ADD - Uma Ferramenta de Projeto de Aceleradores com DataFlow para Alto Desempenho*

Jeronimo Penha[1], Lucas Bragança[1], Danilo Almeida[1], Jose Nacif[1], Ricardo Ferreira[1]

[1]Departamento de Informática – Universidade Federal Viçosa (UFV)
CEP: 36.570-900 – Viçosa – Minas Gerais – Brazil

{jeronimopenha,danilooalmeida94}@gmail.com

{lucas.braganca, jnacif, ricardo}@ufv.br

Resumo. *Aceleradores com FPGA baseados em fluxo de dados se tornaram uma alternativa promissora para se conseguir alto desempenho com eficiência energética. Este artigo apresenta a ferramenta ADD (Accelerator Design and Deploy) para descrição de algoritmos com fluxo de dados, que também possibilita a simulação, a prototipação em FPGA e a integração em um ambiente heterogêneo CPU-FPGA. A ferramenta possui uma biblioteca de operadores síncronos, além de possibilitar o desenvolvimento de novos operadores. Oferece suporte para acoplamento com linguagens de programação de alto nível e foi validada na plataforma para computação heterogênea CPU-FPGA de alto desempenho da Intel/Altera. Como resultado, obteve-se ganhos no tempo de processamento dos benchmarks de até seis vezes em relação às execuções single thread, o que mostra a eficiência da ferramenta proposta.*

1. Introdução

Um dos grandes desafios da computação é obter alto desempenho com eficiência energética. Neste cenário, aceleradores como GPUs e FPGAs tem alto desempenho ao explorar o paralelismo, e ao mesmo tempo, baixo consumo de energia em relação aos processadores de uso geral. Ademais, aplicações mapeadas em FPGA na forma de grafo de fluxo de dados (*data flow graphs* - DFG) reduzem o consumo de energia, pois não é necessário fazer a busca e a decodificação das instruções a cada ciclo de relógio. Entretanto, os FPGAs têm como maior barreira a dificuldade para modelagem, programação e compilação de aplicações como mostrado em [Stitt 2011].

Uma alternativa é o uso de OpenCL (*Open Computing Language*) [Munshi 2009]. A Intel/Altera e a Xilinx (os dois maiores fabricantes) disponibilizaram novas versões de compiladores com C/C++ e OpenCL para mapeamento direto e eficiente em FPGA [OpenCL , IntelFPGA , Xilinx]. A vantagem do OpenCL é o fato de ser difundido na comunidade de alto desempenho, ter a linguagem C/C++ como base e o usuário precisaria aprender apenas as primitivas para expressar o paralelismo. Entretanto, se faz necessário conhecimento do processo de mapeamento do código no fluxo de dados e das arquiteturas de FPGA para fazer ajustes, compreender as informações geradas pelas ferramentas e otimizar o código.

Outra alternativa é o OpenSPL [Kim et al. 2010] que é baseado em Java. O ambiente OpenSPL permite a visualização dos grafos de fluxo de dados gerados para código

*Financiamento: FAPEMIG, CAPES e CNPq

e possui primitivas para trabalhar com os fluxos. Assim como em OpenCL, o OpenSPL mapeia o código parte para processador, parte para o FPGAs de modo automático, assim como o acoplamento entre as duas plataformas. Em ambas abordagens, o código é transformado de forma implícita em um *Data-Flow Graph* (DFG) pelo compilador, portanto o programador não tem total controle sobre o DFG gerado e ao mesmo tempo precisa saber codificar e observar o DFG gerado para fazer ajustes.

Este artigo propõe a ferramenta ADD (*Accelerator Design and Deploy*) para o desenvolvimento de algoritmos por meio de grafos de fluxo de dados a serem mapeados em FPGA que possuem uma interface com códigos escritos em linguagens de alto nível (*Accelerator Design*). Provê facilidades de mapeamento e execução em FPGA (*Deploy*) o que reduz a curva de aprendizado e desenvolvimento. O objetivo é auxiliar a modelagem algoritmos e operadores com o intuito de familiarizar os projetistas com os conceitos de fluxo de dados. Como contribuição, a ferramenta permite realizar a simulação em nível de DFG, com geração automática de código para execução em FPGAs acoplados a processadores. O projeto gerado inclui também uma interface de acoplamento com as plataformas de software para que possam fazer chamadas e transferir dados para o acelerador. O ADD complementa as abordagens de OpenCL e OpenSPL, porém o DFG deve ser explicitamente descrito.

