Etino: Colocação Automática de Computação em Hardware Heterogêneo

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

[1] Departamento de Ciência da Computação –
Universidade Federal de Minas Gerais (UFMG)
Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte

{douglas,kezia.andrade,gleison.mendonca,fernando}@dcc.ufmg.br

Abstract. *Graphics Processing Units (GPUs) revolutionized high performance computing. However, programming these devices is still a hard task. This paper presents* ***Etino****, a tool that aims at reducing the complexity of programming GPUs by hiding details of the underlying hardware as well as inserting OpenACC annotations automatically in programs. Etino uses profiling information to decide what parts of a program should run in the GPU. By sending functions that have high asymptotic complexity to the GPU and keeping those with low complexity in the CPU, Etino is able to improve substantially the running time of programs. Our tool is available at* `http://youtu.be/HdzsRZuPqJM`*.*

Resumo. *As placas de processamento gráfico (GPUs) revolucionaram a programação de alto desempenho, porém a programção desses dispositivos é ainda um desafio. Este artigo apresenta* ***Etino****, uma ferramenta que visa facilitar a programação desses dispositivos, escondendo detalhes de hardware e distribuindo anotações OpenACC em programas de forma 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. Nossa ferramenta está disponível em* `http://youtu.be/HdzsRZuPqJM`*.*

1. Introdução

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 vêm 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.

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. 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.

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 [Standard 2013]. 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 um sistema construído a partir da união de três ferramentas: aprof [Coppa et al. 2012], LLVM [Lattner and Adve 2004] e ipmacc [Lashgar et al. 2014]. A primeira dessas ferramentas, aprof, é um perfilador capaz de coletar informações sobre a complexidade assintótica 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. Os experimentos realizados 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 – de forma totalmente automática – são até 50x mais rápidos que os programas originais.

2. 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 1. 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.

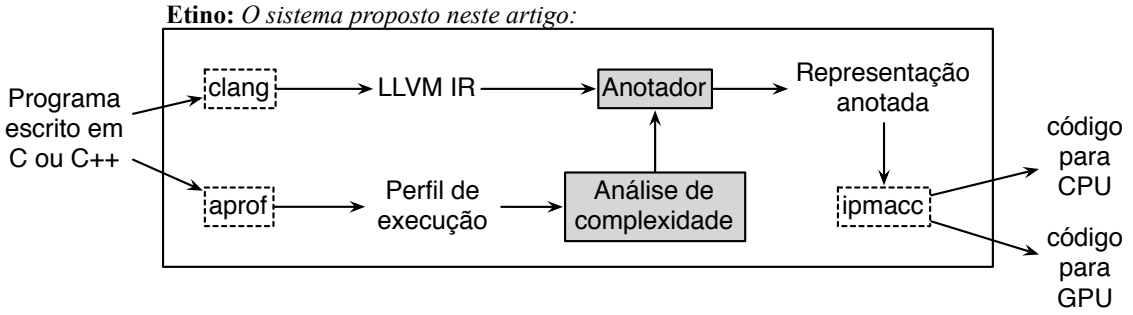

Figure 1. Visão geral do sistema de compilação proposto neste trabalho.

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* [Coppa et al. 2012].

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 [Gawlitza and Monniaux 2012, Gulavani and Gulwani 2008, Gulwani et al. 2009] 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.* [Gulavani and Gulwani 2008, Gulwani et al. 2009], 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 GPU. 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 para onde enviar cada computação é feita via anotações OpenACC. Etino utiliza uma transformação de código, implementada como um passe 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`. Etino usa informações de depuração para encontrar os cabeçalhos de laços onde inserir anotações. LLVM adiciona essas informações à sua representação intermediária.

Geração de Código. A geração de código é feita via ipmacc, um compilador de diretivas OpenACC desenvolvido na Universidade de Vitória, Canadá[1]. Ipmacc é um compilador fonte-a-fonte, que, conforme pode ser visto na Figura 1, 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 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 2 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)

Figure 2. (a) Programa escrito com diretivas OpenACC. (b) Programa que ipmacc produz para a entrada vista na parte (a) desta figura.

