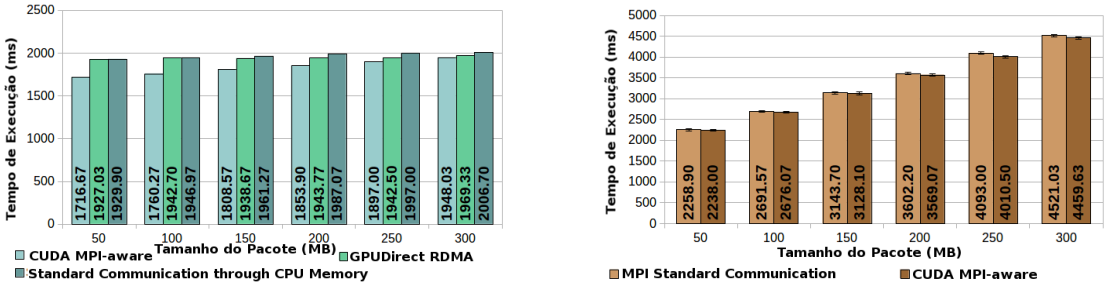
4. Avaliação Experimental e Discussões

Nessa seção apresentamos uma avaliação experimental do *G-PETo*. Primeiramente avaliamos as estratégias de comunicação descritas na seção 3.2.2. O objetivo dessa avaliação é indicar qual a estratégia mais adequada para compor o framework. Em seguida, utilizando duas aplicações reais, uma relacionada a simulações matemáticas e outra de processamento de imagens, avaliamos o impacto da camada de abstração no desempenho da mesma. Todos os experimentos apresentados nesta seção foram realizados usando duas máquinas GNU/Linux 4.1.13: (1) processador Intel(R) Core(TM) i7-6700 3,40GHz, 32GB de memória e duas GPUs NVIDIA Geforce GTX 1070; (2) processador Intel(R) Core(TM) i5-3370 3,40GHz, 16GB de memória e uma GPU NVIDIA GeForce GT 640.

4.1. Avaliação das Estratégias de Comunicação

Neste primeiro conjunto de experimentos analisamos o impacto das estratégias de comunicação no desempenho de nosso framework. Mais especificamente, para a comunicação intra-node avaliamos a **Standard Communication through CPU Memory**, a tecnologia **GPUDirect RDMA** e **CUDA-aware MPI technology**. Para a comunicação *inter-node* avaliamos a **MPI Standard Communication** e a **Cuda-aware MPI technology**. Realizamos experimentos analisando o *overhead* causado pela transferência de dados entre as GPUs responsáveis pela execução dos filtros. Realizamos a transferência de pacotes de dados entre dispositivos variando o tamanho dos pacotes entre 50MB e 300MB, utilizando as estratégias acima mencionadas. Realizamos 30 execuções para cada um dos testes propostos, a fim de extrair a média e o desvio padrão para uma melhor análise dos resultados, os quais são apresentados nos gráficos da figura 2.

Analisando primeiro os testes intra-node, apresentado na figura 2(a), temos que a estratégia *CUDA-aware MPI* mostrou-se a mais eficiente, principalmente para dados menores, apresentando uma redução de até $11,0\%$ no tempo de execução comparado com a estratégia *GPUDirect RDMA*, que foi a segunda estratégia de melhor desempenho. Em um primeiro momento, esse resultado poderia ser considerado uma surpresa, haja



(a) Estratégias *Intra-Node*

(b) Estratégias *Inter-Node*

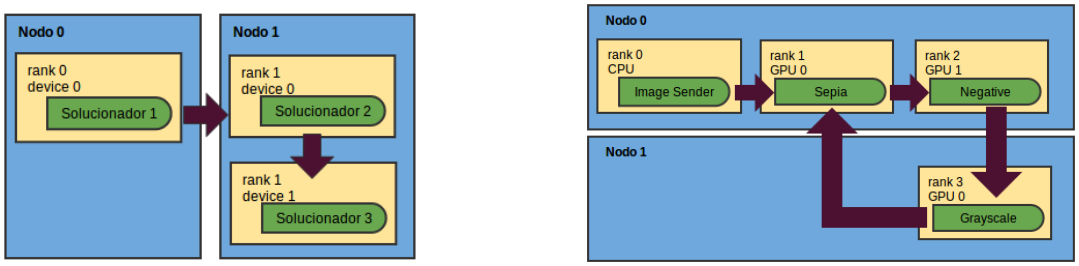
Figura 2. Avaliação das estratégias de comunicação. Estratégia CUDA-aware MPI apresenta melhores resultados em ambos cenários.

