JERONIMO COSTA PENHA

FERRAMENTAS PARA GERAÇÃO AUTOMÁTICA DE ACELERADORES EM PLATAFORMAS HETEROGÊNEAS DE ALTO DESEMPENHO COM FPGA

Dissertação apresentada a Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2018

Ficha catalográfica preparada pela Biblioteca Central da Universidade Federal de Viçosa - Câmpus Viçosa

T
P399f
2018

Penha, Jeronimo Costa, 1984-
Ferramentas para geração automática de aceleradores em plataformas heterogêneas de alto desempenho com FPGA / Jeronimo Costa Penha. – Viçosa, MG, 2018.
xii, 92f. : il. (algumas color.) ; 29 cm.

Inclui apêndice.
Orientador: Ricardo dos Santos Ferreira.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 74-84.

1. Computação de alto desempenho. 2. Fluxo de dados (Computadores) - Projetos. 3. Circuitos integrados. 4. Circuitos lógicos. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22. ed. 004.36

JERONIMO COSTA PENHA

FERRAMENTAS PARA GERAÇÃO AUTOMÁTICA DE ACELERADORES EM PLATAFORMAS HETEROGÊNEAS DE ALTO DESEMPENHO COM FPGA.

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 27 de julho de 2018.

Fernando Magno Quintão Pereira

Janison Rodrigues de Carvalho

Edward David Moreno Ordonez

Ricardo dos Santos Ferreira
(Orientador)

Dedico especialmente à minha esposa Amanda pelo amor, paciência e carinho em todos os momentos dessa etapa em nossas vidas. Você esteve sempre ao meu lado, mesmo nos momentos mais difíceis.

Dedico aos meus pais, Jeronimo e Luzia, e aos meus irmãos, Juliano e Jéssica, pelo amor e por sempre me incentivarem na busca pelos meus objetivos.

Agradecimentos

Agradeço especialmente à minha esposa Amanda por todo apoio, amor, compreensão e paciência em todo este período.

Aos meus pais, por terem me criado com amor e atenção, e por sempre terem me incentivado em minhas escolhas.

Aos meus irmãos, Juliano e Jéssica, pela confiança e amizade.

À Universidade Federal de Viçosa, pela grande oportunidade de participar deste programa de pós graduação.

À inestimável ajuda do meu orientador Ricardo, que auxiliou de forma ativa em todas as etapas de meu trabalho.

Ao Rubens O. M. Filho que se tornou um amigão pra todos os momentos e parte importante por eu ter chegado aqui.

A todos os meus colegas de mestrado pela amizade e companheirismo.

Aos professores do DPI pela paciência, comprometimento e amizade.

Ao Altino Alves pelo apoio durante este tempo.

A todos que contribuíram de diversas formas para que este momento fosse possível.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

PENHA, Jeronimo Costa, Universidade Federal de Viçosa, julho de 2018. **Ferramentas para geração automática de aceleradores em plataformas heterogêneas de alto desempenho com FPGA**. Orientador: Ricardo Santos Ferreira.

A era *Big Data* e a estagnação da evolução dos processadores devido aos desafios de eficiência energética, motiva o surgimento de novas arquiteturas de domínios específicos buscando aumentar o desempenho. Juntamente ao desenvolvimento das placas de vídeo (*Graphics Processing Unit - GPU*), arquiteturas híbridas CPU-FPGA tem surgido e demonstrado potencial para utilização em computação de alto desempenho com eficiência energética para processamento de fluxo de dados. Apesar de serem promissores, os FPGAs (*Field-Programmable Gate Array*) ainda são pouco utilizados devido a baixa velocidade de chaveamento (*clock*), complexidade de desenvolvimento e taxas de transferência de dados limitadas. Ferramentas, como OpenSPL e OpenCL, têm sido desenvolvidas com o objetivo de reduzir a complexidade de desenvolvimento. Porém, os FPGAs ainda requerem conhecimento de arquitetura de computadores e hardware para o desenvolvimento de soluções. Este trabalho tem como objetivo a redução da complexidade no desenvolvimento de algoritmos baseados em fluxo de dados para a execução em arquiteturas heterogêneas com processadores e FPGAs. São propostas duas ferramentas para desenvolvimento e execução de algoritmos. A primeira, "ADD", é uma ferramenta de desenvolvimento de algoritmos com grafos de fluxo de dados que auxilia no projeto, simulação, geração e execução dos arquiteturas desenvolvidas. A segunda ferramenta é um gerador automático específico para um domínio de classificação não supervisionada de dados, implementada com o algoritmo *K-means* capaz de gerar soluções em FPGA. A ferramenta gera também o *K-means* para execução em GPU. Duas plataformas CPU-FPGA foram utilizadas para a validação das ferramentas. Ambas as plataformas foram desenvolvidas pela Altera/Intel, denominadas HARP 1 e HARP 2. Os

resultados obtidos pelas ferramentas desenvolvidas foram promissores e mostram o potencial da utilização de aceleradores com FPGA nos quesitos tempos de execução e eficiência energética.

Abstract

PENHA, Jeronimo Costa, Universidade Federal de Viçosa, July, 2018. **Tools for automatic generation of accelerators in heterogeneous platforms of high performance with FPGA**. Adviser: Ricardo Santos Ferreira.