Execução Completa da Ferramenta. Etino é uma ferramenta que distribui anotações OpenACC em programas de forma automática. Para o uso da ferramenta é necessário executar os seguintes comandos:

1. Geração do binário do programa sequencial: `gcc -O3 matmul.c -o matmul -lm`

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

2. Perfilamento: `./inst/bin/valgrind --tool=aprof -single-log=yes matmul`
3. Coleta das principais informações do perfilameto: `./AprofOutput matMulRelatorio.aprof`
4. Geração do código para GPU: `opt -load /path-to-llvm/LLVMInsertPragmas.so -insert-pragmas matmul.ll -stats -debug`
5. Compilação do código para GPU gerado: `ipmacc matmul.c -o matmul`

3. Estudo de Caso

A fim de validar as idéias discutidas neste artigo, esta seção apresenta dois estudos de caso e seus respectivos resultados 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.

Nesta seção iremos apresentar duas aplicações, escritas em C que foram utilizadas para validar a ferramenta. 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. As aplicações utilizadas são citadas 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: $O(n^2)$.

Perfilamento e Análise de Complexidade O primeiro passo para execução da ferramenta é fazer o perfilamento. A Figura 3 mostra *o perfil de execução* dos dois algoritmos utilizados como caso de uso 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.

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 [Gibbons 1988] (*Parallel Random Access Memory*) pode-se executar a soma matricial em $O(1)$. Assumimos que essa complexidade é possível em uma GPU, dado o elevado grau de paralelismo desses dispositivos [2]. 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 ($O(N^2)$).

[2]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.

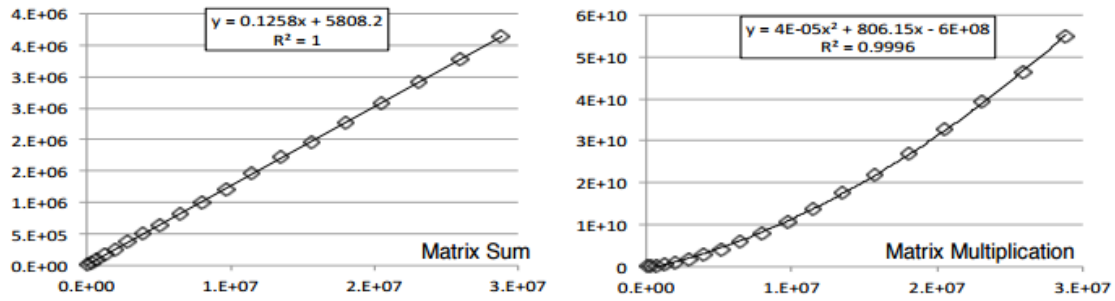

Figure 3. Análise de complexidade dos benchmarks (Eixo X: RMS; Eixo Y: Custo).

Na multiplicação de matrizes o cenário é outro. Nesse caso, o custo computacional inclui o preço de transferência dos dados da CPU 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 deveríamos mover a multiplicação de matrizes para a GPU e deixar a soma de matrizes na CPU.

Tempo de Execução. A principal meta da ferramenta 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.

A Figura 4 mostra os tempos de execução das aplicações citadas na CPU e na GPU, dado o critério acima descrito. 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. 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 [Baghsorkhi et al. 2010].

4. Ferramentas Relacionados

O objetivo da ferramenta Etino 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, outras ferramentas que têm objetivos similares.

