Etino: Colocação Automática de Computação em Hardware Heterogêneo

Douglas do Couto Teixeira, Kézia Andrade, Gleison Souza, Fernando Pereira

UFMG – Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte
{douglas,kezia.andrade,gleison.mendonca,fernando}@dcc.ufmg.br

Resumo As placas de processamento gráfico (GPUs) revolucionaram a programação de alto desempenho, pois elas reduziram o custo do hardware paralelo. A programação desses dispositivos, contudo, é ainda um desafio, pois programadores ainda não são treinados para escrever código que coordene a atuação simultânea de milhares de threads. A fim de lidar com esse problema, a indústria e a academia vem introduzindo sistemas de anotações, como OpenMP 4.0, OpenSs e OpenACC, que permitem indicar quais partes de um programa C ou Fortran deveriam executar em GPU ou em CPU. Contudo, a tarefa de semear tais anotações no código ainda cabe ao programador. Este artigo apresenta uma solução para tal problema. Nós projetamos, implementamos e testamos Etino, uma ferramenta que distribui anotações OpenACC em programas de forma semi-automática. Etino usa informação de perfilamento para decidir quais partes de um programa devem executar na GPU. Ao enviar funções de alta complexidade computacional para a GPU e manter funções de baixa complexidade na CPU, Etino é capaz de melhorar substancialmente o tempo de execução de programas. Nós desenvolvemos um conjunto de benchmarks tipicamente encontrados em sistemas de alto desempenho, e usando o compilador LLVM, em conjunto com o perfilador aprof, construímos um sistema que aponta quais partes paralelas de um programa devem ser enviadas para a GPU. Esse mecanismo permite-nos melhorar em até 50x o tempo de execução de alguns programas.

1 Introdução

Unidades de Processamento Gráfico (GPUs) têm se tornado cada vez mais populares no mundo de Computação de Alta Performance. Elas provêem uma plataforma simples, barata e eficiente para a execução de programas paralelos [18]. Desde o lançamento de CUDA, no início de 2006 [5], uma enorme gama de padrões de programação e algoritmos têm sido projetados para executar nesse ambiente. Isso se justifica pela grande versatilidade das GPUs, que podem ser usadas para resolver problemas tão distintos quanto sequenciamento genético [20], roteamento IP [16], álgebra linear [23] e análise de programas [19].

Entretanto, escrever programas para GPUs ainda é uma tarefa difícil, visto que apesar de ser possível programá-los em linguagens como C e C++, o programador ainda precisa pensar em detalhes do hardware subjacente. Recentemente, tanto a indústria quanto a academia têm somado forças para facilitar a

programação de GPUs. Um dos esforços nesse sentido é o desenvolvimento de sistemas de anotação de programas. Sistemas desse tipo, tais como OpenMP, OpenACC e OpenSs, usam diretivas para sinalizar trechos de código que podem ser executados em paralelo. Essa abordagem possui duas vantagens. Primeiro, ela mantém programadores em sua zona de conforto: eles podem continuar desenvolvendo código em sua linguagem de programação preferida. Segundo, as anotações protegem programadores de minúcias do hardware paralelo, uma vez que elas passam a tarefa de paralelizar programas para o gerador de código.

Apesar de a inserção de diretivas no código esconder detalhes de hardware, ela não resolve todos os problemas do programador: ele ainda precisa identificar quando será vantajoso executar um dado trecho de código na GPU e quando não. A GPU possui uma grande quantidade de unidades lógicas e aritméticas; logo, ela pode realizar uma quantidade grande de trabalho simultâneo. Porém, para que a GPU possa ser utilizada, os dados a serem processados precisam ser transferidos para sua memória. Essa transferência de dados entre CPU e GPU é uma operação muito cara, e inerentemente linear no tamanho dos dados a serem transferidos. Dessa forma, a GPU passa a ser vantajosa quando o custo do trabalho a ser realizado sobre os dados é maior que o custo de mover esses dados entre dispositivos diferentes. Determinar as situações em que a GPU deve ser utilizada é uma tarefa desafiadora, dada a quantidade de fatores envolvidos nesse cálculo. É tal desafio que esse artigo procura resolver.

Na tentativa de retirar do programador o peso de decidir quais trechos de código devem ser movidos para a GPU, nós desenvolvemos uma técnica para automatizar essa decisão. A idéia chave do presente trabalho é utilizar informação de perfilamento (*profiling*) para decidir em que parte do hardware heterogêneo cada computação deve ser executada. Se a complexidade computacional de um trecho de código é super-linear, nós o enviamos para a GPU, doutro modo, o executamos na CPU. Indicamos tal decisão para o gerador de código via diretivas OpenACC [21]. OpenACC é um sistema de anotações projetado exatamente para permitir que programadores possam adaptar um programa C, escrito de forma sequencial, de forma que ele use os vários elementos de uma arquitetura heterogênea. Etino exige somente uma intervenção de usuários: eles precisam indicar quais laços possuem iterações independentes via a anotação OpenACC `#pragma acc loop independent`.

A fim de validar nossas idéias, nós as materializamos em uma sistema construído a partir da união de três ferramentas: aprof [4], LLVM [13] e ipmacc [12]. A primeira dessas ferramentas, aprof, é um perfilador capaz de coletar informações como o tamanho da memória lida e o custo de execução de programas. Especulamos que funções de complexidade super-linear são boas candidatas para serem enviadas à GPU. LLVM é uma infra-estrutura de compilação que possui um *front-end* para C. Nós utilizamos esse compilador para inserir diretivas OpenACC sobre o programa a ser paralelizado. Finalmente, ipmacc é um compilador fonte-a-fonte que traduz o código C aumentado com as diretivas OpenACC para código escrito em C para CUDA. A ferramenta que esse artigo descreve, a partir deste ponto, será chamada *Etino*. Esse nome advém do fato da molécula de etino

possuir composição química C^2H^2, a mesma sigla que adotamos para *Colocador de Computação em Hardware Heterogêneo*.