The Big Data generation and the stagnation of the processor's evolution, caused by the energy efficiency challenges, motivated the rise of new domain-specific architectures to increase performance. With the advance and development of better graphics processing units (GPUs), hybrid CPU-FPGA architectures arose and proved the potential to be used in high-performance computing with energy efficiency in data flow processing. Although promising, the FPGAs (Field-Programmable Gate Array) solutions still little used because they have a low clock speed, a high complexity of development and a limited data transfer rates. Tools, such as OpenSPL and OpenCL, have been developed with the purpose to reduce the complexity of development. However, FPGAs still require knowledge of computer architecture and hardware to solution development. This work aims to reduce complexity in the development of algorithms based on data flow for execution in heterogeneous architectures with processors and FPGAs. Two tools are proposed for developing and algorithm's execution. The first one, called "ADD", is an algorithm development tool with data flow graphs that assists in the design, simulation, generation and execution of the developed architectures. The second tool is a specific automatic generator for an unsupervised data classification domain capable of generating FPGA solutions and implemented with K-means algorithm. The tool also generates K-means for GPU execution. Those tools can be used as a didatic resourse for the training of programmers for high performance computing. Two CPU-FPGA platforms were used for the validation of the tools. Both platforms were developed by Altera/Intel, called HARP v1 and HARP v2. The results achieved by the tools developed were promising and show the potential of the accelerators with FPGA

can achieve considering execution times and energy efficiency.

Capítulo 1

Introdução

Ao considerar a evolução do desempenho dos processadores ao longo das últimas décadas, pode-se observar que o desempenho encontra-se estagnado conforme ilustrado na figura 1.1. Este fato foi destacado pelos ganhadores do prêmio *Turing* de 2017, David Patterson e John Hennessy [1]. A medida de desempenho dos processadores é relativa ao processador VAX11-780. Neste contexto, a exploração do paralelismo e a eficiência energética são dois grandes desafios da computação de alto desempenho para melhorar o estado da arte dos processadores atuais.

A diminuição do tamanho dos transistores e sua integração em um único substrato (circuitos integrados) possibilitou vários avanços no período de 1970 à 2010. Conforme previsto pela Lei de Moore [2], a densidade de integração dobrou a cada dezoito meses. Com mais transistores foi possível desenvolver os processadores CISC (*Complex Instruction Set Computer*) que proporcionaram uma taxa de aumento de desempenho de 22% ao ano. Na fase seguinte, uma evolução mais significativa ocorreu com os processadores RISC (*Reduced Instruction Set Computer*) os quais, além dos benefícios diretos da lei de Moore (maior densidade e frequência de chaveamento pelo fato dos transistores possuírem menor dimensão), obtiveram desempenho adicional devido às técnicas de exploração do paralelismo em nível de instrução (ILP - *Instruction level parallelism*), no qual pode-se destacar a utilização de *pipeline* (paralelismo temporal), memórias caches (localidade temporal e espacial de dados), predição de desvios e técnicas de compilação como otimização de laços de computação intensiva [3].

O aumento de desempenho na fase RISC chegou a 52% ao ano. Uma vez que o ganho de um monoprocessador foi saturado, a solução adotada, para utilizar a disponibilidade de mais transistores, foi o emprego de mais núcleos de processamento por circuito integrado: os processadores com múltiplos núcleos (*multicores*). Porém,

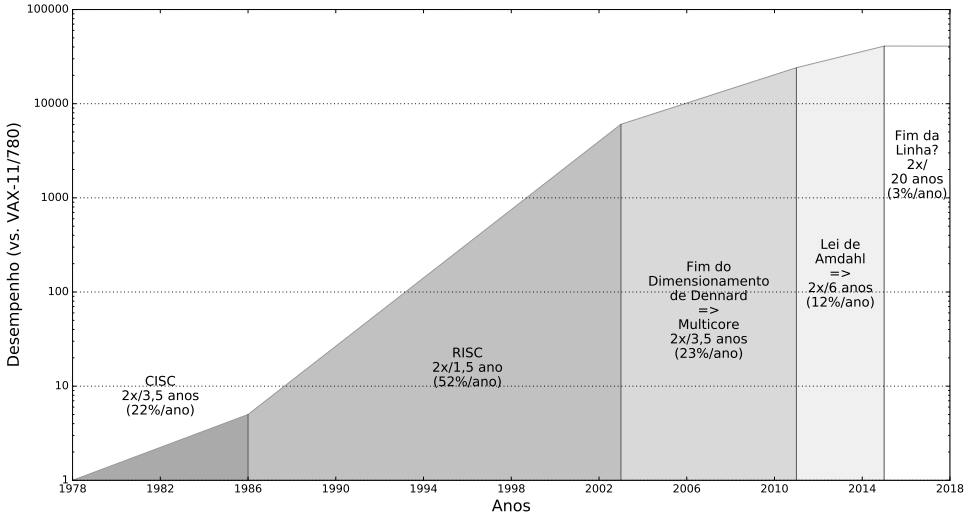


Figura 1.1: Quarenta anos de evolução do desempenho dos processadores.