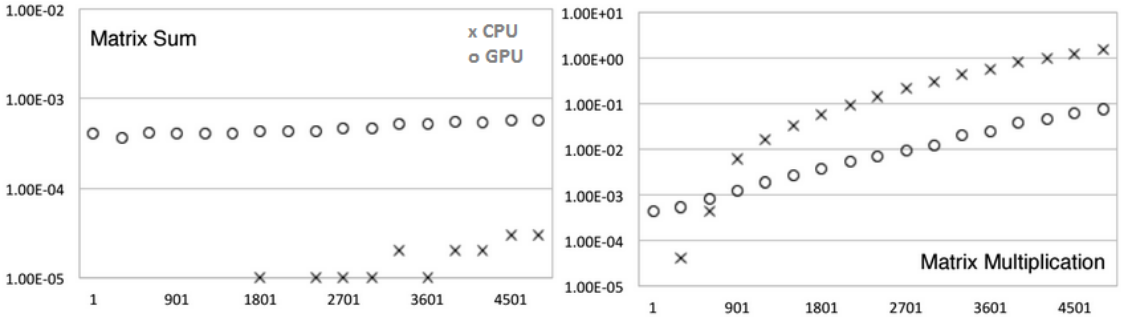

Figure 4. Comparação entre o tempo de execução dos algoritmos na CPU e na GPU (Eixo X: Tamanho de uma matriz quadrada; Eixo Y: Tempo de execução do algoritmo).

Existe, contudo, uma vasta literatura descrevendo diferentes tentativas de reimplementar algoritmos tradicionais em GPUs. Exemplos de tais algoritmos envolvem sequenciamento genético [Sandes and de Melo 2010], ordenação [Cederman and Tsigas 2009], roteamento [Mu et al. 2010], entre outros. Ao contrário desses trabalhos, a técnica de compilação aqui implementada gera código para placas gráficas – sem – a intervenção do usuário. Em outras palavras, a abordagem proposta neste artigo é uma forma automática de reimplementar em C para CUDA código originalmente feito para uma CPU. A implementação 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.* [Amini et al. 2011], 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.

5. 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 [Coutinho et al. 2011]. Esse fenômeno, as divergências, podem acabar com os ganhos de desempenho obtidos ao mover-se código para a GPU.

References

[Amini et al. 2011] Amini, M., Coelho, F., Irigoin, F., and Keryell, R. (2011). Static compilation analysis for host-accelerator communication optimization. In *LCPC*, pages 237–251. Springer.

[Baghsorkhi et al. 2010] Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., and Hwu, W.-M. W. (2010). An adaptive performance modeling tool for gpu architectures. In *PPoPP*, pages 105–114. ACM.

[Cederman and Tsigas 2009] Cederman, D. and Tsigas, P. (2009). GPU-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics*, 14(1):4–24.

[Coppa et al. 2012] Coppa, E., Demetrescu, C., and Finocchi, I. (2012). Input-sensitive profiling. In *PLDI*, pages 89–98. ACM.

[Coutinho et al. 2011] Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Jr., W. M. (2011). Divergence analysis and optimizations. In *PACT*, pages 320–329. IEEE.

[Gawlitza and Monniaux 2012] Gawlitza, T. M. and Monniaux, D. (2012). Invariant generation through strategy iteration in succinctly represented control flow graphs. *Logical Methods in Computer Science*, 8(3).

[Gibbons 1988] Gibbons, A. (1988). *Efficient Parallel Algorithms*. Cambridge University Press.

[Gulavani and Gulwani 2008] Gulavani, B. and Gulwani, S. (2008). 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.

[Gulwani et al. 2009] Gulwani, S., Mehra, K. K., and Chilimbi, T. (2009). Speed: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM.

[Lashgar et al. 2014] Lashgar, A., Majidi, A., and Baniasadi, A. (2014). IPMACC: open source openacc to cuda/opencl translator. *CoRR*, abs/1412.1127.

[Lattner and Adve 2004] Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

[Mu et al. 2010] Mu, S., Zhang, X., Zhang, N., Lu, J., Deng, Y. S., and Zhang, S. (2010). IP routing processing with graphic processors. In *DATE*, pages 93–98. IEEE.

[Sandes and de Melo 2010] Sandes, E. F. O. and de Melo, A. C. M. (2010). Cudalign: using gpu to accelerate the comparison of megabase genomic sequences. In *PPoPP*, pages 137–146. ACM.

[Standard 2013] Standard, O. (2013). The openacc programming interface. Technical report, CAPs.