Este trabalho se divide em cinco seções. Na Seção 2, a ferramenta de desenvolvimento e implantação de aceleradores ADD é apresentada. Nesta Seção, as características de funcionamento e utilização da ferramenta são descritas, tais como: a construção de DFGs, suporte à criação de algoritmos, operadores contidos na biblioteca, desenvolvimento de novos operadores, a geração de *Verilog*, simulação e execução em FPGA. A apresentação dos resultados dos experimentos com a execução dos algoritmos: *Gourand*, *FIR8*, *FIR16*, *Histogram*, *Paeth* e *Reduce SUM*; projetados no **ADD** e executados no ambiente de alto desempenho híbrido **CPU-FPGA**, da fabricante **Intel/Altera**, são apresentados e discutidos na Seção 3. A Seção 4 aborda trabalhos relacionados e, por fim, as considerações finais são apresentadas na Seção 5.

2. Ferramenta ADD

A ferramenta ADD permite a criação, simulação e validação de DFGs em uma interface gráfica que possibilita a edição e alteração dos algoritmos a serem construídos. A simulação foi implementada como extensão do editor/simulador HADES [Hendrich 2000] através da criação de uma biblioteca com operadores descritos em Java. O usuário pode também desenvolver novos operadores e adicionar a biblioteca. A ferramenta automaticamente converte o grafo criado para a linguagem de descrição de hardware, no caso *Verilog*, para que seja sintetizada em FPGAs usando ferramentas da Xilinx ou Intel/Altera.

As linguagens de descrição de *hardware* como VHDL e *Verilog* HDL são complexas porque a sua utilização exige um conhecimento sobre *hardware* e seu funcionamento. Isto traz barreiras aos profissionais de desenvolvimento de *software* [Stitt 2011]. Para contornar essa dificuldade as linguagens OpenCL/OpenSPL inserem palavras reservadas (*pragmas* e/ou estruturas de dados) em linguagens de alto nível que possibilitam a criação de algoritmos que serão mapeados em *hardware* de modo integrado às linguagens C/Java. O código gerado é sintetizado com ferramentas comerciais de FPGA, como

Quartus da Intel/Altera e depois executados na plataforma CPU-FPGA. Estes ambientes permitem apenas visualizar o grafo de fluxo de dados gerado mas sem a possibilidade de edição [Ling et al. 2017, Winans 2015].

Complementando o OpenCL e OpenSPL, o ADD exercita o desenvolvimento de algoritmos na forma explícita de DFG que são automaticamente mapeados em FPGA, além de permitir a criação de novos operadores que amplia as possibilidades de descrição de novos DFGs. O processo envolve dois passos. O primeiro é a especificação do DFG onde é possível simular, realizar depuração e analisar o desempenho no nível de DFG. Posteriormente, um gerador mapeia o DFG em código *Verilog*, que posteriormente pode ser sintetizado para um FPGA, semelhante ao fluxo de projeto de OpenCL/OpenSPL. O segundo passo é a execução do código em uma plataforma heterogênea CPU-FPGA.

O ADD oferece suporte às linguagens Java e C/C++ para comunicação entre seu código na transferência de dados e acionamento do acelerador que é encapsulado por uma chamada de função. A interface de comunicação foi desenvolvida para duas plataformas de FPGA. A primeira de baixo custo com fins educacionais, usa a interface JTAG para a comunicação computador-FPGA e pode ser usada em diversos kits da fabricante Altera conectados a computadores comuns (*notebook* ou *desktop*), permitindo acesso a muitos programadores e estudantes. A segunda plataforma é de alto desempenho, desenvolvida pela Intel, que combina um processador Xeon e um FPGA acoplados à memória compartilhada por meio do barramento QPI [Gupta 2016, Gupta 2015].

2.1. Operadores

Para descrição dos DFG, a biblioteca de operadores é organizada nas seguintes categorias: acumuladores, aritméticos, *branches*, comparadores, I/O, lógicos, registrador, memória e *shift*. Os operadores foram baseados em uma equivalência com instruções RISC. A lista de operadores pode ser observada na Tabela 1. Na tabela pode-se ver os nomes dos operadores separados por categorias. Os operadores que possuem a letra "I"no fim do nome são operadores que trabalham com valores imediatos. Além dos operadores de uso geral, foram implementados alguns operadores específicos para exemplificar novas possibilidades para uso de DFGs. Por exemplo, o operador *Histogram* que possui uma memória interna para a construção de um histograma e os operadores acumuladores.

Diferente de uma GPU, o DFG permite a execução simultânea de diversos caminhos divergentes em pipeline. Para a lógica de controle de fluxo foram propostos inicialmente os operadores *Branch* e *Merge*. As possibilidades de execução dos desvios geram fluxos independentes e o operador *Merge* é o responsável por selecionar qual deles será repassado aos próximos operadores. Os operadores podem ser configurados para se ajustar ao algoritmo. Estes ajustes podem ser, por exemplo, a quantidade de bits de uma porta de entrada ou saída.

