Aplicando Model-Driven Development à Plataforma GPGPU

Ademir Carvalho Jr., Thiago S. M. C. Farias, João Marcelo X. N. Teixeira,
Veronica Teichrieb, Judith Kelner
Centro de Informática, Universidade Federal de Pernambuco
{ajcj,tsmcf, jmxnt, vt, jk}@cin.ufpe.br

Resumo

GPUs (Graphics Processing Units) são dispositivos gráficos que vêm ganhando destaque nos últimos anos pela sua eficiência em processamento paralelo. Neste contexto, o termo GPGPU (General-Purpose computation on GPU) é um novo conceito que visa explorar as vantagens das GPUs em áreas não necessariamente relacionadas a processamento gráfico. Este trabalho aplica princípios do MDD (Model-Driven Development) ao desenvolvimento de aplicações para GPU, visando produzir um ambiente mais adequado para a construção desse tipo de software. O resultado do trabalho foi o desenvolvimento de uma ferramenta que enxerga uma aplicação como um modelo e gera automaticamente parte significativa do código desta aplicação. O código gerado é expresso na linguagem definida por CUDA (Compute Unified Device Architecture), uma plataforma de programação para GPGPU.

1. Introdução

O campo da Engenharia de Software estuda e desenvolve técnicas e ferramentas que podem ser aplicadas a diversos domínios relacionados ao processo de desenvolvimento de software. A principal motivação deste campo é o aumento de produtividade que os seus recursos são capazes de oferecer.

Neste trabalho, o MDD (*Model-Driven Development*) [1], uma abordagem existente na Engenharia de Software, foi aplicada ao desenvolvimento de aplicações de alto desempenho, no contexto de GPGPU (*General Purpose computation on GPU*) [2]. O objetivo é produzir um ambiente mais adequado para a construção de softwares deste tipo.

O uso do MDD envolve a definição de um meta-modelo, um documento que descreve o domínio em que se está trabalhando. Assim, um meta-modelo para descrever o contexto de GPGPU foi elaborado neste trabalho. Estes meta-modelos costumam ser utilizados em ferramentas existentes, que auxiliam os desenvolvedores na geração do código final de uma aplicação. Neste trabalho, como estudo de caso, utilizamos o GMF (*Graphical Modeling Framework*) [3], para construir uma ferramenta similar. A ferramenta construída aceita modelos como entrada do usuário e, como saída, gera automaticamente parte significativa do código da aplicação. A linguagem escolhida para a geração de código foi a linguagem definida por CUDA (*Compute Unified Device Architecture*) [4], uma plataforma de programação para GPGPU.

Para fins de validação deste trabalho, uma comparação foi realizada. Foi avaliado o desenvolvimento de um software, tanto através dos meios existentes atualmente, como através da ferramenta construída.

O restante do trabalho está estruturado da seguinte forma: a Seção 2 introduz os conceitos e o estado da arte em relação aos temas abordados, a Seção 3 aborda alguns trabalhos relacionados, a Seção 4 descreve o meta-modelo elaborado neste trabalho, a Seção 5 apresenta a ferramenta desenvolvida. A Seção 6 demonstra a validação da aplicação. Por fim, a Seção 7 conclui a apresentação do trabalho, destacando alguns trabalhos futuros.

2. Background

2.1 GPU

GPUs (*Graphics Processing Units*) [5] são dispositivos que vêm ganhando destaque nos últimos anos, principalmente pela sua eficiência em processamento. Esta eficiência é obtida através de sua arquitetura, que contém um alto grau de paralelismo, característica necessária para a execução de sistemas gráficos.

De acordo com [4], as recentes arquiteturas das GPUs proporcionam, além de um vasto poder computacional, uma grande velocidade no barramento de dados, sendo, em certas situações, superiores às CPUs comuns nestes aspectos.

Um importante conceito é o modo de execução de instruções de uma GPU. GPUs são processadores orientados para a execução paralela de instruções. Eles são otimizados para processar operações sobre vetores, executando no modo SIMD (*Single Instruction, Multiple Data*)[1]. Isto se reflete em sua arquitetura, de modo que ela inclui mais transistores dedicados ao processamento de dados, em detrimento da realização do *caching* de dados e fluxo de controle, quando comparado a uma CPU comum.

Vale ressaltar que a arquitetura de uma GPU é bastante relacionada ao denominado *pipeline gráfico*. Sua arquitetura foi projetada de modo a realizar as operações contidas neste *pipeline* de forma simultânea.