Nossa abordagem, até o presente momento, é totalmente estática: decidimos onde cada função deve executar durante a compilação do programa, e tal decisão não muda durante a sua execução. Ainda assim, nossos resultados são animadores. Nós implementamos um conjunto de 8 algoritmos que admitem versões paralelas, tais como multiplicação matricial, decomposição de Cholesky, o algoritmo dos K vizinhos mais próximos e casamento de *strings*. Os experimentos da Seção 4 mostram que nossa ferramenta foi capaz de indicar corretamente os trechos de código que devem ser executados na GPU, de forma similar ao que seria feito por um programador experiente na área. Como consequência dessa precisão, os programas que produzimos são até 50x mais rápidos que os programas originais.

2 Visão Geral

Anotações para Hardware Heterogêneo. Muitos sistemas de computação são, atualmente, formados por *hardware heterogêneo*. Um hardware heterogêneo é constituído por diferentes tipos de processadores. Uma configuração comum é termos, em um mesmo sistema, uma CPU e uma GPU. Existem várias maneiras de programar aplicações para esse tipo de hardware. Há, por exemplo, linguagens de programação específicas para essa tarefa, como CUDA [5] e OpenCL [22]. Além disso, há também *sistemas de anotações* que podemos usar para decidir em qual processador cada parte de uma aplicação deve ser executada. Exemplos de sistema de anotações que permitem a colocação de computações em GPUs incluem OpenMP 4.0 [11], OpenSs [15] e OpenACC [21]. Neste artigo usaremos diretivas OpenACC como o mecanismo básico para decidir onde executar cada laço de um programa paralelo.

OpenACC[21] é um conjunto de diretivas de compilador, rotinas de biblioteca e variáveis de ambiente que permitem que programas sejam transferidos do processador (CPU) para um coprocessador ou acelerador (uma GPU, por exemplo). As diretivas OpenACC são pequenas anotações no código que indicam trechos que podem ser executados em um coprocessador mas elas não fazem parte do código-fonte em si. Em outras palavras, o código-fonte em OpenACC pode ser compilado tanto por um compilador OpenACC quanto por um compilador convencional.

Os três principais tipos de diretivas OpenACC são: *kernel*, *data*, e *loop*. As diretivas *kernel* são usadas para indicar que uma porção de código pode ser convertida em um kernel para ser executado no coprocessador. As diretivas *data* definem regiões do programa nas quais dados são acessíveis pelo acelerador, além de permitirem ao programador especificar quais dados devem ser copiados para o acelerador. E as diretivas *loop* são aplicadas a um laço e aos laços a ele aninhados para indicar que tipo de paralelismo o coprocessador deverá usar para executar as iterações deste laço. A Figura 1 ilustra o uso dessas diretivas.

```
void mat-sum-GPU(float **a, float **b, float **c, int s) {
  int i, j;
  # pragma acc data pcopyin(a[0:s],b[0:s]) pcopy(c[0:s]) {
  # pragma acc kernels pcopyin(a,b) pcopy(c) {
  # pragma acc loop independent {
    for (i = 0; i < s; ++i)
      for (j = 0; j < s; ++j)
        c[i * s + j] = a[i * s + j] + b[i * s + j];
  } // end pragma loop
  } // end pragma kernels
  } // end pragma data
}
```

(a)

```
void mat-mul-GPU(float **a, float **b, float **c, int s) {
  int i, j, k;
  float sum = 0.0;
  # pragma acc data pcopyin(a[0:s],b[0:s]) pcopy(c[0:s]) {
  # pragma acc kernels pcopyin(a,b) pcopy(c) {
  # pragma acc loop independent {
    for (i = 0; i < s; ++i)
      for (j = 0; j < s; ++j) {
        sum = 0.0;
        for (k = 0; k < s; ++k)
          sum = sum + a[i * s + k] * b[k * s + j];
        c[i * s + j] = sum;
      }
  } // end pragma loop
  } // end pragma kernels
  } // end pragma data
}
```

(b)

Figura 1: Cálculo de soma e multiplicação de matrizes em OpenACC.

```
__gloabl__ void foo(...) {...}
```

```
int main(void) {
  int N = 1000000;
  float *x, *dx;
  // Allocate memory on the device
  cudaMalloc(&d_x, N* sizeof(float));
  // Copy data from the host to the device:
  cudaMemcpy(d_x, x, N * sizeof(float), cudaMemcpyHostToDevice);
  // Call foo on the device:
  foo<<<...>>>(...);
  // Copy data back to the host:
  cudaMemcpy(x, d_x, N * sizeof(float), cudaMemcpyDeviceToHost);
}
```

Figura 2: Código necessário para mover um vetor para a GPU.

A Figura 1 mostra dois trechos de código em OpenACC: na Figura 1(a) temos o cálculo da soma de duas matrizes, e na Figura 1(b) o cálculo da multiplicação de duas matrizes. Esses trechos de código mostram diretivas OpenACC indicando quais laços podem ser paralelizados. Mas a decisão de mover ou não esses laços para a GPU é mais complicada, pois deve levar em consideração o tempo gasto para mover tais dados da CPU para a GPU.