Fonte: [4, 1]

devido aos problemas de dissipação de potência e consumo de energia, o projeto de processadores teve que ser repensado. Segundo a lei de Dennard sobre escalabilidade dos transistores [5], a redução das dimensões dos transistores geram uma redução na tensão e corrente necessárias para o seu funcionamento, o que possibilita a utilização de transistores menores e mais rápidos sem o aumento no consumo de energia. Entretanto, a escalabilidade de Dennard falhou pois ignorou a corrente de fuga (corrente reversa) e o fato da existência de um valor mínimo para a tensão de limiar para o correto funcionamento dos transistores. Estes fatores geraram um aumento na dissipação de potência em altas frequências. Como consequência, a frequência dos processadores estagnou em 4 Ghz em 2006 ao atingir a barreira de potência (*power wall*). A próxima barreira é o limite de exploração do paralelismo determinado pela lei de Amdahl [6]. Mesmo que se tenha N processadores, se apenas parte do código é paralelizável na arquitetura com múltiplos núcleos, o ganho será limitado. Para que seja possível a continuação do aumento de desempenho, outros modelos de arquiteturas vem sendo implementados como as GPU, os modelos sistólicos e os FPGAs que podem explorar paralelismo em domínios específicos de aplicação com o intuito de trabalhar em conjunto com os multiprocessadores de uso geral.

As Unidades de Processamento Gráfico (GPU - *Graphics Processing Unit*) [7] seguem o modelo SIMD (*Single Instruction Multiple Data*) de acordo com a classificação de Flynn [8], mais especificamente SIMT (*Single Instruction Multiple*

Thread). As GPUs possuem milhares de unidades funcionais (ALU de inteiros e ponto flutuante e unidades de Load/Store) que executam um grande número de *threads*, na ordem de milhares a milhões, o que esconde a latência das operações de memória. As operações com a memória podem demorar de 400 à 800 ciclos de relógio. As GPU escondem também a latência das unidades funcionais no paralelismo em nível de instrução, o que elimina a necessidade da execução fora de ordem, ao trocar *Threads* quando ocorrem situações de dependência de dados no nível de registradores. Recentemente, a geração de GPU da Nvidia com a arquitetura VOLTA [9], incorporou unidades de multiplicação e soma de matrizes com um modelo sistólico (*Tensor Flow*). Cada unidade é capaz de realizar a operação $A * B + C$ em um único ciclo de relógio, onde A, B e C são matrizes de ponto flutuante de 16 bits com dimensões 4x4. A GPU Volta V100 possui 640 unidades sistólicas com o desempenho de mais de 100 Teraflops com operandos de 16 bits.

GPUs são adequadas para o modelo SIMD de paralelismo. São capazes de processar de forma eficiente milhares de dados simultaneamente. Entretanto, para que seja extraída a máxima eficiência de um código implementado em GPU, é necessário que a mesma sequência de instruções seja executada sobre todos os dados. Já, os FPGAs podem ser entendidos como um modelo complementar aos processadores. Um processador fixa uma pequena quantidade de dados nos registradores ou caches para executar instruções sobre este conjunto de dados, enquanto que os FPGAs fixam um conjunto de instruções na sua estrutura de hardware e os dados passarem por elas. Segundo, Doug Burger do projeto Catapult [10] da Microsoft, este modelo pode ser denominado como computação estrutural (ou espacial), onde um grafo de operações é posicionado no hardware e um fluxo de dados passa através dele.

Como destacado na aula do prêmio Turing de 2017 [3] pelos pesquisadores David Patterson e John Hennessy, uma alternativa para aceleração do desempenho é o emprego de arquiteturas de domínio específico com linguagens de domínio específico. Em função do anunciado fim da lei de Moore, o foco em domínios específicos é uma alternativa para explorar o paralelismo com uma visão integrada de software e hardware. Pode-se dizer que é uma visão ampliada que considera o domínio de aplicação, a linguagem, o compilador e o hardware.

Em 2018, a tendência da utilização de domínios específicos foi reforçada nas palestras convidadas na conferência de arquitetura de computadores, a 45th International Symposium on Computer Architecture (ISCA), onde a palestrante Kim Hazelwood do *Facebook* destacou a importância de configurações especializadas para os serviços nos *Datacenters* aos invés dos modelos de uso geral. O palestrante Kunle

Olukotun da Universidade de Stanford destacou a necessidade do uso de aceleradores reconfiguráveis para os algoritmos de aprendizado de máquina, juntamente com o uso de técnicas avançadas de compilação para fornecer eficiência e abstração que os programadores possam trabalhar em um alto nível com fluxo de dados.

Neste trabalho de mestrado foram desenvolvidas ferramentas para oferecer suporte de aceleração de desempenho de aplicações a partir da utilização de fluxo de dados e domínios específicos com implementações em sistemas heterogêneos com processadores e FPGAs.

O texto de introdução da dissertação está organizado da seguinte forma: Primeiro são apresentados os objetivos do trabalho em 1.1. Posteriormente na seção 1.2 são descritas as principais características dos FPGAs. Na seção 1.3 o modelo de computação com grafo de fluxo de dados e suas implementações são explanados. Neste trabalho são propostas ferramentas baseadas nos grafos de fluxo de dados para implementação com alto desempenho em FPGA. Na seção 1.4 é descrita a estrutura da dissertação que foi formatada no modelo de coletânea de artigos. Nesta seção são destacados os três artigos que compõem esta dissertação e suas contribuições. Os capítulos 2, 3 e 4 apresentam os três artigos que compõem a dissertação. Finalmente, no capítulo 5, são apresentadas as principais conclusões deste trabalho. Ademais, um quarto artigo é apresentado no Apêndice A aplicado ao ensino de processadores RISC que utiliza FPGAs de forma didática e que segue a sequência do livro dos ganhadores do prêmio *Turing* de 2017.