Recentemente, novas funcionalidades foram adicionadas às GPUs, incluindo a habilidade de modificar o processo de renderização do *pipeline* através dos chamados *shaders* [6]. Esta flexibilidade é considerada uma característica chave da arquitetura, e é o pilar básico para GPGPU.

2.2 GPGPU

GPGPU é um conceito que visa explorar as vantagens das GPUs em áreas não necessariamente relacionadas a processamento gráfico. Seu principal desafio é identificar os problemas que podem ser solucionados, e como mapear estes problemas para a plataforma, já que ela foi inicialmente desenvolvida com o foco em aplicações gráficas.

Nem todos os tipos de problemas se encaixam no contexto das GPUs, mas existem vários casos de aplicações que conseguiram um aumento significativo de desempenho através do uso de hardware gráfico. Uma boa métrica para avaliar a chance de uma aplicação tomar proveito das vantagens de uma GPU é a sua intensidade aritmética, uma métrica que mede a razão entre a quantidade de operações matemáticas e a quantidade de acessos à memória existentes em uma aplicação. Quando mapeadas para as GPUs, aplicações que possuem uma alta intensidade aritmética costumam ser bem sucedidas.

Porém, a natureza gráfica das GPUs torna árdua a programação para outros fins, exigindo do programador conhecimentos adicionais, como a adaptação da aplicação ao *pipeline gráfico*. Por exemplo, no caso de uma aplicação em GPGPU que utiliza *shaders* é necessária a renderização de alguma primitiva gráfica (por exemplo, um quadrilátero texturizado) para que o aplicativo seja inicializado, assim como a obtenção do *feedback* da aplicação, que é realizada através da leitura de texturas.

Conceitualmente, uma aplicação comum não possui relação com o desenho de gráficos e manipulação de texturas, mas as linguagens existentes forçam os programadores a pensarem desta forma. As plataformas mais recentes, como Cg (*C for Graphics*) [7], Accelerator [8], GLSL (*OpenGL Shading Language*) [9] e HLSL (*High Level Shading Language*) [10], tentam lidar com este problema abstraindo alguns detalhes da GPU para o programador. Na próxima subseção, a plataforma utilizada neste trabalho, CUDA, será melhor detalhada.

O Brook [11], uma linguagem para GPGPU, é uma extensão da linguagem ANSI C. Esta extensão contém conceitos de um modelo de programação denominado *streaming*. Um *stream* conceitualmente significa um *array*, exceto pelo fato de que todos os seus elementos podem ser operados em paralelo através de funções conhecidas como *kernels*. Esta linguagem mapeia automaticamente *kernels* e *streams* em *shaders* e memórias de texturas. O Folding@Home [12], um projeto de computação distribuída para o estudo do comportamento de proteínas, é um exemplo de aplicação que utiliza o Brook.

2.3 CUDA

A arquitetura de CUDA [4] foi lançada pela NVIDIA em Novembro de 2007, para ser utilizada em conjunto com suas placas gráficas. Os primeiros processadores compatíveis com CUDA são os da série G80. Esta é uma arquitetura que permite realizar computação de propósito geral utilizando a GPU.

O modelo de software proposto por CUDA e pela maioria das linguagens para GPGPU enxerga a GPU como um co-processador de dados paralelo. Neste contexto, a GPU é considerada como o dispositivo (ou *device*) e a CPU como anfitrião (ou *host*).

Alguns conceitos existentes nesta plataforma são *threads*, *blocks*, *grids* e *kernels*, conforme ilustrado na Figura 1. *Threads* são as unidades de execução paralela em uma GPU. Elas são agrupadas em *blocks*, onde *threads* pertencentes a um mesmo *block* podem sincronizar sua execução e compartilhar um mesmo espaço de memória (*shared memory*). Um conjunto de *blocks* representa um *grid*. E um *kernel* consiste no

[1] SIMD é uma técnica aplicada para obtenção de paralelismo em processadores. No SIMD, uma mesma instrução é executada em paralelo em diferentes partes do hardware e sobre diferentes dados.

código que é executado por uma *thread*. Cada chamada a um *kernel* precisa especificar uma configuração contendo o número de *blocks* em cada *grid*, o número de *threads* em cada *block* e opcionalmente a quantidade de memória compartilhada a ser alocada.