Tradeoff: executar o código na CPU ou movê-lo para a GPU? Para que seja possível executar uma aplicação na GPU, é necessário que os dados que essa aplicação vai processar sejam enviados para a memória da placa gráfica. Essa é uma operação cara, pois a GPU e a CPU não compartilham memória. A Figura 2 mostra a quantidade de código necessária para mover um vetor da CPU para a GPU e depois de volta para a CPU.

Pela Figura 2, podemos perceber que o custo computacional de mover os dados para a GPU é linear no tamanho do dado a ser transferido, pois o procedimento `cudaMemcpy` possui este custo computacional. Assim, uma possível

regra a ser seguida é só mover código para a GPU quando a complexidade total (tanto de transferência de dados quanto de computação na GPU) for inferior à complexidade total de execução na CPU. Caso contrário o custo de transferir os dados para a GPU pode fazer com que o custo total do hardware gráfico fique maior que o custo na CPU.

A Figura 3 exibe o tempo de execução de dois algoritmos diferentes na CPU e na GPU. A figura sugere que, embora saibamos que ambos os métodos podem ser paralelizados, precisamos examinar com cuidado os custos totais desses métodos na CPU e na GPU antes de decidirmos onde eles serão executados. Na soma de matrizes na GPU, o custo total será o custo de transferência dos dados da CPU para a GPU ($O(N^2)$) somado ao custo de se realizar esses cálculos na GPU. Em um modelo PRAM [7] (*Parallel Random Access Memory*) pode-se executar a soma matricial em $O(1)$. Naturalmente, essa complexidade não é possível em uma GPU, porem, por simplicidade, nós a assumiremos, dado o elevado grau de paralelismo desses dispositivos [1]. Assim, o custo total da soma de matrizes na GPU ($O(N^2) + O(1)$) não é inferior ao custo de se realizar essa operação na CPU ($O(N^2)$).

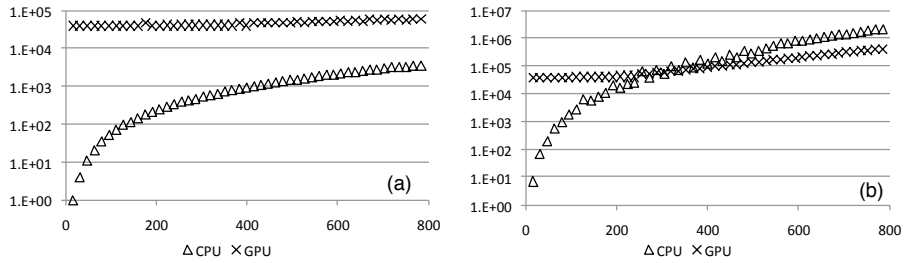

Figura 3: (Esquerda) Tempo de execução de soma de matrizes. (Direita) Tempo de execução de multiplicação matricial. Eixo X mostra número de linhas de matriz quadrada. eixo Y mostra tempo de execução, em milissegundos. CPU: Intel Xeon CPU E5-2620 2.00GHz. GPU: GeForce GTX 670 2GB.

Na multiplicação de matrizes o cenário é outro. Nesse caso, o custo computacional inclui o preço de transferência dos dados da CPU para a GPU ($O(N^2)$), somado ao preço de se realizar esses cálculos na GPU. Em uma máquina PRAM pode-se executar a multiplicação matricial em $O(\ln N)$. Uma implementação mais simples atinge $O(N)$; essa é a complexidade que assumimos para a GPU. Assim, o custo total da multiplicação de matrizes na GPU ($O(N^2) + O(N)$) é inferior ao custo de se realizar essa operação na CPU ($O(N^3)$). Baseado nisso

[1] Note que essa é uma aproximação grosseira: uma GPU não é uma máquina PRAM: ela possui uma quantidade limitada de processadores. Porém, para fins práticos, essa suposição serve aos nossos propósitos.

deveríamos mover a multiplicação de matrizes para a GPU e deixar a soma de matrizes na CPU.

3 Colocação Automática de Computação em Hardware Heterogêneo

Conforme mencionado na Seção 1, o objetivo deste trabalho é determinar, com um mínimo de intervenção do usuário, em qual processador cada parte de uma aplicação paralela deve ser executada. Com tal propósito, foi construída Etino, uma ferramenta publicamente disponível, que está representada na Figura 4. Etino agrupa, em um mesmo pacote, três ferramentas bem conhecidas, a saber: o compilador LLVM, o perfilador *aprof* e o tradutor de código ipmacc. Além de ligar tais aplicações, a construção de Etino demandou a implementação de uma análise de complexidade assintótica, e um anotador de código. A iteração entre esses vários produtos de *software* será explicada no restante da presente seção.

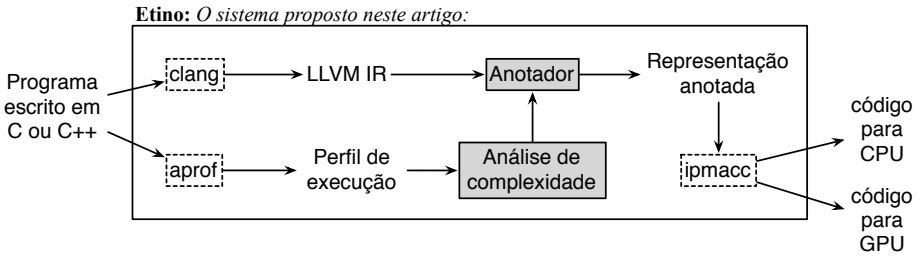

Figura 4: Visão geral do sistema de compilação proposto neste trabalho.