1.1 Objetivos

Esta dissertação tem como objetivo reduzir a complexidade no desenvolvimento de algoritmos baseados em fluxo de dados para execução em arquiteturas heterogêneas com processadores e FPGAs. Através do desenvolvimento de ferramentas, os usuários poderão ter como ponto de partida grafos de fluxo de dados que de forma automatizada são transformados em implementações que irão executar em arquiteturas heterogêneas. Outro objetivo é minimizar o esforço dos programadores da comunidade de *software*, de modo que pouco ou nenhum conhecimento de linguagens de descrição de hardware seja necessário para descrever os aceleradores que irão executar no FPGA.

1.2 FPGA

Os *Field-Programmable Gate Array* (FPGAs) são circuitos integrados que permitem a sua reconfiguração interna após a fabricação [11] e podem ser utilizados para a aceleração de uma gama de algoritmos através da implementação em hardware com circuitos lógicos, interconexões e memória gerando resultados de desempenho comparáveis aos obtidos por processadores modernos [12]. Os FPGAs surgiram na década de 80 com o intuito de substituir as soluções implementadas nos *Programmable Array Logic* (PAL). Nos anos 90, os FPGAs foram utilizados em emulações de circuitos lógicos e prototipagem de projetos de circuitos digitais e continuam a ser utilizados até hoje, como no simulador FireSim da Universidade de Berkeley [13] com o qual é possível a utilização de mais de 1000 FPGAs simultaneamente e é capaz de simular circuitos complexos com processadores com milhões de componentes com alto desempenho [14].

Os FPGAs podem ser personalizados para tarefas específicas de processamento tais como como o processamento de vídeo, visão e redes, que muitas vezes demandam muito processamento [15]. Atualmente os *data centers* tem como principal desafio a eficiência energética. As alternativas adotadas para minimizar este problema são as arquiteturas heterogêneas para aceleração formadas por GPUs (*Graphical Processing Units*), FPGAs (*Field Programmable Gate Arrays*) e ASICs (*Application Specific Integrated Circuits*). Dentre as várias opções de aceleradores apresentadas, os FPGAs são considerados uma direção promissora a ser tomada por oferecer flexibilidade e eficiência energética [16].

Apesar da sua flexibilidade, os FPGAs ainda são pouco utilizados em aceleradores [17]. A baixa utilização se deve a várias razões, algumas das mais importantes são a velocidade de chaveamento e a complexidade da programação para desenvolvimento e validação de um projeto, segundo [12].

Quanto velocidade de chaveamento ou frequência de relógio, os FPGAs suportam frequência de relógio da ordem de 200-400 MHz, ou seja, de 10 a 20 vezes mais lentos que os processadores e as GPUs. O principal motivo é a possibilidade de reconfiguração de seus componentes do FPGA após a fabricação. Além disso, não possuem circuitos eficientes para a implementação de recursividade, estruturas baseadas em ponteiros dentre outras [12]. Já a complexidade dos FPGAs decorre da flexibilidade oferecida pela programação das interconexões e dos blocos lógicos, além das unidades funcionais (DSP) e blocos de memória RAM disponíveis nos FPGAs atuais. Em geral, os FPGA são programados através de linguagens de descrição de *hardware* (HDL) como VHDL e Verilog, que exigem conhecimento de circuitos

digitais [12]. Os compiladores para FPGA denominados pelo termo ferramentas de síntese lógica são capazes de transformar as descrições em alto nível ou no nível de linguagem de hardware em blocos lógicos do FPGA. Entretanto, o tempo de síntese é bem elevado em comparação com o tempo de compilação em software, podendo consumir horas de processamento.

Em contrapartida, os FPGAs são capazes de realizar tarefas massivamente paralelizadas, espacialmente e temporalmente, pela implementação em hardware das operações com baixo consumo de energia [17], o que os coloca em uma posição promissora para soluções com eficiência energética. Um exemplo é a solução criada pela Microsoft chamada "Catapult" onde a associação de FPGAs (como aceleradores) e processadores proveu um aumento de duas vezes na taxa de resposta da ferramenta de pesquisa Bing, o que reduziu em duas vezes a quantidade de servidores com um aumento de consumo de energia de apenas 10% para cada conjunto processador/FPGA. Os fabricantes de FPGAs como Xilinx e Intel-Altera vem desenvolvendo várias formas de integração entre processadores e FPGA através da interface PCIe, sendo um dos mais recentes o *Intel-Altera Heterogeneous Architecture Research Platform* (HARP) voltado para *data centers* [16], que é uma das plataformas de hardware utilizadas nesta dissertação.

Um FPGA pode ser definido como um conjunto bidimensional de unidades lógicas reprogramáveis interligados por uma rede também reprogramável. As unidades lógicas implementam qualquer função lógica programável de até n entradas. A quantidade de entradas das unidades lógicas pode variar de três a seis entradas. Nos FPGAs atuais, estas unidades lógicas, agora denominadas de CLBs (*Configurable Logic Block*), são mais complexas, o que permite a execução de funções mais elaboradas [18], que incluem multiplexadores, *flip flops* e interconexões em cascata.

A conexão entre os CLBs é feita por barramentos em linhas e colunas de que são interconectados por roteadores (*switches* - SW) posicionados nos cruzamentos linha x coluna em uma estrutura bi-dimensional em ilha. A interface externa é composta por pinos reconfiguráveis como entradas ou saídas (IOB). A figura 1.2 exibe a estrutura interna básicas dos FPGAs onde CLBs (indicados por Bloco Lógico) e barramentos de conexão podem ser observados [18].