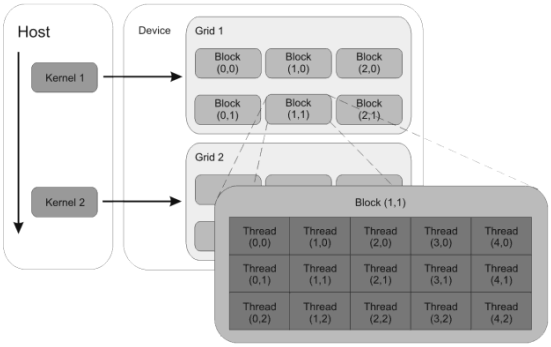

Figura 1 – Modelo de execução de CUDA.

A sintaxe da linguagem é similar à sintaxe da linguagem C, com algumas extensões. Estas extensões basicamente definem o escopo de uma função ou variável. Os escopos permitidos são `host`, `global` e `device`. O primeiro define que uma função ou variável só pode ser utilizada/executada na CPU, o segundo define que a função deve ser executada na GPU, mas que pode ser invocada pela CPU, e o último define que a função ou variável pode apenas ser executada na GPU. Extensões também foram criadas para informar que uma variável faz parte da memória compartilhada e desta forma, estará acessível apenas a *threads* de um mesmo *block*.

CUDA pode ser utilizada em conjunto com a ferramenta Microsoft Visual Studio [13], através da configuração das ferramentas de CUDA neste ambiente.

2.4 MDD

MDD [1] é uma filosofia de desenvolvimento de software que possibilita a construção do modelo de um sistema e, a partir deste, a geração do sistema real.

Esta filosofia transfere o foco de desenvolvimento das linguagens de programação de terceira geração para modelos, mais especificamente modelos descritos em UML (*Unified Modeling Language*) [14] e seus *profiles*. O objetivo dessas abordagens é facilitar o desenvolvimento de aplicações, permitindo o uso de conceitos ligados ao domínio em questão ao invés dos conceitos oferecidos pelas linguagens de programação [15]. Todavia, uma vez que esta ainda não é uma abordagem bastante difundida, existem poucas iniciativas que a utilizam. Uma destas iniciativas é o MDA (*Model-Driven Architecture*) [16], que provê um conjunto de padrões para o desenvolvimento baseado em modelos.

Um dos principais desafios do MDD é a dificuldade técnica envolvida na transformação de modelos em código. Existe ainda uma descrença por parte dos desenvolvedores sobre a geração de código eficiente, compacto, e que atenda ao propósito contido em seu modelo. Pode-se pensar nesta situação analogamente à enfrentada há alguns anos atrás, com a introdução dos compiladores. Estes têm a função de transformar objetos descritos em uma linguagem de alto nível para objetos em uma linguagem de mais baixo nível (linguagem de máquina) e aspectos de eficiência e desempenho são considerados nesta transformação [1].

2.5 MDA

MDA [16] é um padrão definido pelo OMG (*Object Management Group*)[2], um consórcio entre empresas de software, indústria, governo, e instituições acadêmicas. O MDA pode ser visto como um framework conceitual para a definição de um conjunto de padrões que suportam o MDD.

Em MDA, um modelo significa uma representação de parte de uma funcionalidade, estrutura ou comportamento de um sistema e precisa estar relacionado, sem ambigüidades, a uma definição de sintaxe e semântica.

MDA possui alguns tipos de modelos para descrição de um sistema. Os principais são o PIM (Modelo Independente de Plataforma) que fornece informações sobre a estrutura e a funcionalidade de um sistema, porém, abstraindo os detalhes técnicos, e o PSM (Modelo Específico de uma Plataforma) que funciona como uma extensão ao PIM, incluindo os detalhes específicos de uma tecnologia ou plataforma. Existe também o CIM (Modelo Independente de Computação ou Modelo do Domínio), que representa a visão de um sistema sem levar em consideração detalhes relativos à sua computação.

Como o PSM é descrito em UML, existe uma incompatibilidade na representação dos elementos da plataforma, através dos elementos existentes em UML. Por exemplo, como representar uma função da linguagem C em UML?