A Figura 1 apresenta um exemplo com controle de fluxo e um código ilustrativo (a). Nela pode-se observar como modelar os *Branches*. O problema possui três possibilidades diferentes para a computação do fluxo de saída *Y*. Neste exemplo, as três computações são executadas e ao fim, apenas uma é selecionada. Observe que as condições e as computações são executadas em paralelo. As variáveis *I* representam os valores imediatos de cada operador.

O grafo apresentado na Figura 1 pode ser replicado várias vezes, para explorar

Tabela 1. Lista de operadores disponíveis na biblioteca do ADD.

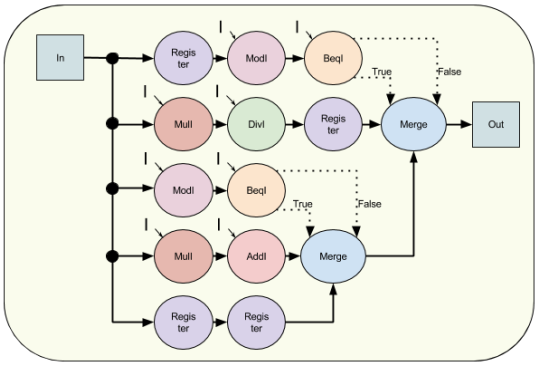
Categoria	Operador	Função
Aritméticos	Abs	$y = \|a\|$
	Add	$y = a + b$
	Div	$y = a/b$
	Mod	$y = a\%b$
	Mul	$y = a * b$
	Sub	$y = a - b$
	AddI	$y = a + I$
	DivI	$y = a/I$
	ModI	$y = a\%I$
	MulI	$y = a * I$
	SubI	$y = a - I$
Lógicos	And	$y = a\&b$
	Or	$y = a\|b$
	Not	$y = \ a$
	AndI	$y = a\&I$
	OrI	$y = a\|I$
I/O	In 1-32	-
	Out 1-32	-
Memória	Histogram	-
Registrador	Register	$y = a$

Tipo	Operador	Função
Controladores de Fluxo	Beq	$if = (a == b) : 1; 0$ $else = (a == b) : 0; 1$
	Bne	$if = (a! = b) : 1; 0$ $else = (a! = b) : 0; 1$
	BeqI	$if = (a == I) : 1; 0$ $else = (a == I) : 0; 1$
	BneI	$if = (a! = I) : 1; 0$ $else = (a! = I) : 0; 1$
	Merge	$y = a$ se $if == 1$ $y = b$ se $else == 1$
Shift	Shl	$y = a << b$
	Shr	$y = a >> b$
	ShlI	$y = a << I$
	ShrI	$y = a >> I$
Comparadores	Max	$y = Max(a, b)$
	Min	$y = Min(a, b)$
	Slt	$y = (a < b)?1 : 0$
	MaxI	$y = Max(a, I)$
	MinI	$y = Min(a, I)$
	SltI	$y = (a < I)?1 : 0$
Acumuladores	AccAdd	-
	AccMax	-
	AccMin	-
	AccMul	-

```
...
foreach x in  stream_In
  if(x%2 == 0){
    y = (x*2)/1;
  }else if(x%3 == 1){
    y = (x*3)+2;
  }else {
    y = x;
  }

}
...
```

(a)



(b)

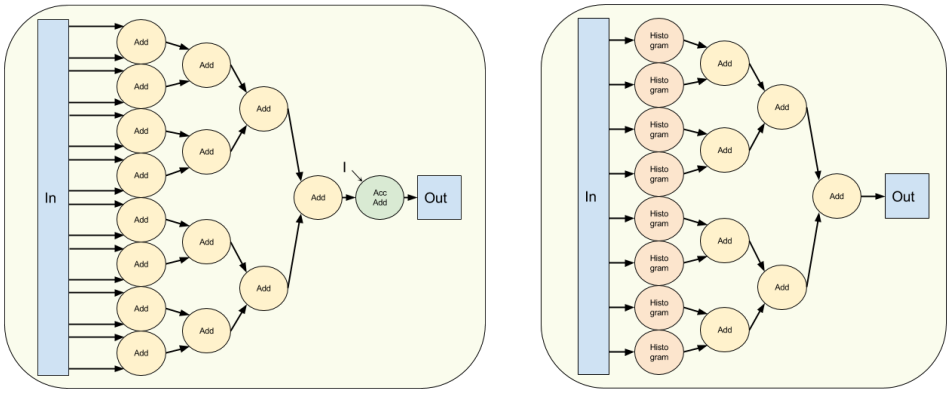
Figura 1. Exemplo de algoritmo com *Branches* (a) e o DFG equivalente esquemático com os operadores da biblioteca do ADD (b).

o paralelismo espacial. A versão atual do ADD traz a possibilidade da utilização de operadores de entrada de dados ("*In*") com até trinta e duas saídas, o que permite que trinta e duas cópias sejam executadas em paralelo. É possível alterar estes operadores e criar novos com mais recursos, como será mostrado na Seção 2.2. Outra opção para o exemplo anterior seria calcular a condição e usá-la para alimentar apenas um dos fluxos que irá executar a computação correta através do uso de novos operadores.