Perfilamento sensível à entrada. A fim de determinar o comportamento assintótico de uma aplicação, Etino produz um perfil de sua execução. Tal perfil é criado por meio da ferramenta *aprof*. Aprof retorna, para cada função de um programa, um gráfico que relata o tamanho da entrada lida com o número de instruções necessário para processar aquela entrada. O tamanho da entrada de um programa é definido como a quantidade de posições de memória que ele lê sem antes escrevê-las. Tal métrica é chamada RMS, sigla de *Read Memory Size* [4].

Uma vez que aprof usa a memória lida como medida de tamanho de entrada, essa ferramenta é robusta o suficiente para determinar a complexidade de funções que usam estruturas de dados esparsas, tais como listas encadeadas e grafos. Essa abordagem, na opinião dos autores desse artigo, é superior às alternativas puramente estáticas de inferência de complexidade [6,9,10] que foram recentemente introduzidas. A principal desvantagem das abordagens puramente

estáticas é o seu curto alcance: elas dependem de entradas de dados regulares e produzem resultados muito conservadores para programas cujo fluxo de controle pode atravessar diversos caminhos. Em face desse tipo de desafio, elas assumem que o comportamento do programa é sempre o pior caso, ainda que tal situação ocorra raramente. Além disso, a experiência dos autores com Loopus, uma ferramenta baseada nas técnicas de Gulwani *et al.* [9,10], revela que análises de complexidade puramente estáticas tendem a retornar resultados para uma quantidade relativamente pequena de laços em uma aplicação.

Análise de Complexidade e Anotação de Código. Uma vez produzido um perfil de execução para um programa, Etino passa à fase de análise de complexidade. Nesta fase analisam-se pares $(I \times T)$, onde I é o tamanho da entrada, em RMS, e T é o tempo de execução, medido como o número operações executadas. Enfatiza-se que ambas grandezas são discretas, isto é, um mesmo programa produz sempre o mesmo par $(I \times T)$, para a mesma entrada. A partir de um conjunto de pares, Etino procura encontrar uma curva que melhor se adeque àqueles pontos. Neste estágio de seu desenvolvimento, Etino não utiliza qualquer heurística elaborada: funções cuja complexidade seja super-linear são marcadas para execução na CPU. Considera-se como super-linear qualquer função cujo coeficiente de regressão linear (R) seja inferior a 0.9, e cuja reta de integração possua inclinação superior a 1.0.

A decisão de onde enviar cada computação é feita via anotações OpenACC. Etino utiliza uma transformação de código, implementada como um passo de transformação do LLVM, para inserir tais anotações. Somente laços paralelos são considerados nesta fase. Um laço paralelo é marcado com a diretiva `#pragma acc loop independent`. O usuário de Etino deve indicar quais laços são paralelos: a ferramenta ainda não consegue resolver dependências entre iterações de laços. Cada laço paralelo tem sua complexidade analisada, conforme descrito anteriormente, e aqueles considerados promissores são anotados com a diretiva `#pragma acc kernels` automaticamente. Etino usa informações de depuração para encontrar os cabeçalhos de laços onde inserir anotações. Com isso, ao analisarmos a representação intermediária de um programa, conseguimos associar uma pragma à linha de código fonte C que desejamos. Nosso passe gera uma lista com essas pragmas e as linhas nas quais elas devem ser inseridas. Posteriormente, o LLVM adiciona essas informações à sua representação intermediária quando o programa é compilado com a extensão `-g`.

Geração de Código. A geração de código é feita via ipmacc, um compilador de diretivas OpenACC desenvolvido na Universidade de Vitória, Canadá[2]. Ipmacc é um compilador fonte-a-fonte, que, conforme pode ser visto na Figura 4, produz dois tipos de código: funções escritas em C e funções escritas em C para CUDA. O mecanismo de geração de código adotado por ipmacc é simples. Laços que são marcados com a diretiva `kernel` são transformados em funções CUDA. Essa

[2] Nesse trabalho foi usado a versão de ipmacc de 15 de Março de 2015, disponível neste endereço: https://github.com/lashgar/ipmacc

transformação atribui uma *thread* para cada iteração daquele laço. Assim, é vital que não existam dependências entre iterações diferentes, ou o programa resultante dessa transformação pode ficar semanticamente errado. A Figura 5 ilustra essa transformação. O compilador ipmacc é capaz de traduzir o código visto na parte (a) da figura para o código visto na parte (b).

```
void saxpy(int n, float alpha, float *x, float *y) {
 #pragma acc data copyin(x[0:l]) copy(y[0:l]) {
  #pragma acc kernels {
   #pragma acc loop independent {
    for (int i = 0; i < n; i++) {
     y[i] = alpha*x[i] + y[i];
    }
   }
  }
 }
}
```

(a)

```
__global__ void
saxpy(int n, float alpha, float *x, float *y) {
 int i = blockIdx.x * blockDim.x + threadIdx.x;
 if (i < n) {
  y[i] = alpha * x[i] + y[i];
 }
}
```

(b)

Figura 5: (a) Programa escrito com diretivas OpenACC. (b) Programa que ipmacc produz para a entrada vista na parte (a) desta figura.

4 Resultados Experimentais

A fim de validar as idéias discutidas neste artigo, esta seção apresenta resultados experimentais produzidos com Etino. O hardware utilizado nesses experimentos é descrito a seguir:

CPU: processador Intel Xeon CPU E5-2620 com ciclo de 2.00GHz e 16 GB de memória RAM (DDR2). O sistema operacional usado é Linux Ubuntu 12.04 3.2.0;

GPU: placa gráfica GeForce GTX 670, com 2 GB de memória RAM.