Esse problema pode ser solucionado com o auxílio de decisões contidas em um Perfil UML (ou *UML Profile*). Um Perfil UML consiste em um conjunto de

[2] Para mais informações sobre o OMG visite http://www.omg.org.

extensões à linguagem UML que utiliza recursos para rotular elementos e denotar que o elemento possui alguma semântica em particular. Perfis UML constituem a base para a construção de ferramentas de modelagem. Porém, esta é uma tecnologia ainda não consolidada, e faltam ferramentas apropriadas para o seu uso. O AndroMDA [17] e o OptimalJ [18] constituem algumas destas ferramentas, que são denominadas *MDA-Oriented-Tools*. Ferramentas que trabalham com modelagem UML como o IBM Rational Rose [19] ou o ArgoUML [20] também podem ser consideradas ferramentas MDA apesar de não terem sido desenvolvidas com este propósito.

3. Trabalhos Relacionados

3.1 COSA

O COSA (*Complementary Objects for Software Applications*) [21] é um ambiente de construção e execução de software, baseado em sinais, que foi projetado para aplicações concorrentes. Um dos objetivos deste projeto diz respeito à produção de aplicações livres de *bugs*. Argumenta-se que a prática tradicional de desenvolvimento de software baseada em algoritmos é não-confiável. Assim, este projeto propõe uma mudança para um modelo visual baseado em sinais, na tentativa de amenizar este problema.

O modelo proposto pelo COSA tem a visão de um computador não como uma simples máquina para execução de instruções, mas sim como uma coleção de objetos que interagem entre si. Um objeto consiste em uma entidade de software que reside na memória do computador e que aguarda um sinal para realizar uma operação e emitir um sinal de saída.

O desenvolvimento de softwares no COSA consiste em conectar objetos elementares utilizando uma ferramenta de composição gráfica. Não existem procedimentos, sub-rotinas ou ciclos de compilação e execução. No COSA não existe sequer a necessidade de aprender alguma linguagem de programação. A Figura 2 ilustra uma estrutura em *loop* desenvolvida no ambiente COSA.

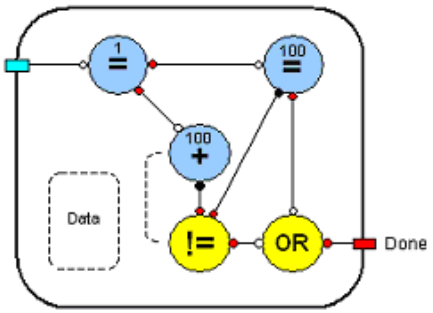

Figura 2 – Exemplo de *loop* em COSA.

3.2 Array-OL

O Array-OL [22] é uma linguagem utilizada para expressar concorrência, com foco na modelagem de aplicações para processamento de sinais. Esta linguagem é baseada em um modelo que permite expressar tanto o paralelismo de tarefas (*task-level parallelism*) como o paralelismo de dados (*data-level parallelism*) de uma aplicação. Neste modelo, tarefas são conectadas umas as outras através de dependências de dados.

A descrição de uma aplicação em Array-OL utiliza dois modelos. O modelo global (Figura 3) define um grafo de seqüência entre as diferentes partes de uma aplicação, determinando assim, o paralelismo de tarefas. Cada nó neste grafo representa uma tarefa, e cada aresta representa um *array*. Já o modelo local especifica ações elementares em cima dos dados, definindo o paralelismo de dados em cada tarefa. Este modelo local permite especificar como os *arrays* são consumidos e produzidos em uma dada tarefa. É importante ressaltar que o Array-OL é apenas uma linguagem de especificação; não existem regras para a execução de uma aplicação descrita nesta linguagem.

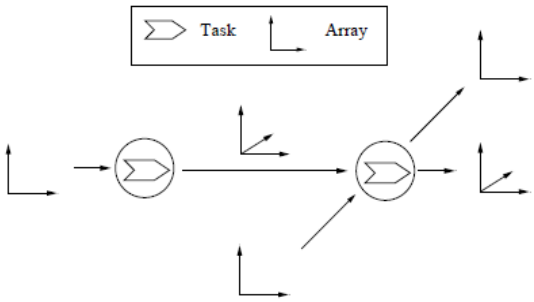

Figura 3 – Modelo Global do Array-OL.

3.3 Gaspard2

Um dos ambientes que utilizam o Array-OL é o Gaspard2 [23], um ambiente voltado para o *design* de sistemas SoC (*System-on-Chip*) orientados a modelos. Gaspard2 baseia-se em uma abordagem que a partir de dois modelos UML, um descrevendo a aplicação e o outro descrevendo o hardware, um mapeamento da aplicação no hardware é realizado, através do chamado modelo de associação. Este modelo de associação é então projetado em descrições de baixo nível como modelos de simulação ou de síntese.