Outro exemplo de operador específico é a operação de redução com o auxílio dos

operadores *acumuladores*. O acumulador processa e armazena o resultado temporário. Após todos os elementos do vetor serem processados, o resultado da redução é entregue para o estágio posterior. Na Figura 2(a), um exemplo de redução de dezesseis elementos por vez. Diferente de uma GPU que pode ter baixa ocupação no final da redução, a abordagem com DFG alocou, neste exemplo, 15 operadores e todos são usados ao mesmo tempo em *pipeline*.

Outro exemplo é a modelagem de histogramas onde o operador possui uma memória interna e contadores. A execução é realizada em paralelo seguida de uma redução. A quantidade de dados a serem lidos pelo operador pode ser configurada a cada execução da mesma forma que a configuração dos operadores que trabalham com imediatos e dos operadores acumuladores. Um exemplo de histograma com oito operadores trabalhando em paralelo com redução é apresentado na Figura 2 (b) seguido de uma redução.



(a) (b)

Figura 2. Algoritmos *Reduce Add* (a) e *Histogram* (b).

2.2. Desenvolvimento de Novos Operadores

Novos operadores podem ser criados para darem suporte à implementação de mais algoritmos, modelos e ferramentas de projetos com novas linguagens e compiladores. A biblioteca oferece operadores genéricos e parametrizáveis que foram criados para servirem como uma base para novos operadores. Na versão atual, os operadores base possuem as características: número de portas de entrada (1 ou 2), acúmulo de valores, controle de fluxo (*Branches*) com e sem uso de valores imediatos, controle de entrada de dados, controle de dados de saída de dados. Através da herança na linguagem Java, os novos operadores podem ser rapidamente codificados e testados para simulação. Para execução no FPGA, a descrição *Verilog* do novo operador deve ser adicionada ao gerador do ADD.

A Figura 3 apresenta o código que implementa o operador *MulI*, incluso na biblioteca do ADD. Este operador executa a multiplicação do valor do barramento de entrada de dados por uma constante (imediato) e herda as características básicas do operador genérico *GenericI*.

2.3. Geração de *Verilog*

O gerador do ADD permite ao programador abstrair do código *Verilog* e da complexidade de todos os detalhes de sincronismo no nível de circuito, que envolvem máquinas de esta-

```
package add.dataflow.sync;
public class MulI extends GenericI {
    public MulI() {
        super();
        setCompName("MULI");
    }
    @Override
    public int compute(int data) {
        setString(Integer.toString(id), Integer.toString(immediate));
        return (int) (data * immediate);
    }
}
```

Figura 3. Código que o implementa o operador MulI.

dos, sinais de relógio, *reset*, habilitação, codificação, etc. A descrição *Verilog* sintetizável é gerada com o auxílio da biblioteca Veriloggen [Takamaeda-Yamazaki]. Semelhante ao fluxo de projeto em OpenCL da Intel, o código gerado pelo ADD deve ser sintetizado com uma ferramenta de projeto de FPGA como o Quartus.

2.4. Simulação, Comunicação e Execução no FPGAs

A comunicação com o acelerador é feita através de filas de entrada e saída de dados, seja no nível de simulação e/ou execução. O uso das filas de entrada e saída desacopla o DFG da plataforma FPGA que irá implementar o sistema. Duas interfaces foram implementadas, uma fracamente e outra fortemente acopladas. A primeira usa a interface JTAG para a comunicação com os FPGAs e pode ser usadas em kits para fins didáticos e de depuração. A segunda desenvolvida com o uso da API AAL/QPI da Intel/Altera para acoplamento CPU-FPGA por memória compartilhada.

O ADD traz consigo uma API com métodos que permitem a simulação e a execução do circuito desenvolvido. O simulador HADES [Hendrich 2000] foi escolhido como base para o desenvolvimento da ferramenta ADD por ser portátil e por possuir flexibilidade para a criação de novos operadores, várias extensões do HADES já foram propostas [Penha et al. 2016, Ferreira et al. 2015, Ferreira et al. 2005, Ferreira et al. 2004, Marwedel et al. 2002], porém outros simuladores podem ser utilizados futuramente.