vista que o *CUDA-aware MPI* é baseado no *GPUDirect RDMA*. Entretanto, observando a extensa documentação da biblioteca OpenMPI (https://www.open-mpi.org/), é mencionado que a mesma utiliza como suporte o *CUDA IPC*, possibilitando mover dados entre GPUs que estejam no mesmo nodo e mesma raiz PCI, usando algumas otimizações no empacotamento dos dados. À medida que aumentamos o total de dados a ser transferido, o tempo gasto na comunicação utilizando o *CUDA-Aware MPI* também cresce, uma vez que o total de chamadas de funções MPI também aumenta. Entretanto, é importante mencionar que apesar disso, o desempenho do *CUDA-aware MPI* continua melhor do que outras estratégias. Analisando os testes das estratégias inter-node, apresentados na figura 2(b), observamos resultados similares aos alcançados pelas estratégias intra-node, mas com adição do *overhead* da comunicação de rede. Como esperado, a implementação usando *CUDA-aware MPI* apresentou o melhor tempo de execução. Comparado o tempo dessa estratégia com a *MPI Standard Communication*, observamos uma redução de 2% no tempo total. Assim, utilizamos no *G-PETo* o *CUDA-aware MPI* como nossa estratégia de comunicação, independentemente do local da GPU (intra-node ou inter-node).

4.2. Avaliação do Impacto do Desempenho da Aplicação

Nesta seção nosso objetivo é avaliar o impacto da camada de abstração do *G-PETo* no desempenho de aplicações reais. Para isso, consideramos duas aplicações, uma relacionada à álgebra linear e outra a processamento de imagens. A primeira consiste em um filtro que gera uma matriz esparsa tridiagonal A e um vetor b e envia esse dados para outro filtro que resolve o sistema linear representado pela matriz ($Ax = b$) usando o método do gradiente conjugado. O filtro de resolução, então envia o resultado (x) para outro filtro que resolverá o sistema novamente utilizando o resultado recebido como o lado esquerdo (b) de um novo sistema. A configuração de filtros utilizada em nossa avaliação está apresentada na figura 3 (a). A segunda aplicação real implementada lida com fluxo de imagens que são entradas para diferentes filtros executando uma determinada transformação na imagem (*sépia, negative e grayscale*). Resumidamente, para cada filtro (transformação) executado, gera-se uma nova imagem (fluxo) que será entrada para outra transformação. Os filtros são implementados por kernels CUDA com os respectivos algoritmos das transformações. A figura 3 (b) sumariza a configuração de filtros adotada em nossos experimentos. Variamos o total de imagens consideradas, bem como o tamanho das mesmas, realizando sobre cada imagem 200 transformações.

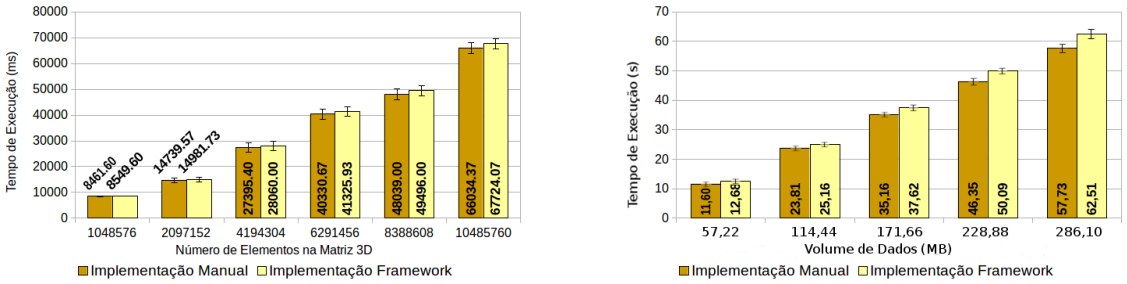
Consideramos duas implementações para cada uma das aplicações: (1) uma



(a) Álgebra Linear (b) Processamento de Imagens

Figura 3. Configuração das Aplicações Implementadas.

completamente implementada utilizando o *G-PETo* (**Implementação Framework**), sem qualquer preocupação com a instanciação dos filtros e comunicação entre eles; e (2) uma **Implementação Manual**, utilizando as funções relacionadas ao *CUDA-aware MPI*, alocando os filtros e realizando a comunicação manualmente. O objetivo não é demonstrar que o desempenho do nosso framework é melhor que a implementação manual, uma vez que seu desempenho é limitado pelas tecnologias adotadas. Como podemos observar na figura 4, o tempo de execução das implementações manuais e utilizando o *G-PETo* são praticamente equivalentes. Apesar do *overhead* relacionado à camada de abstração criada para facilitar o processo de programação e ao controle automático da instanciação dos filtros, o desempenho geral da aplicação não foi comprometida.