Assim como o Array-OL, o Gaspard2 é um ambiente dedicado à especificação de aplicações de processamento de sinal. Ele estende o Array-OL, permitindo, além da modelagem, funcionalidades como simulação, teste, e geração de código de aplicações SoC e arquiteturas de hardware. O Gaspard2 utiliza um Perfil UML [23] para modelagem das aplicações, e é capaz de gerar código em SystemC [24].

4. Modelo do Domínio

O meta-modelo definido neste trabalho contempla os principais elementos do contexto de paralelismo: Tarefas e Dados. A simplicidade deste meta-modelo reflete a simplicidade das abstrações existentes em GPGPU. Porém, isso não significa que as aplicações que podem ser elaboradas a partir deste modelo estarão restritas. Deve-se entender que este é um meta-modelo, ou seja, é a partir dele que os modelos reais de aplicação serão instanciados. Estas instâncias podem ter dezenas de elementos diferentes combinados de diversas formas.

A Figura 4 ilustra o meta-modelo do domínio especificado para o contexto de GPGPU.

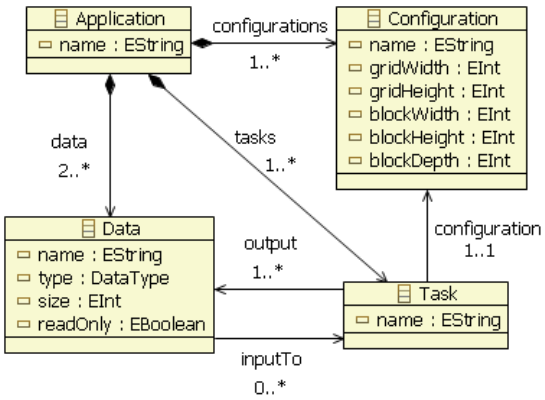

Figura 4 – Meta-Modelo em UML.

Os elementos básicos deste modelo são as tarefas, representadas pela classe `Task`, que consistem em atividades que possuem potencial para serem realizadas paralelamente, e os dados, representados pela classe `Data`, que representam as informações que são manipuladas pelas tarefas. O modelo inclui os relacionamentos, `inputTo` e `output`, entre tarefas e dados, para explicitar as entradas e saídas de cada tarefa. Dependências entre tarefas podem ser representadas através do encadeamento dos dados de saída de uma tarefa para a entrada de outra tarefa. O modelo inclui ainda, para cada elemento, informações descritivas como o nome do dado ou tarefa, e no caso do dado, o seu tipo e tamanho. O tipo de um dado pode possuir qualquer valor contido na enumeração `DataType`, que não está ilustrada no meta-modelo por questões de espaço. O conjunto de todos estes elementos forma uma aplicação, representado pela classe `Application`, que consiste no resultado final de um modelo. Há ainda o elemento `Configuration`, um conceito existente nas linguagens para GPGPU, que especifica a organização das *threads* que irão executar uma certa tarefa.

Como exemplo a Figura 5 ilustra um modelo de aplicação de GPGPU, elaborado na ferramenta desenvolvida neste trabalho, que realiza a soma e a subtração dos valores de dois *arrays*.

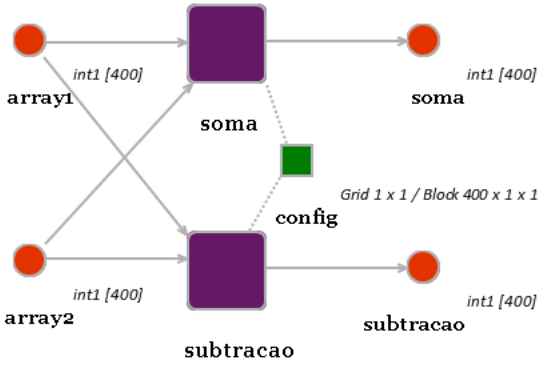

Figura 5 – Modelo de uma Aplicação.

Este modelo contém dois elementos `Task`, representando as tarefas de soma e subtração, quatro elementos `Data`, representando os dois *arrays* de entrada e os dois *arrays* produzidos como resultado, e um elemento `Configuration`, especificando a organização das *threads* para as duas tarefas. Todo esse conjunto de elementos forma um elemento `Application`.