Os FPGAs atuais incorporam estruturas de grão mais grosso como blocos de memória RAM com capacidades que variam entre dois a vinte Kilobytes com palavras de tamanhos que variam de 16 a 36 bits. Estes blocos de memória ficam distribuídos entre os CLBs. Porém é uma quantidade pequena de memória se comparada as caches dos processadores e os bancos de registradores e outras estruturas de memória disponíveis nas GPUs. Além dos blocos de memória RAM, os FPGAs

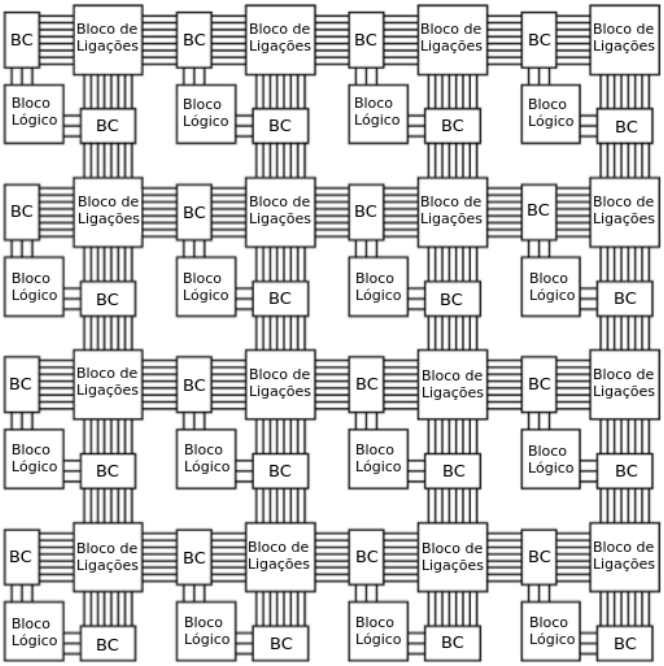


Figura 1.2: Modelo básico da constituição interna dos FPGAs.

Fonte: [18].

incorporam os blocos de processamento digital de sinais (DSP) e unidades lógico-aritméticas (ALU) capazes de executar funções lógicas e aritméticas como soma, subtração, multiplicação. Os modelos atuais de FPGA da Intel para computação de alto desempenho, Arria 10 e Stratix 10 possuem 1518 e 6760 DSPs respectivamente [19, 20]. Caso mais blocos de memória RAM sejam necessários, os próprios CLBs podem ser utilizados como módulos de memória de poucos bits. Algum fabricantes incrementam os circuitos integrados que FPGA com processadores. Como exemplo, tem-se o Zynq-7000 da fabricante Xilinx traz integrado um processador ARM Cortex A9 *dual core* [18].

Várias empresas, como Microsoft, Intel-Altera, Xilinx, IBM e Convey, disponibilizam atualmente equipamentos baseados em FPGA para aceleração. Segundo [16], estes equipamentos podem ser classificados de acordo com a tabela 1.1.

A diferença básica está na forma de comunicação entre o processador e o FPGA. A Alpha Data Board e o Microsoft Catapult possuem sua comunicação de dados via barramento PCI-e e possuem memória privada como uma GPU. A Alpha Data FPGA Board foi desenvolvida com um FPGA Xilinx Fabric [21] o qual realiza a integração com o ambiente de desenvolvimento Xilinx SDAccel, o que permite a criação de projetos de aceleradores e dá suporte a linguagens de alto nível como

Tabela 1.1: Classificação das plataformas modernas CPU-FPGA.

	Memória Privada	**Memória Compartilhada**
Periférico PCIe Interconectado	Alpha Data Board Microsoft Catapult	IBM CAPI
Interconectado ao Processador	N/A	Intel/Altera HARP (QPI + PCIe) Convey HC-1 (FSB)

Fonte: [16].

C/C++ e OpenCL. O acelerador Catapult foi criado pela empresa Microsoft com o intuito de reduzir tempo de resposta da ferramenta de pesquisas "BING" na qual a associação de FPGAs (como aceleradores) e processadores proveu um aumento na taxa de resposta do *ranking* da ferramenta e uma redução no consumo de energia. Já as soluções da IBM e Intel são baseadas em memória compartilhada. A solução da IBM, denominada *Coherent Accelerators Processos Interface* (CAPI), é conectada pelo barramento PCIe e utiliza memória compartilhada com o processador [22].

Outras soluções têm utilizado uma nova classe de conexão com o processador como o *Front Side Bus* (FSB) e o novo *QuickPath Interconnect* (QPI). Estes permitem acesso a memória compartilhada coerente e têm sido utilizados pela *Convey Machine* e pelo *Intel-Altera Heterogeneous Architecture Research Platform* (HARP), respectivamente, e são voltados mais especificamente para *data centers* [16].

1.3 Arquitetura com Fluxo de dados

As aplicações que envolvem o processamento de sequências (*streams*) vem crescendo em diversas áreas como vídeo, áudio, busca de dados na internet, redes de computadores, bioinformática, entre outras. Nas arquiteturas tradicionais, baseadas no modelo de Von Neumann (paradigma processador-memória), a busca centralizada das instruções acaba sendo um dos gargalos do sistema, mesmo que várias unidades de cálculo estejam disponíveis. O fluxo de dados elimina os passos de busca e decodificação, uma vez que a computação fica mapeada no hardware e apenas os dados irão fluir pela estrutura. Portanto, podem reduzir o consumo de energia ao eliminar a busca e decodificação, além de aumentar o desempenho com a exploração do paralelismo temporal e espacial.