A execução com o FPGA se dá de forma semelhante à execução no simulador. O aplicativo a ser executado deverá utilizar a API do ADD para iniciar a transferência de dados para o FPGA e após a execução repassar os dados processados de volta à chamada da função. O método responsável por esta execução é bloqueante, isto é, a instrução seguinte a chamada será executada apenas ao término do processamento do DFG. Para que o aplicativo que fará uso do acelerador não fique bloqueado, a chamada para a API deve ser executada em uma *thread* independente, assim o aplicativo ficará livre para executar outras tarefas durante a execução do DFG.

Para a execução dos aceleradores em FPGAs, é necessária a conversão do arquivo estrutural criado pelo ADD para uma linguagem de descrição de hardware e uma interface de acoplamento entre o sistema e o FPGA. O ADD traz consigo uma implementação com JTAG capaz de executar DFGs em kits de FPGA. Detalhes sobre o funcionamento da interface disponível podem ser vistos na Figura 4. A transferência de dados se dá por

meio do JTAG e, internamente, os circuitos de interface repassam e recebem os resultados do DFG. O ADD e juntamente com a interface JTAG podem ser obtidos em: `https://github.com/ComputerArchitectureUFV/ufv_add.git`.

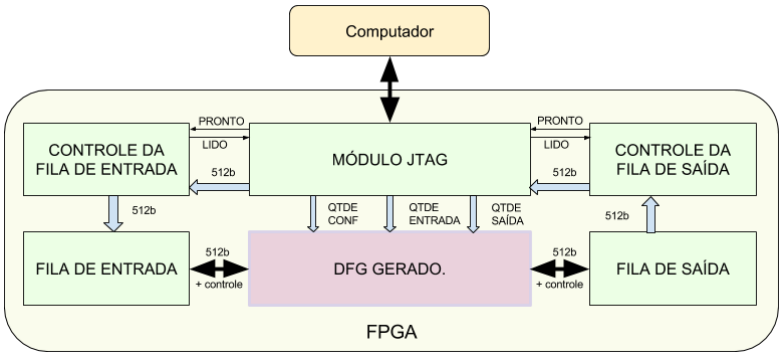


Figura 4. Interface para execução em FPGAs.

O ADD pode suportar novas plataformas, como o Catapult da Microsoft [Putnam et al. 2014], com a adição de pequenas modificações em sua API. Além da interface JTAG, o ADD possui suporte para a execução na plataforma híbrida CPU-FPGA da fabricante Intel/Altera. A Figura 5 detalha o processo de execução dos aceleradores com as duas interfaces: JTAG e XEON/FPGA com QPI.

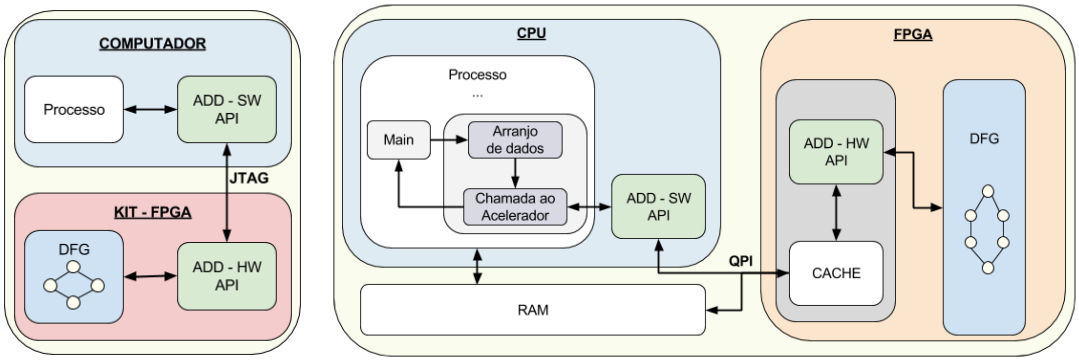


Figura 5. (a) Execução de um DFG em kit de FPGA; (b) Ambiente de alto desempenho da fabricante Intel/Altera

2.5. Protocolo de Comunicação das Interfaces de Entrada e Saída de Dados

A comunicação entre os operadores de controle de dados e as FIFOs de entrada e saída segue um protocolo de comunicação simples e funcional que proporciona um desacoplamento entre o FPGA e o processo solicitante. Para a entrada de dados, a fila de entrada informa ao circuito a existência de dados a serem processados. O controlador de entrada de dados do DFG então realiza leitura dos mesmos e os repassa para processamento. Caso seja necessária a configuração de operadores que trabalham com imediatos, as configurações são executadas antes do início do processamento. Este processo de controle é transparente para a fila de entrada. Sempre que houver a falta de dados para leitura,

devido a atrasos na transferência, o processamento é interrompido e reiniciado até que hajam dados disponíveis.