OpenACC, sendo um sistema de anotações relativamente novo, ainda não possui um conjunto definitivo de benchmarks. Por isso, foi criada uma coleção de programas para testar as técnicas propostas neste trabalho. Este conjunto de benchmarks consiste de oito aplicações. Todas essas aplicações foram escritas em C e cada aplicação foi executada 20 vezes. Os laços independentes de cada programa foram marcados com a diretiva `#pragma loop independent`. Etino não precisa de qualquer outra indicação, além dessa, para adicionar pragmas sobre um programa, indicando que algumas de suas partes deveriam ser usadas na GPU. Os benchmarks criados são descritos a seguir:

- **mat-mul**: realiza a multiplicação de duas matrizes quadradas de lado n. Complexidade: $O(n^3)$.
- **mat-sum**: realiza a adição de duas matrizes quadradas de lado n. Complexidade: $O(n^2)$.

– **cholesky**: realiza a decomposição de uma matriz quadrada de lado n utilizando o algoritimo de cholesky. Complexidade: $O(n^3)$.
– **string-matching**: realiza a contagem do número de ocorrências de uma sequência de c caracteres em uma *string* de tamanho s. Complexidade: $O(c \times s)$.
– **KNN**: implementa a busca por K vizinhos mais próximos. Dado um vetor V com v pivots e um vetor P com p pontos, KNN encontra, para cada elemento de V os K pontos mais próximos em P. Esse benchmark usa um algoritmo quadrático para ordenar pontos. Complexidade: $O(v \times p \times K^2)$.
– **colinear-list**: conta o número de triplas de pontos colineares em um universo de n pontos. Esse algoritmo é implementado com uma tabela *hash*. Par de pontos é usado para produzir uma reta, que é armazenada na tabela. Colisões são verificadas em busca de triplas colineares. Complexidade: $O(n^2)$.
– **linear-search**: busca por um ponto p em um universo de n pontos. Complexidade: $O(n)$.
– **vector-prod**: realiza o produto vetorial de dois vetores de n dimensões. Complexidade: $O(n)$.

Perfilamento e Análise de Complexidade A Figura 6 mostra *o perfil de execução* de cada um dos oito benchmarks utilizados neste artigo. O perfil de execução de um programa é uma curva que relaciona o tamanho de sua entrada ao tempo necessário para processar aquela entrada. Entradas são medidas em RMS e tempo é medido em número de operações executadas. Junto a cada perfil é mostrado a curva que melhor descreve aquela complexidade, e o coeficiente de correlação (R). Quanto mais próximo de 1.0, mais apropriada é a curva encontrada via regressão polinomial. Três benchmarks, a saber: `mat-sum`, `linear-search` and `vector-prod` possuem um claro comportamento linear. Vale a pena notar que esse é o comportamento de cada programa que *aprof* reporta. A adição de matrizes implica a leitura de $O(n^2)$ dados, e realiza $O(n^2)$ operações: comportamento linear.

Tempo de Execução. A principal meta deste trabalho é permitir que programas aproveitem melhor os recursos disponíveis em hardware heterogêneo. O melhor aproveitamento de recursos, nesse contexto, traduz-se em menor tempo de execução de aplicações. Esta seção compara o tempo de execução de programas na CPU e na GPU. Os programas executados na GPU foram produzidos automaticamente por ipmacc, dadas as pragmas OpenACC que inserimos nos programas. O critério adotado para inserção de pragmas foi o seguinte: Etino foi configurado para marcar como `kernel` todo laço independente, cujo comportamento é superlinear, de acordo com o relatório produzido por *aprof*. Em geral, Etino foi capaz de identificar corretamente os casos em que o código executado no hardware gráfico apresenta ganho de performance em relação ao código executado na CPU. Contudo, houve casos interessantes, cuja análise faremos a seguir.

A Figura 7 mostra os tempos de execução (tempo de computação efetivamente realizada, não foi medido o tempo gasto com entrada e saída) desses benchmarks na CPU e na GPU, dado o critério acima descrito. Cada benchmark