Vale ressaltar que, apesar da ferramenta construída neste trabalho utilizar este meta-modelo para gerar aplicações usando CUDA, o modelo é genérico o

suficiente para gerar aplicações para outras linguagens de GPGPU, como OpenCL, etc.

5. Aplicação Desenvolvida

Podemos definir o CudaMDA como um software para construção de aplicações usando CUDA que, a partir de modelos, similares ao da Figura 7, é capaz de gerar parte significativa do código da aplicação resultante deste modelo.

Seguem descrições dos quatro componentes principais do software desenvolvido:

- Perfil UML de CUDA: este componente da arquitetura é necessário aos demais componentes, pois especifica os elementos disponíveis nos modelos elaborados pela aplicação.

- Interface Gráfica: o objetivo deste componente é fornecer uma interface gráfica do software para o usuário, que exponha todas as funcionalidades da ferramenta, de forma adequada.

- Componente de Manipulação de Modelos: componente que contém as classes que representam e interpretam os modelos. Este componente consegue extrair toda a informação útil de um modelo.

- Componente de Geração de Código: sua função é receber um modelo como entrada e gerar o código da aplicação usando CUDA como saída.

No desenvolvimento da ferramenta, a IDE Eclipse [25] foi utilizada. O Eclipse é um ambiente de desenvolvimento Java [26], porém, sua arquitetura fornece condições para que ele seja estendido a outras linguagens e funcionalidades. Isso é possível por ter sua arquitetura baseada no conceito de *plugins*, extensões que podem ser integradas ao ambiente. Os plugins desempenham um importante papel neste trabalho, pois alguns plugins, como o EMF e o GMF, que serão descritos a seguir, foram utilizados no desenvolvimento dos componentes, enquanto a própria ferramenta desenvolvida constitui um plugin para o Eclipse.

O EMF (*Eclipse Modeling Framework*) [27] é um *plugin* cuja funcionalidade consiste na definição e implementação de modelos de dados estruturados. O resultado final no uso do EMF é um *plugin* para o Eclipse capaz de lidar com os modelos cuja definição foi fornecida como entrada. O componente Perfil UML de CUDA foi elaborado a partir do EMF. O modelo dado como entrada foi o meta-modelo apresentado na seção anterior.

O GMF (*Graphical Modeling Framework*) [3] é um *plugin*, contendo uma extensão ao EMF, que permite produzir editores gráficos mais complexos melhorando a qualidade da representação visual de um modelo. O componente de Manipulação de Modelos, e parte do componente de Interface Gráfica, foram implementados na forma de um *plugin* gerado pelo GMF.

O JET (*Java Emitter Templates*) [28] é um recurso contido no EMF para geração de código. No JET é utilizada uma sintaxe similar à sintaxe de JSP (*Java Server Pages*) para a criação de *templates* que especificam o código a ser gerado. O componente de Geração de Código foi implementado com cinco *templates* JET e algumas classes Java. As classes Java foram escritas para extrair a informação contida em um modelo elaborado pelo usuário e repassar essas informações aos *templates* para que o código CUDA possa ser gerado. As ações realizadas pelos *templates* consistem basicamente em:

- Mapeamento de cada tarefa existente no modelo para um *kernel* da linguagem CUDA.

- Geração de uma função principal, que encapsula a execução paralela das tarefas.

Os *kernels* gerados pelo mapeamento não contêm código. A idéia é que o usuário forneça o código específico de cada tarefa no corpo dos *kernels*. Todavia, a lista de parâmetros dos *kernels* é gerada de acordo com os dados de entrada e saída existentes para a tarefa no modelo.

Já a função gerada contém código em seu interior. Esse código realiza automaticamente as chamadas aos *kernels*, o gerenciamento de memória, a transferência de dados entre GPU e CPU, e o encadeamento da saída de um *kernel* para a entrada de outro, caso estes venham a ser dependentes. Elementos `Data` especificados como `readOnly` são mapeados em memórias de textura no código.

Como o código que executa dentro da GPU é o que será escrito pelo usuário dentro dos *templates* dos *kernels*, a performance da aplicação está ligada diretamente à implementação do programador, de forma que a ferramenta não influencia no tempo de execução da mesma.