O retorno dos dados processados segue um protocolo semelhante ao de entrada. Caso a fila de saída fique cheia, o controlador de saída paralisa o processamento momentaneamente até que a fila de saída esteja novamente disponível. O programador não precisa se preocupar com o controle de entrada e saída que é implementado pela API.

3. Experimentos e Resultados

Para a validação do funcionamento e da eficiência dos DFGs desenvolvidos usando o ADD, foi usado um conjunto de *benchmarks* de algoritmos para processamento de sinais que podem ser vistos na Tabela 2. Os recursos necessários para a gravação no FPGA STRATIX V da plataforma de alto desempenho da fabricante Intel/Altera são mostrados na tabela nos campos Elementos Lógicos (*ALMs*), Registradores (*Reg*), módulos de memória embarcadas *M20K* e módulos *DSPs*. Os *benchmarks* foram replicados respeitando as restrições de número de entrada e saída de cada DFG e limitados ao máximo de trinta e duas portas de 16 bits. Esta limitação é devido à restrições do barramento implementado pela Intel/Altera que trabalha com pacotes de 64 bytes, onde os testes foram realizados. Na coluna "*Benchmark*"pode-se ver entre parênteses o número de cópias paralelas que foram sintetizadas para cada DFG.

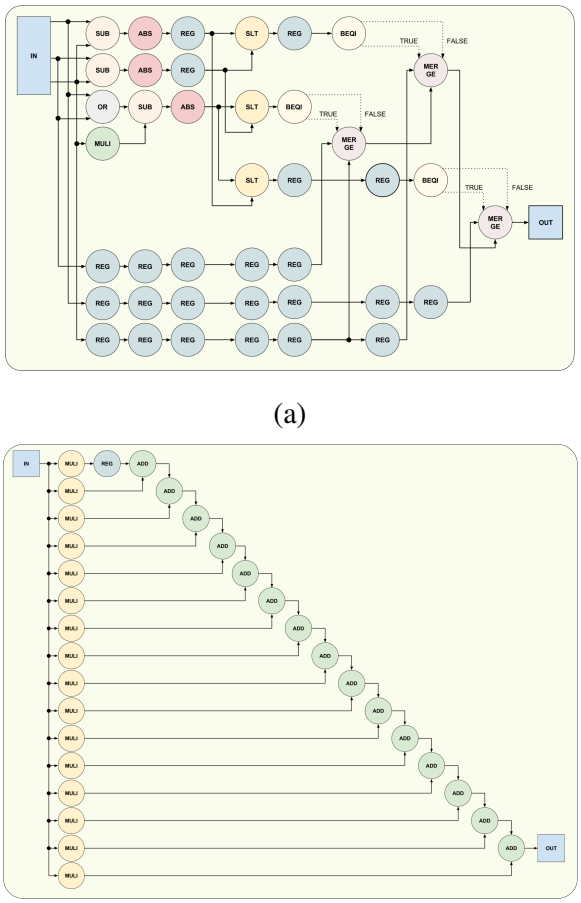
Tabela 2. Relação de resultados de execução dos algoritmos em CPU e FPGA.

Benchmark	Stratix V 5SGXEA7N1F45C1					CPU	T(cpu)/T(fpga)
	ALMs (%)	Registers	M20K	DSP (%)	T(ms)	T(ms)	
Gourand (8x)	35	82.352	168	0	168	358	2,130
FIR 8 (32x)	37	89.726	168	100	229	553	2,414
FIR 16 (32x)	44	104.111	168	100	207	1.394	6,734
Histogram (32x)	98	233.206	168	0	135	165	1,223
Paeth (8x)	35	84.226	168	3	170	335	1,970
Reduce SUM (32x)	35	78.988	168	0	135	77	0,570

O sistema CPU-FPGA usa *FPGA Stratix V 5SGXEA7N1F45C1* fortemente acoplado a um processador XEON com 10-*cores*. Esta plataforma permite o acesso à memória do computador através do barramento QPI que realiza a transferência em pacotes de 64 *Bytes* a uma taxa de transferência de aproximadamente 6 GBps [Choi et al. 2016]. A plataforma traz implementado um sistema de requisições de dados com cache que é responsável pela solicitação e gravação de dados na memória RAM. A API-HW do ADD trabalha em conjunto com esta estrutura para que o acelerador possa ser acessado e utilizado. Os circuitos foram sintetizados através da IDE Quartus II 64-Bit Version 13.1.0 e os recursos utilizados para a implementação de cada algoritmo foi retirado pelo relatório de síntese do mesmo.