As arquiteturas de fluxo de dados foram apontadas como uma solução promissora na década de 70 e, posteriormente, no início dos anos 80. Porém, devido a tecnologia de implementação, o alto custo e tempo para o desenvolvimento de circuitos específicos, esta abordagem foi deixada de lado. Atualmente, com o curto tempo

de desenvolvimento proporcionado pelas arquiteturas reconfiguráveis baseadas em FPGA, as arquiteturas de fluxo de dados aparecem como uma solução escalável, de alto desempenho, baixo consumo e fácil conectividade, tornando-se uma alternativa viável e de grande potencial frente às arquiteturas superescalares.

Entretanto ainda existe uma escassez de ferramentas e linguagens para expressar e mapear os fluxos de dados. Mais relacionadas ao escopo desta dissertação pode-se citar as extensões de C/C++ para FPGA com OpenCL (*Open Computing Language*) da Intel/Altera [23] e Xilinx [24], como também a extensão de Java com OpenSPL (*Open Spatial Programming Language*) [25]. O OpenCL mapeia no fluxo de dados de forma transparente. Porém, a eficiência do código depende da forma como o fluxo de dados e as operações com a memória são mapeadas. Além disso, o tempo de compilação é elevado (de minutos a horas de processamento)

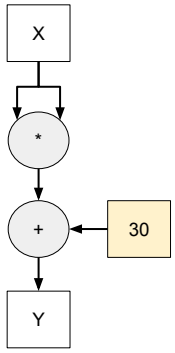
O OpenSPL é uma linguagem para o desenvolvimento de algoritmos com exploração de paralelismo espacial para gerar aceleradores em FPGA. Tem como objetivo a unificação das áreas de desenvolvimento de *software* e *hardware* através de um modelo simples de programação que permita aos desenvolvedores criarem sistemas seguindo os ciclos de vida tradicionais de projetos de *softwares* com código baseado na linguagem JAVA, o que reduz a curva de aprendizado. Tem operadores e objetos para manipular *streams*. O hardware é gerado em VHDL, que precisa de uma ferramenta de síntese para geração *bitstream* do FPGA. Os dados são transferidos por *streams* e os resultados são colhidos de volta na memória de forma transparente ao programador. Na Figura 1.3 mostra um exemplo de código escrito em OpenSPL (a) para a execução em FPGA, e em (b), a representação gráfica do fluxo de dados gerado. É importante salientar que o fluxo de dados é gerado automaticamente a partir do código escrito e sua ilustração é apenas para visualização [26].

Os operadores utilizados para o desenvolvimento de grafos de fluxo de dados podem ser síncronos ou assíncronos. No modo síncrono, os resultados de cada operador são entregues a cada ciclo de relógio. Caso ocorra alguma interrupção, todos os operadores são bloqueados até que o fluxo retorne sua execução.

Além da execução síncrona, em muitas aplicações é necessário indicar a existência de valores válidos, falta de dados, término da execução, dentre outros. Estas informações podem ser inseridas através de sinais adicionais no grafo de fluxo de dados. O controle pode também ser feito através de operadores especiais para a ativação e direcionamento do fluxo de dados processados de acordo com decisões tomadas durante a execução tanto para grafos síncronos quanto assíncronos [27]. Os operadores assíncronos possuem um protocolo de transferência de dados com sinalização de solicitação de envio e aviso de recebimento de dados. Ou seja, nem todos

```
class Loop extends Kernel{
  Loop(){
    DFEVar x = io.input("x",dfeFloat(8,24));
    DFEVar y = x * x + 30;
    io.output("output",y,dfeFloat(8,24));
  }
}
```

(a)



(b)

Figura 1.3: Exemplo de código OpenSPL (a) e representação do fluxo de dados gerado (b).

Fonte: [26].

os operadores precisam estar ativos ao mesmo tempo. Este recurso pode ser interessante para economia de energia se os operadores inativos estiverem em modo de baixo consumo. Além disso podem evitar o uso de registradores de balanceamento de fluxo entre caminhos de comprimento diferente.

1.4 Estrutura da Dissertação

O corpo desta dissertação está estruturada em conformidade com o formato de coletânea de "artigos científicos" normalizado pelo Conselho Técnico de Pós-Graduação da Universidade Federal de Viçosa [28]. Este formato é composto pelas partes básicas: introdução geral (capítulo 1), artigos científicos (capítulos 2,3 e 4) e conclusões gerais (capítulo 5).

O capítulo 2 apresenta o artigo *ADD - Uma Ferramenta de Projeto de Aceleradores com DataFlow para Alto Desempenho* [29], publicado em 2017 na WSCAD 2017 [30]. O artigo propõe a ferramenta ADD para desenvolvimento com FPGA e grafos de fluxo de dados síncronos (*Dataflow* - DFG). O termo ADD é uma abreviação para (*Accelerator Design and Deploy*). O ADD permite a criação/edição de DFGs de forma visual ou descritiva em um formato estrutural. A principal contribuição do ADD é simplificar a simulação, o mapeamento e a execução de DFG em ambientes heterogêneos com processadores e FPGAs. A ferramenta disponibiliza uma biblioteca de operadores DFG. Além disso, novos operadores podem ser criados como extensão dos operadores disponíveis. O ADD foi codificado em Java. O DFG pode ser simulado para depuração e correções. Posteriormente, o DFG é