Boa parte do componente de Interface Gráfica foi implementada através do *plugin* gerado pelo GMF, porém alguns recursos precisaram ser incluídos na interface para melhorar a experiência do usuário na

utilização da ferramenta. Esta parte da interface foi implementada como um outro *plugin*. Este *plugin* contém opções para acessar as funcionalidades do CudaMDA, *wizards* para facilitar a criação de projetos e modelos dentro da ferramenta, além de customizações em alguns componentes gráficos, para diferenciá-los dos componentes comuns do Eclipse.

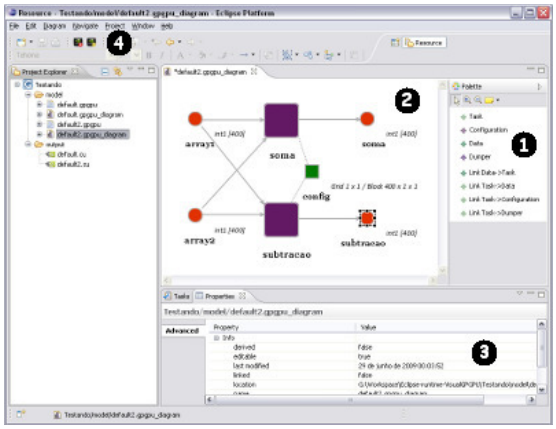

Figura 6 – Screenshot da aplicação.

A Figura 6, apesar do tamanho reduzido, apresenta a interface principal do CudaMDA. Em (1), está a paleta contendo as opções disponíveis para incluir em um modelo: dados, tarefas, configurações e conexões entre os elementos. Em (2), está a representação visual do modelo. Em (3), encontra-se a tabela de propriedades, que permite visualizar e alterar informações do elemento do modelo selecionado no momento. Em (4), estão posicionados os botões que permitem a geração do código do modelo que editado. Utilizando esses recursos o usuário é capaz de elaborar um modelo para sua aplicação e gerar o código da mesma.

6. Validação

Para validar o trabalho proposto, um teste da ferramenta foi realizado. Seis desenvolvedores com razoável conhecimento e experiência em CUDA foram selecionados, e lhes foi passada a especificação de duas aplicações, ambas de complexidade similar.

Uma destas aplicações deveria ser implementada com o auxílio da ferramenta, ou seja, o desenvolvedor iria elaborar o modelo da aplicação, gerar parte do código (através da ferramenta), e complementar este código, enquanto a segunda aplicação deveria ser implementada sem o seu auxílio, ou seja, através de métodos tradicionais, que consiste em utilizar o Visual Studio para produzir todo o código. Ficou a critério dos desenvolvedores a escolha do método para cada aplicação.

A primeira aplicação consistia em um filtro de média em uma imagem 1024x1024, e a segunda, em uma soma de duas matrizes transpostas, de tamanho 1024x1024. O objetivo deste teste foi avaliar o tempo que os desenvolvedores levariam e o esforço para desenvolver as aplicações especificadas, e averiguar se a ferramenta seria ou não útil neste processo de desenvolvimento. A Tabela 1 contém os resultados.

Tabela 1. Resultados dos testes.

Desenvolvedor	Tempo (sem a Ferramenta)	Tempo (com a Ferramenta)
1	29 minutos	25 minutos
2	43 minutos	28 minutos
3	43 minutos	36 minutos
4	80 minutos	81 minutos
5	60 minutos	29 minutos
6	20 minutos	16 minutos

Nota-se que na maioria dos casos os desenvolvedores levaram um tempo menor na situação em que eram auxiliados pela ferramenta. Os principais relatos foram os gastos de tempo na correção de pequenos *bugs*, porém de difícil identificação, na situação em que a ferramenta não foi utilizada, e pequenas dificuldades na utilização da ferramenta, mas que não impactaram no desenvolvimento da aplicação.

7. Conclusões

O presente trabalho demonstrou o uso do desenvolvimento baseado em modelos aplicado à área de GPGPU, através da implementação do CudaMDA, uma ferramenta que automatiza parte do processo de construção de um software nesta plataforma através de modelos.

A principal contribuição da área de Engenharia de Software foi atingida neste trabalho, ou seja, o aumento de produtividade na elaboração de aplicações, conseqüência da redução no tempo para desenvolvimento de uma aplicação usando a ferramenta CudaMDA. A abordagem orientada a modelos adotada neste trabalho fornece uma visão de alto nível da aplicação, o que reduz a probabilidade de inserção de erros por parte do desenvolvedor, algo comum quando se está lidando com programação de baixo nível, que normalmente trabalha com códigos não triviais. Além disso, há o benefício de que o modelo funciona como uma auto-documentação da aplicação e aumenta a qualidade do software final.