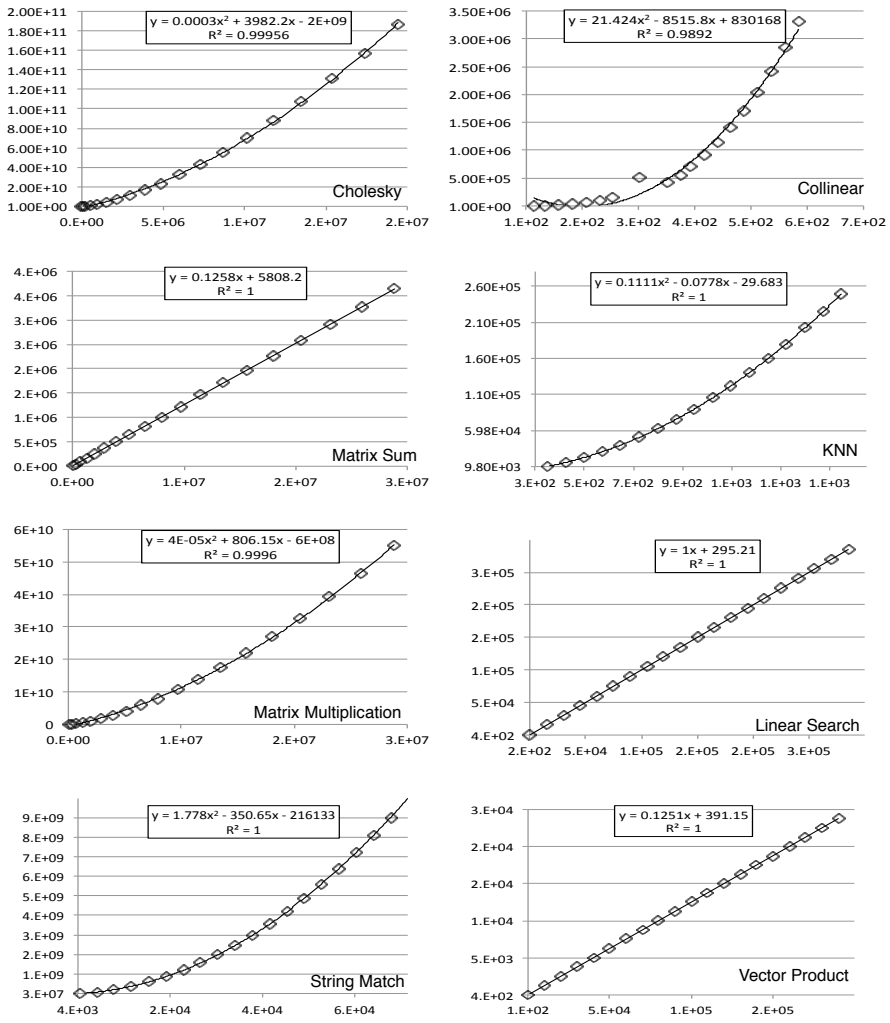

Figura 6: Análise de complexidade de cada um dos nossos benchmarks (RMSx-Custo).

foi executado 20 vezes e os valores mostrados nos gráficos são a média simples dos tempos de execução obtidos. A primeira conclusão imediata que obtém-se dos experimentos aqui descritos é que não é vantajoso, em termos de desempenho computacional, mover código linear ou sublinear para a GPU. Em outras palavras, nossos três benchmarks linears não se beneficiam do alto poder computacional da GPU. O custo de mover dados da CPU para o dispositivo gráfico torna irrelevante o ganho em termos de paralelismo. Essa constatação não quer

dizer que tais programas jamais deveriam ser enviados para a GPU. Na opinião dos autores deste trabalho, há duas situações em que tal execução é providencial: (i) quando os dados de entrada do algoritmo já estão na GPU, devido a computações prévias e (ii) quando é possível enviar os dados para a GPU em paralelo com processamento útil na CPU. Esse padrão, conhecido como *pipeline de cópia*, é bastante comum entre desenvolvedores CUDA ou OpenCL [2].

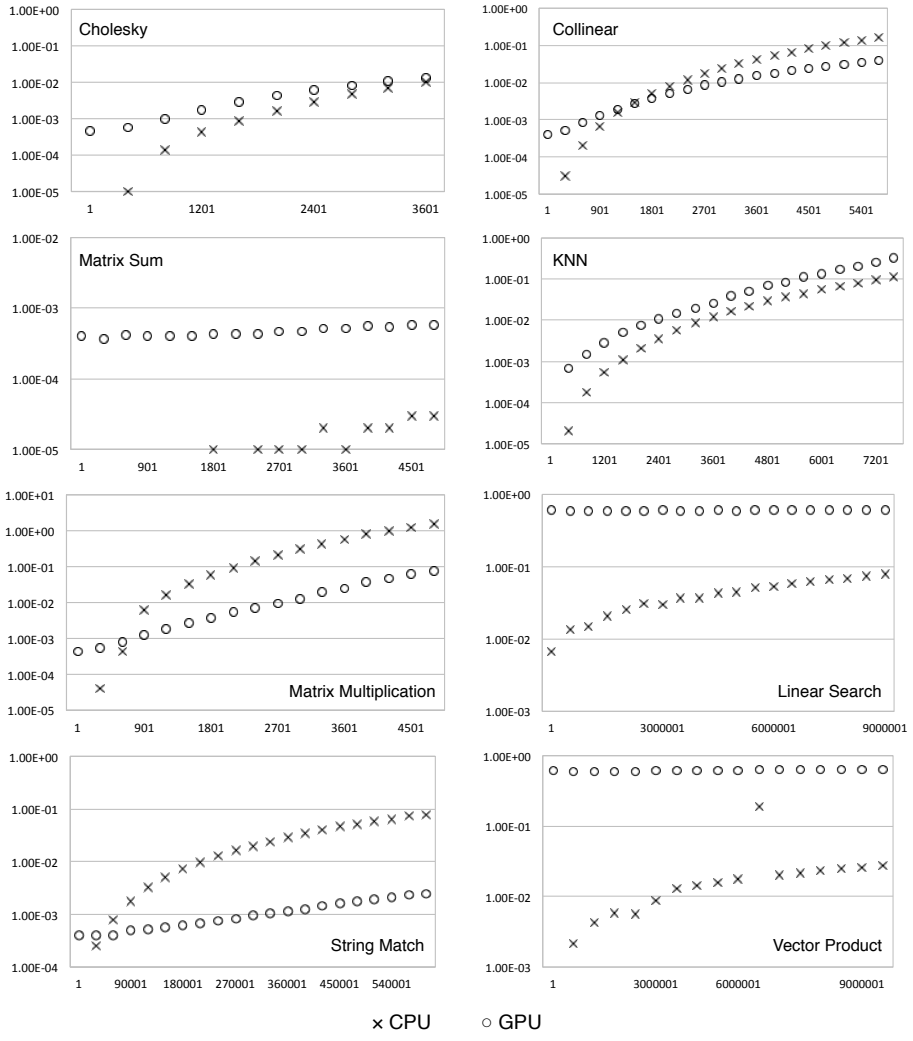

Figura 7: Comparação entre o tempo de execução de cada benchmark na CPU e na GPU. Eixo X mostra número de linhas de matriz quadrada. Eixo Y mostra tempo de execução, em milissegundos.