exportado para uma ferramenta de FPGA onde é sintetizado e gravado no FPGA. O processador pode então chamar o DFG que foi gravado no FPGA para executá-lo. O ADD é compatível com a plataforma HARP v.1 da Intel/Altera e com placas de baixo custo. A comunicação com as placas de baixo custo é feita através do modelo JTAG, enquanto que no HARP v.1 a comunicação é feita através da *Application Programming Interface* (API) AAL/QPI da Intel [31]. A síntese é feita pelo conversor desenvolvido do formato DFG para a linguagem Verilog HDL. O usuário não precisa ter conhecimentos em descrição de *hardware*. O ADD foi avaliado com DFGs de processamento de sinal onde o desempenho foi comparado com a execução sequencial em um processador Intel(R) Xeon(R) CPU E5-2680 v2 com frequência de *clock* de 2.80GHz. O trabalho foi selecionado como o melhor artigo da WSCAD 2017.

O capítulo 3 apresenta o artigo *ADD: Accelerator Design and Deploy - A Tool for FPGA High Performance Dataflow* submetido para o periódico *Concurrency and Computation Practice and Experience* (CCPE) [32] que se trata de uma versão extendida do artigo apresentado no capítulo 2, com pelo menos 50% de conteúdo novo, e submetido após convite realizado pelo corpo editorial do periódico. A primeira contribuição em relação ao artigo do capítulo 2 está relacionada às plataformas de execução dos DFGs. O gerador de código foi estendido para a nova plataforma heterogênea processador/FPGA da Intel/Altera denominado HARP v.2. A nova plataforma melhorou a taxa de transferência processador/FPGA pois, além do barramento QPI (*QuickPath Interconnect*), inclui também dois barramentos PCIe, o que possibilita taxas de transferências de dados na ordem de 27 GBps. O HARP v.2 inclui também a API OPAE [33]. Outra contribuição é a possibilidade de utilização de mais de um *buffer* de entrada/saída de dados e o uso de vários aceleradores distintos e independentes. Com relação às placas de baixo custo, além da interface JTAG, foi adicionada a interface RS232. O protocolo de comunicação dos operadores do grafo de fluxo de dados foi estendido, abrindo novas possibilidades de controle. Além dos DFGs síncronos, a nova versão do ADD trabalha com DFGs assíncronos. Novos casos de testes foram acrescentados. Ademais, a verificação de desempenho foi realizada através da comparação com a execução em GPU, processador sequencial e versões paralelas implementadas com a biblioteca OpenMP [34]. O processador utilizado foi o Intel Xeon CPU E5-2630v3 e a GPU foi uma Tesla K40c com 12GB de memória RAM.

O capítulo 4 traz como contribuições, a geração parametrizada de código para a execução do *K-means* em GPU e a geração transparente, também parametrizada, de circuitos para a plataforma CPU/FPGA HARP v.2 de alto desempenho da Intel.

O artigo foi publicado na conferência *Simpósio em Sistemas Computacionais de Alto Desempenho* (WSCAD) 2018. O código em GPU gerado se baseia na execução da classificação e no acúmulo de valores para o cálculo dos novos valores dos centroides com divisão de trabalho para cada ponto em *threads* separadas. A arquitetura do acelerador gerado para o FPGA realiza apenas a classificação dos dados. O acúmulo e o cálculo das novas dimensões dos centroides é feita pela CPU em paralelo ao FPGA. Para os experimentos, foram geradas combinações de centroides iguais a 2,4, e 16 com dimensões iguais a 2,4,8,16 e 32. Para a execução, foram utilizados uma CPU Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz com código sequencial, uma GPU Tesla K40 e o protótipo HARP v.2 com um FPGA Arria 10. Nos resultados, foi possível verificar que o tempo de execução do acelerador foi limitado da taxa de transferência de dados entre o FPGA e a CPU do protótipo utilizado, porém os resultados mostraram que um potencial de eficiência energética em comparação com a solução com GPU e que existe possibilidades de melhorias para futuras extensões do gerador em FPGA.

Além dos artigos anteriormente citados, no Apêndice A encontra-se o artigo *MIPSFPGA - Um Simulador MIPS Incremental com Validação em FPGA* [35] apresentado no WEAC no ano de 2016 e publicado no periódico Qualis B5 na área de Ciência da Computação *International Journal of Computer Architecture Education* (IJCAE) [36]. No artigo, a ferramenta didática nomeada MIPS FPGA foi apresentada como um auxílio no aprendizado da arquitetura de processadores RISC. A ferramenta possui uma interface gráfica que traz os caminhos de dados apresentados no livro *Computer organization and design: the hardware/software interface* dos autores John Hennessy and David Patterson [37]. A ferramenta possibilita a visualização e alteração dos conteúdos dos bancos de registradores e bloco de memória RAM. Permite a edição da memória de instruções com a possibilidade de carregamento de códigos salvos em disco para execução no processador MIPS de cada datapath. Além disso, a ferramenta possibilita alteração dos caminhos de dados para dar suporte a execução de mais instruções. Outra contribuição inclusa é o mapeamento e execução dos caminhos de dados em FPGA dos fabricantes Xilinx e Intel/Altera. Foram criadas interfaces de depuração com displays de acesso aos valores de partes do caminho de dados: acesso às posições do banco de registradores e do bloco de memória RAM, valor do PC (*Program Counter*) e da quantidade de clocks executados. A interface permite que o usuário escolha quantos *clocks* e quando serão executados. Como complemento ao ensino, é possível a criação de uma série de exercícios com respostas em múltipla escolha que podem ser utilizados durante a simulação.