Uma possível extensão do trabalho seria abordar a automatização na construção de softwares em outras

plataformas de programação, como Java ou C++. O refinamento do modelo do domínio utilizado na ferramenta, com o objetivo de aperfeiçoar a representação de uma aplicação na plataforma GPGPU também pode ser considerada uma extensão do trabalho desenvolvido.

Otimizações no código gerado também agregariam bastante valor à ferramenta, já que se trata de um contexto em que o desempenho das aplicações é fundamental. Além disso, a interface gráfica do *plugin* precisa ser avaliada do ponto de vista de usabilidade, e caso necessário, adaptada após esta avaliação.

8. Referências

[1] Stephen J. Mellor, Anthony N. Clark, Takao Futagami, Guest Editors' Introduction: Model-Driven Development, IEEE Software, vol. 20, no. 5, pp. 14-18, Sep/Oct, 2003.

[2] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. 2005. A survey of general-purpose computation on graphics hardware. In Proceedings of Eurographics 2005, State of the Art Reports. 21-51.

[3] Eclipse Foundation. Graphical Modeling Framework (GMF). http://www.eclipse.org/gmf.

[4] NVIDIA. CUDA Programming Guide. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, acesso em: Novembro de 2008.

[5] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. Proceedings of the IEEE, 96(5), May 2008.

[6] Shirley, P. 2002. Fundamentals of Computer Graphics. A. K. Peters, Ltd.

[7] Fernando, R. and Kilgard, M. J. 2003. The Cg Tutorial: the Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley Longman Publishing Co., Inc.

[8] Tarditi, D., Puri, S., and Oglesby, J. 2006. Accelerator: using data parallelism to program GPUs for general-purpose uses. SIGOPS Oper. Syst. Rev. 40, 5 (Oct. 2006), 325-335.

[9] Rost, R. J. 2004. Opengl(R) Shading Language. Addison Wesley Longman Publishing Co., Inc.

[10] Microsoft MSDN. 2006. HLSL. http://msdn.microsoft.com/enus/library/bb509561(VS.85).aspx.

[11] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. 2004. Brook for GPUs: Stream Computing on Graphics Hardware. ACM Trans. Graph. 23, 3 (Aug. 2004), 777-786.

[12] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. Computational Genomics (to appear).

[13] Microsoft. 2005. Microsoft Visual Studio.

[14] Rumbaugh, J., Jacobson, I., and Booch, G. The UML Reference Manual. Addison-Wesley, Reading, England, 1999.

[15] Selic, B. 2003. The Pragmatics of Model-Driven Development. IEEE Softw. 20, 5 (Sep. 2003), 19-25.

[16] J. Miller and J. Mukerji, MDA Guide Version 1.0.1, doc. no. omg/2003-06-01, Junho 2003. http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf.

[17] AndroMDA.org. 2008. AndroMDA. http://www.andromda.org/.

[18] Compuware. 2008. OptimalJ. http://www.compuware.com/.

[19] Quatrani, T. 1998. Visual Modeling with Rational Rose and UML. Addison-Wesley Longman Publishing Co., Inc.

[20] Tigris. 2006. ArgoUML. http://argouml.tigris.org/.

[21] Rebel Science. 2006. COSA Project. http://www.rebelscience.org/Cosas/COSA.htm.

[22] Boulet, P.: Array-OL revisited, multidimensional intensive signal processing specification. Research Report RR-6113, INRIA (2007).

[23] Ben Atitallah, R., Boulet, P., Cuccuru, A., Dekeyser, J.L., Honor´e, A., Labbani, O., Le Beux, S., Marquet, P., Piel, E., Taillard, J., Yu, H.: Gaspard2 uml profile documentation. Technical Report 0342, INRIA (2007)

[24] Open SystemC Initiative, SystemC. http://systemc.org/.

[25] Eclipse Foundation. 2004. Eclipse. http://www.eclipse.org/.

[26] Arnold, K. and Gosling, J. 1998. The Java Programming Language (2nd Ed.). ACM Press/Addison-Wesley Publishing Co.

[27] Eclipse Foundation. Eclipse Modeling Framework (EMF). http://www.eclipse.org/emf.

[28] Eclipse Foundation (2007a), 'Eclipse Modeling: Java Emitter Templates'. http://www.eclipse.org/emft/projects/jet/.