Os nossos resultados, contudo, não mostram que é sempre vantajoso mover código super-linear para a GPU. Tal vantagem, de fato, pôde ser observada nos seguintes benchmarks: `cholesky`, `collinear`, `mat-mul` e `string-match`. Por outro lado, o ganho de desempenho não foi observado em `KNN`. O principal responsável por essa falha são as *divergências*. Existem dois tipos de divergências: de controle e de dados. Divergências de controle ocorrem quando duas threads discordam em qual caminho de um desvio elas devem seguir. Divergências de dados ocorrem quando threads acessam a memória de forma irregular. O algoritmo dos K vizinhos apresenta ambos os tipos de divergências. A parte computacionalmente mais intensiva desse algoritmo é mostrada na Figura 8. O comando condicional na linha 6 apresenta uma divergência de controle, pois ele não leva todas as threads ao mesmo caminho. E os acessos a memória nas linhas 8-10 apresentam divergências de dados: cada índice `i` será transformado no identificador de uma thread diferente. Logo, cada um desses acessos, para valores grandes de `s`, acontecerão em regiões muito distantes na memória, e não poderão ser feitos de forma *agrupada*. Quando re-escrevemos esses índices, conforme visto na própria Figura 8, os ganhos de desempenho da GPU são impressionantes. Tais ganhos podem ser vistos no gráfico à direita. Em sua forma atual, Etino não é capaz de re-escrever índices de acesso à memória, tampouco é capaz de identificar código divergente. Essa última tarefa é a próxima meta dos autores deste trabalho.

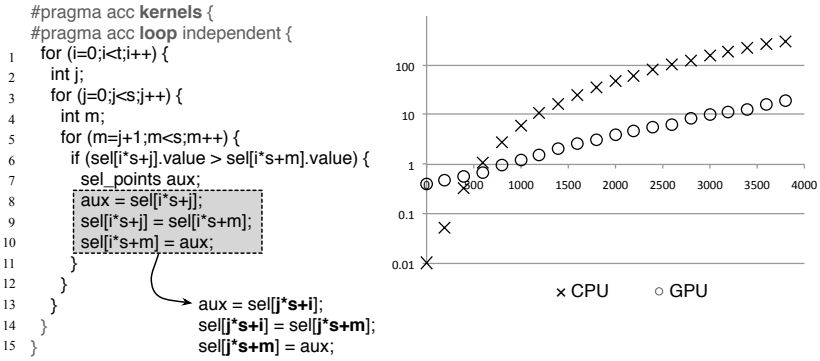

Figura 8: O núcleo do algoritmo KNN, mostrando a inversão de índices, e o correspondente ganho em desempenho. Eixo X mostra número de pontos de busca.

5 Trabalhos Relacionados

O objetivo deste trabalho é decidir, em tempo de compilação, em que parte de um hardware heterogêneo os diferentes laços de um programa devem ser executados. Não existem, até o presente momento, outros trabalhos que têm objetivos

similares. Existe, contudo, uma vasta literatura descrevendo diferentes tentativas de reimplementar algoritmos tradicionais em GPUs. Exemplos de tais algoritmos envolvem sequenciamento genético [20], ordenação [3], roteamento [16], entre outros. Ao contrário desses trabalhos, a técnica de compilação aqui implementada gera código para placas gráficas com quase nenhuma intervenção do usuário. Em outras palavras, a abordagem proposta neste artigo é uma forma semi-automática de reimplementar em C para CUDA código originalmente feito para uma CPU. A implementação semi-automática de código dá, ao desenvolvedor, uma forma rápida de prototipar idéias na placa gráfica.

Embora não se tenha conhecimento de algum trabalho que faça o que a ferramenta proposta faz, existem trabalhos que realizam tarefas semelhantes. Amni *et al.* [1], por exemplo, desenvolveram uma técnica para melhorar a cópia de dados entre CPU e GPU re-escrevendo código de forma automática. Contudo, ao contrário do presente trabalho, eles não decidem se computação deve ser executada na GPU. Tal decisão é tomada pelo desenvolvedor de programas.

Perfiladores A técnica proposta neste artigo utiliza um perfilador para identificar quais partes de um programa deveriam ser movidas para a GPU. Perfiladores tradicionais como o `gprof` [8] informam em quais métodos o programa gasta a maior parte do seu tempo de execução, mas essas ferramentas falham em mostrar como o tempo de execução dessas rotinas se comporta quando o tamanho da entrada varia. Neste trabalho foi utilizado `aprof` [4]. Esse perfilador, construído sobre *Valgrind* [17], é capaz de relacionar o tempo de execução de uma função com o tamanho dos dados que tal função lê. O objetivo deste trabalho não é implementar um perfilador novo; ao contrário, a técnica proposta utiliza uma ferramenta já existente, o *aprof*, para decidir quais partes de um programa devem executar na placa gráfica.