Com intuito de demonstrar a eficiência dos DFGs criados, foi feita a comparação com a execução dos mesmos algoritmos no processador Intel(R) Xeon(R) CPU E5-2680 v2 com frequência de *clock* de 2.80GHz. Para a implementação foi utilizada a linguagem C e a compilação e otimização foi feita pelo compilador GCC 4.8.4. Os tempos de execução, levando em consideração o tempo de transferência de dados, em milissegundos, para cada algoritmo foram medidos e podem ser observados na Tabela 2. A intenção

da comparação é demonstrar que os DFGs criados através da ferramenta são eficientes. Como ilustração, os DFGs para uma instância dos algoritmos FIR16 e Paeth podem ser vistos na Figura 6.



(a)

(b)

Figura 6. DFGs para os algoritmos Paeth (A) e FIR 16 (B).

Para a execução dos *benchmarks* foram utilizados 512MB de dados em palavras de 16 *bits* tanto para o acelerador quando para o processador. Os resultados exibidos na Tabela 2 mostram que os tempos de execução para o acelerador são competitivos como os apresentados pela execução em CPU de forma sequencial. O melhor resultado foi observado na execução do algoritmo FIR16, replicado 32 vezes, que foi seis vezes mais rápido do que a execução do mesmo algoritmo na CPU. É importante levar em consideração que o mesmo utilizou 100% dos DSPs disponíveis no FPGA, o que aumenta consideravelmente a resposta deste algoritmo. As instruções básicas do mesmo são multiplicação e soma. Em relação a execução para o Reduce SUM, o tempo medido foi superior ao da CPU pelo fato de este realizar a soma de apenas 32 elementos em paralelo e a operação executada ser simples.

4. Trabalhos Relacionados

Um DFG expressa explicitamente o paralelismo do código, o que possibilita a execução de partes de algoritmos em elementos processadores independentes em diversas plataformas. Trabalhos recentes exploram vários aspectos dos DFGs para as plataformas heterogêneas com FPGAs, GPUs e arquiteturas *multi-core* e *many-cores*. Uma abordagem de otimização de DFG baseada em transformações foi apresentada em [Stewart et al. 2017]. Para um algoritmo de processamento de vídeo, os resultados experimentais mostram uma melhoria de aproximadamente 50% na frequência do FPGA que superaram as otimizações das ferramentas comerciais de FPGA.

Uma abordagem para vetorização de código para GPUs a partir de DFGs com atores foi apresentada [Barford et al. 2014], onde um DFG de um filtro e um DFG com máquinas de estados foram avaliados. Considerando arquiteturas *many-cores* com estruturas de conexão baseadas em *network-on-chip*, a abordagem apresentada em [Ul-Abdin and Yang 2017], mostra, para um estudo de caso, que a partir de um DFG, um ganho de aceleração de até 4 vezes pode ser obtido na arquitetura *many-core* com 10 núcleos em comparação com um processador embarcado.

Um outro exemplo de otimização baseada na modelagem de problemas com DFG para sistemas embarcados foi apresentado recentemente em [Palumbo et al. 2017] onde o objetivo é reduzir a potência dissipada e a arquitetura alvo utilizada foi um arranjo reconfigurável CGRAs (*Coarse-Grained Reconfigurable Arrays*). Em comparação com ASICs de 45nm, os circuitos desenvolvidos conseguiram uma redução de mais de 70% no consumo de energia estático e mais de 90% no consumo dinâmico.

O trabalho proposto tem como finalidade a criação visual de DFGs e a possibilidade de simulação e execução em vários ambientes. Diferentemente dos trabalhos relacionados, o foco é no desenvolvimento e depuração de DFGs, enquanto que nos trabalhos relacionados o foco é mais concentrado na otimização de DFGs já desenvolvidos. Não foram encontrados outros trabalhos atuais com proposta semelhante a deste para que uma melhor comparação pudesse ser feita.

5. Considerações finais

Este trabalho apresenta a ferramenta ADD para uso de grafos de fluxo de dados para a criação de projetos com a possibilidade de simulação e testes de algoritmos em FGPAs. O ADD é mais uma opção de ambiente de desenvolvimento que traz uma forma alternativa de projetar DFGs em um ambiente flexível que pode ser expandido com novos operadores. É acessível por possibilitar seu uso em kits didáticos de FPGAs, além de possibilitar a execução em plataformas heterogêneas de alto desempenho. Um conjunto de algoritmos foi descrito na forma de DFG e o ADD fez a transformação automática para mapeamento em FPGA.

Os exemplos foram avaliados na plataforma de alto desempenho da Intel composta por um processador XEON fortemente acoplado através da memória a um FPGA. A API do ADD abstraiu a API de software e hardware da Intel permitindo o uso da plataforma de forma transparente a partir do DFG inicial. A interface de comunicação gerada pelo ADD provê um sistema de filas que promove o desacoplamento entre o DFG e a plataforma alvo o que permite também a comunicação através da interface JTAG para validação dos DFGs em FPGAs de baixo custo.