(a) Aplicação de Álgebra Linear (b) Aplicação de Processamento de Imagens

Figura 4. Impacto da Camada de Abstração na Performance da Aplicação. Como podemos ver, o G-PETo não comprometeu a performance geral das aplicações.

5. Conclusão e Trabalhos Futuros

Neste trabalho propomos, implementamos e avaliamos uma abstração genérica para a troca direta e transparente de dados entre GPUs em cenários filtro-fluxo em *clusters* heterogêneos. Essa abstração permite que todos os detalhes de implementações de baixo nível, relacionados à comunicação com a GPU e o controle relacionado à localização dos filtros sejam feitos de forma transparente aos programadores. Depois de avaliar várias estratégias de comunicação possíveis, mostramos que nossa proposta, além de facilitar a programação em si e controlar automaticamente a instanciação dos filtros, não compromete o desempenho geral da aplicação. Como trabalhos futuros, pretendemos melhorar o framework fazendo com que ele verifique se as GPUs disponíveis possuem suporte para utilizar o *GPUDirect RDMA*, escolhendo a melhor opção disponível para cada hardware. Além disso, nossa meta é acoplar nosso framework em um ambiente filtro-fluxo completo, tal como o Watershed [De Souza Ramos et al. 2011].

Referências

Acharya, A., Uysal, M., and Saltz, J. (1998). Active disks: Programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):81–91.

Agbaria, A. M. and Friedman, R. (1999). Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *Proceedings of HPDC*, pages 167–176. IEEE.

Ammendola, R., Bernaschi, M., Biagioni, A., Bisson, M., Fatica, M., Frezza, O., Lo Cicero, F., Lonardo, A., Mastrostefano, E., Paolucci, P. S., et al. (2013). Gpu peer-to-peer techniques applied to a cluster interconnect. In *IPDPSW*.

Beynon, M. D., Kurc, T., Catalyurek, U., Chang, C., Sussman, A., and Saltz, J. (2001). Distributed processing of very large datasets with datacutter. *Parallel Computing*.

Cabezas, J., Jordà, M., Gelado, I., Navarro, N., and Hwu, W.-m. (2015). Gpu-sm: shared memory multi-gpu programming. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, pages 13–24. ACM.

De Souza Ramos, T. L. A., Oliveira, R. S., De Carvalho, A. P., Ferreira, R. A. C., and Meira Jr, W. (2011). Watershed: A high performance distributed stream processing system. In *SBAC-PAD*.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., and Lumsdaine (2004). Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*.

Melo, D., Toledo, S., ao, F. M., Sachetto, R., Andrade, G., Ferreira, R., Parthasarathy, S., and Rocha, L. (2016). Hierarchical density-based clustering based on gpu accelerated data indexing strategy. Proc. of ICCS.

Rocha, L., Ramos, G., Chaves, R., Sachetto, R., Madeira, D., Viegas, F., Andrade, G., Daniel, S., Gonçalves, M., and Ferreira, R. (2015). G-knn: an efficient document classification algorithm for sparse datasets on gpus using knn. In *Proc. of ACM SAC*.

Shainer, G., Ayoub, A., Lui, P., Liu, T., Kagan, M., Trott, C. R., Scantlen, G., and Crozier, P. S. (2011). The development of mellanox/nvidia gpudirect over infiniband - a new model for gpu to gpu communications. *Computer Science-Research and Development*.

Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73.

Tang, K., Yu, Y., Wang, Y., Zhou, Y., and Guo, H. (2012). Ema: Turning multiple address spaces transparent to cuda programming. In *7th IEEE ChinaGrid*.

Wienke, S., Springer, P., Terboven, C., and an Mey, D. (2012). Openacc: First experiences with real-world applications. In *Proc. of Euro-Par'12*.

Wu, W., Bosilca, G., vandeVaart, R., Jeaugey, S., and Dongarra, J. (2016). Gpu-aware non-contiguous data movement in open mpi. In *Proc. of the ACM HPDC*.

Young, J., Shon, S. H., Yalamanchili, S., Merritt, A., Schwan, K., and Fröning, H. (2013). Oncilla: a gas runtime for efficient resource allocation and data movement in accelerated clusters. In *IEEE CLUSTER*.