Sistemas de Anotação para Hardware Heterogêneo. Neste trabalho foi usado um sistema de anotações, OpenACC [21] para determinar em qual processador cada laço de um programa deve ser executado. Existem vários outros sistemas de anotação similares a OpenACC. A mais conhecida dentre tais metalinguagens é OpenMP [11], criada em 1997. OpenMP 4.0 possui, atualmente, um conjunto de diretivas, rotinas e variáveis de ambiente que dão ao compilador informação suficiente para transformarem código em C para código CUDA. Um terceiro sistema de anotações é OpenSs [15], produto de pesquisa realizada no Centro de Computação de Barcelona. Este artigo não definiu um sistema de anotações novo. OpenACC foi usado para indicar como computação deveria ser distribuída. A escolha de OpenACC foi pragmática: um dos autores deste trabalho possui familiaridade com ipmacc, um compilador que lê diretivas naquele sistema. Tanto OpenMP 4.0 quanto OpenSs poderia prestar-se aos mesmos propósitos que OpenACC para este trabalho.

Vários compiladores traduzem código anotado para Cuda. Neste trabalho foi usado ipmacc, porém alternativas similares estão hoje publicamente disponíveis. O OpenARC [14] é um compilador OpenACC aberto. Foi concebido como um framework de pesquisa e inclui transformações do compilador e otimizações para OpenACC. Há alguns compiladores comerciais que reconhecem diretivas

OpenACC. Tais compiladores são produzidos por empresas como Cray, CAPS entreprise e *The Portland Group* (PGI). Espera-se que nos anos vindouros, compiladores mais populares, como gcc e LLVM, também sejam capazes de reconhecer diretivas OpenACC em código. A técnica de compilação descrita neste trabalho não tem por objetivo traduzir código C em código CUDA. Ao contrário, neste trabalho foi usado um compilador que faz tal trabalho: ipmacc. O que a técnica proposta neste artigo faz é distribuir diretivas OpenACC em um programa. Ipmacc não faz tal distribuição: ele já espera um programa anotado.

6 Conclusão

Este artigo apresentou Etino, uma ferramenta que decide em que processadores, GPU ou CPU, cada parte paralela de um programa deve ser executada. Etino exige que seus usuários indiquem, via pragmas OpenACC, quais laços são independentes. Porém, a decisão de onde executar cada computação é automática, e baseia-se tão somente nos resultados produzidos por um perfilador de execução. Os experimentos realizados neste trabalho mostraram que, em geral, é interessante enviar código super-linear para a GPU, enquanto código linear, ou sub-linear, ainda que parelelo, não leva a ganhos de desempenho. O próximo passo no projeto de Etino é adicionar a essa ferramenta a capacidade de reconhecer código divergente. Esse fenômeno, as divergências, podem acabar com os ganhos de desempenho obtidos ao mover-se código para a GPU.

Acknowledgment

Este trabalho é patrocinado pela LG Electronics, via o projeto [CTO_Parallelization_UFMG] e pelo CNPq. O suporte de Ahmad Lashgar, autor de ipmacc, foi fundamental para a realização deste trabalho.

Referências

1. Mehdi Amini, Fabien Coelho, François Irigoin, and Ronan Keryell. Static compilation analysis for host-accelerator communication optimization. In *LCPC*, pages 237–251. Springer, 2011.
2. Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-Mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *PPoPP*, pages 105–114. ACM, 2010.
3. Daniel Cederman and Philippas Tsigas. GPU-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics*, 14(1):4–24, 2009.
4. Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *PLDI*, pages 89–98. ACM, 2012.
5. Michael Garland. Parallel computing experiences with CUDA. *IEEE Micro*, 28:13–27, 2008.

6. Thomas Martin Gawlitza and David Monniaux. Invariant generation through strategy iteration in succinctly represented control flow graphs. *Logical Methods in Computer Science*, 8(3), 2012.
7. Alan Gibbons. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.
8. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *PLDI*, pages 49–57, 1982.
9. BhargavS. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, volume 5123 of *LNCS*, pages 370–384. Springer, 2008.
10. Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.
11. Julien Jaeger, Patrick Carribault, and Marc Pérache. Fine-grain data management directory for openmp 4.0 and openacc. *Concurrency and Computation: Practice and Experience*, 27(6):1528–1539, 2015.
12. Ahmad Lashgar, Alireza Majidi, and Amirali Baniasadi. IPMACC: open source openacc to cuda/opencl translator. *CoRR*, abs/1412.1127, 2014.
13. Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.
14. Seyong Lee and Jeffrey S. Vetter. Openarc: open accelerator research compiler for directive-based, efficient heterogeneous computing. In *HPDC*, pages 115–120. ACM, 2014.
15. Cor Meenderinck and Ben H. H. Juurlink. Nexus: Hardware support for task-based programming. In *DSD*, pages 442–445. Springer, 2011.
16. Shuai Mu, Xinya Zhang, Nairen Zhang, Jiaxin Lu, Yangdong Steve Deng, and Shu Zhang. IP routing processing with graphic processors. In *DATE*, pages 93–98. IEEE, 2010.
17. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM, 2007.
18. John Nickolls and William J. Dally. The GPU computing era. *IEEE Micro*, 30:56–69, 2010.
19. Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. EigenCFA: Accelerating flow analysis with GPUs. In *POPL*. ACM, 2011.
20. Edans Flavius O. Sandes and Alba Cristina M.A. de Melo. Cudalign: using gpu to accelerate the comparison of megabase genomic sequences. In *PPoPP*, pages 137–146. ACM, 2010.
21. OpenACC Standard. The openacc programming interface. Technical report, CAPs, 2013.
22. Tristan Vanderbruggen and John Cavazos. Generating opencl C kernels from openacc. In *IWOCL*, page 9, 2014.
23. Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the gpu. In *PPoPP*, pages 127–136. ACM, 2010.