Para trabalhos futuros, pretende-se aprimorar a ferramenta para que dê suporte a mais ambientes com FPGA já consolidados, híbridos ou não, acoplamento com compiladores e linguagens de domínio específico, outras ferramentas de geração ou transformação de código e otimização dos DFGs desenvolvidos na ferramenta. Outro aspecto é o mapeamento para arquiteturas de grão grosso como os CGRAs (*Coarse-Grained Reconfigurable Architecture*), que podem ser implementados como *overlays* em FPGAS ou diretamente em silício.

Referências

Barford, L., Bhattacharyya, S. S., and Liu, Y. (2014). Data flow algorithms for processors with vector extensions: handling actors with internal state. In *Signal and Information Processing (GlobalSIP), 2014 IEEE Global Conference on*, pages 20–24. IEEE.

Choi, Y.-k., Cong, J., Fang, Z., Hao, Y., Reinman, G., and Wei, P. (2016). A quantitative analysis on microarchitectures of modern cpu-fpga platforms. In *Design Automation Conference (DAC)*. ACM/EEE.

Ferreira, R., Cardoso, J. M., and Neto, H. C. (2004). An environment for exploring data-driven architectures. In *Int. Conference on Field-Programmable Logic and Applications (FPL)*.

Ferreira, R., Cardoso, J. M., Toledo, A., and Neto, H. C. (2005). Data-driven regular reconfigurable arrays: design space exploration and mapping. In *Int. Conf. on Embedded Computer Systems Architectures, Modeling and Simulation SAMOS*.

Ferreira, R., Nacif, J., Magalhaes, S., de Almeida, T., and Pacifico, R. (2015). Be a simulator developer and go beyond in computing engineering. In *Frontiers in Education Conference (FIE)*. IEEE.

Gupta, P. (2016). Accelerating datacenter workloads. In *26th International Conference on Field Programmable Logic and Applications*.

Gupta, P. K. (2015). Xeon+ fpga platform for the data center. In *Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, volume 119.

Hendrich, N. (2000). A java-based framework for simulation and teaching: Hades—the hamburg design system. In *Microelectronics Education*, pages 285–288. Springer.

IntelFPGA. Intel FPGA - Acceleratiing The Smart and Connected World. `https://www.altera.com/`. Accessed: 2017-08-09.

Kim, E., Kim, K., and In, H. P. (2010). A multi-view api impact analysis for open spl platform. In *Advanced Communication Technology (ICACT), 2010 The 12th International Conference on*, volume 1, pages 686–691. IEEE.

Ling, A. C., Aydonat, U., O'Connell, S., Capalija, D., and Chiu, G. R. (2017). Creating high performance applications with intel's fpga opencl™ sdk. In *Proceedings of the 5th International Workshop on OpenCL*, page 11. ACM.

Marwedel, P., Cong, K., and Schwenk, S. (2002). Ravi: Interactive visualization of information system dynamics using a java-based schematic editor and simulator.

Munshi, A. (2009). The opencl specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–314. IEEE.

OpenCL. Opencl™ zone – Accelerate Your Applications. http://developer.amd.com/tools-and-sdks/opencl-zone/. Accessed: 2017-08-09.

Palumbo, F., Fanni, T., Sau, C., and Meloni, P. (2017). Power-awarness in coarse-grained reconfigurable multi-functional architectures: a dataflow based strategy. *Journal of Signal Processing Systems*, 87(1):81–106.

Penha, J. C., Fontes, G., and Ferreira, R. (2016). MIPSFPGA - Um simulador mips incremental com validação em fpga. *International Journal in Computer Architecture Education (IJCAE)*, 5(1):19–25.

Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., et al. (2014). A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE.

Stewart, R., Bhowmik, D., Wallace, A., and Michaelson, G. (2017). Profile guided dataflow transformation for fpgas and cpus. *Journal of Signal Processing Systems*, 87(1):3–20.

Stitt, G. (2011). Are field-programmable gate arrays ready for the mainstream? *IEEE Micro*, 31(6):58–63.

Takamaeda-Yamazaki, S. Veriloggen: A library for constructing a verilog hdl source code in python. https://github.com/PyHDI/veriloggen. Accessed: 2017-07-20.

Ul-Abdin, Z. and Yang, M. (2017). A radar signal processing case study for dataflow programming of manycores. *Journal of Signal Processing Systems*, 87(1):49–62.

Winans, J. (2015). On dataflow computing with openspl.

Xilinx. Xilinx – All Programmable. https://www.xilinx.com/. Accessed: 2017-08-09.