Além dos artigos como primeiro autor, outros trabalhos foram desenvolvidos relacionados ao uso de FPGAs ao longo do mestrado. Em conjunto com o mestrando Hector Perez Baranda foi desenvolvido o artigo *Implementação de um Preditor de Desvio no MIPS 5 Estágios*: um preditor de desvio de 1 bit para o processadores MIPS inspirado no livro *Computer organization and design: the hardware/software interface*, onde a ferramenta MIPS FPGA [35] foi utilizada com um novo caminho de dados e componentes para implementação do preditor com simulação e execução em FPGA. O artigo foi apresentado na conferência WEAC no ano de 2017 e publicado no IJCAE [36].

Em parceria com os mestrandos Lucas Bragança, Fredy Alves, e Gabriel Coimbra, todos do Campus Florestal, e orientados pelos professores Ricardo Ferreira e José Nacif, foi desenvolvido o artigo *Simplifying HW/SW Integration to Deploy Multiple Accelerators for CPU-FPGA Heterogeneous Platforms* [38], uma interface de comunicação, acompanhada de uma API para execução de aplicativos escritos nas linguagens C e JAVA, para o ambiente CPU-FPGA HARP v.2 da fabricante Intel/Altera. A interface pode ser gerada de modo automático a partir de parâmetros que indicam a quantidade de aceleradores independentes e a quantidade de *buffers* de entrada e saída de dados para cada um. Toda a transferência de dados entre a memória e o acelerador é intermediada pela interface, que é responsável por gerenciar os pedidos de escrita e de leitura de dados. O gerador da interface foi desenvolvido com o auxílio da biblioteca de *hardware* Intel BBB [39] fornecida pela própria fabricante e a API foi desenvolvida juntamente com a API de *software* OPAE [33]. A interface é capaz de utilizar a máxima taxa de transferência disponível pelo ambiente com algoritmos de *loopback*. Com a criação da API para as linguagens C e JAVA, o acesso à plataforma foi simplificado. O usuário possui total controle da execução em hardware sem a necessidade de conhecimentos aprofundados sobre o funcionamento do ambiente. O artigo foi aceito para a conferência SAMOS 2018 [40].

Outro trabalho desenvolvido foi *From Java to FPGA: an Experience with the Intel HARP System* em conjunto com a Universidade Federal de Minas Gerais com o mestrando Pedro Caldeira (UFMG) orientado pelos professores Renato Ferreira e Fernando M. Q. Pereira, e com o mestrando Lucas Bragança (UFV), orientado pelo professor José Nacif do campus Florestal. Neste trabalho foi desenvolvido um compilador para aplicações Java com a biblioteca *ParallelME* [41] para execução em aceleradores com FPGA. A plataforma de hardware utilizada para a execução foi a plataforma HARP v.2 da fabricante Intel/Altera. O compilador mapeia parte do código Java com as primitivas da biblioteca *ParallelME*, gera um grafo de fluxo de

dados que é convertido para Verilog HDL, sintetizado e gravado no FPGA. O compilador também gera o código para execução da parte não mapeada no processador e da parte mapeada no acelerador gerado no FPGA do Harp v.2.

Capítulo 2

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

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.

2.1 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 executar 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 [12].

Uma alternativa é o uso de OpenCL (*Open Computing Language*) [42]. 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 [43, 44, 45]. 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, faz-se necessário o conhecimento do processo de mapeamento do código no fluxo de dados e das arquiteturas de FPGA para possíveis ajustes, compreender as informações geradas pelas ferramentas e otimizar o código.

Outra alternativa é o OpenSPL [46] que é baseado em Java. O ambiente OpenSPL permite a visualização dos grafos de fluxo de dados gerados para código 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 possíveis 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 realizar 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 2.2, 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 2.3. A seção 2.4 aborda trabalhos relacionados e, por fim, as considerações finais são apresentadas na seção 2.5.

2.2 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 [47] 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 com a utilização das 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* [12]. 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 [48, 49].

Como complemento ao 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, o gerador desenvolvido 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 encapsu-

lado por uma chamada de função. A interface de comunicação foi desenvolvida para duas plataformas de FPGA. A primeira de baixo custo com fins educacionais, utiliza a interface JTAG para a comunicação computador-FPGA e pode ser utilizada 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 [50, 51].

2.2.1 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 2.1. Na tabela pode-se observar 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 2.1 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 2.1 pode ser replicado várias vezes, para explorar 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 per-

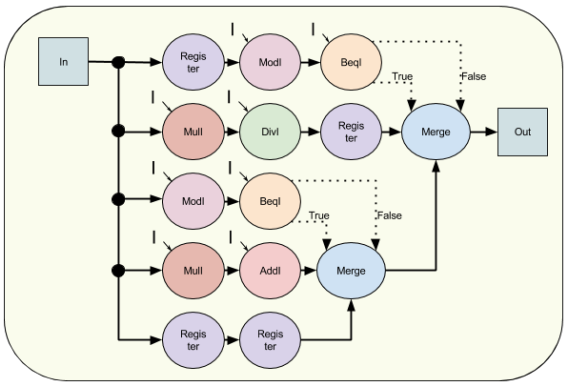
Tabela 2.1: Lista de operadores disponíveis na biblioteca do ADD.

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) / 5;
  }else if(x % 3 == 0){
    y = (x * 3) + 2;
  }else {
    y = x;
  }
}
...
```

(a)



(b)

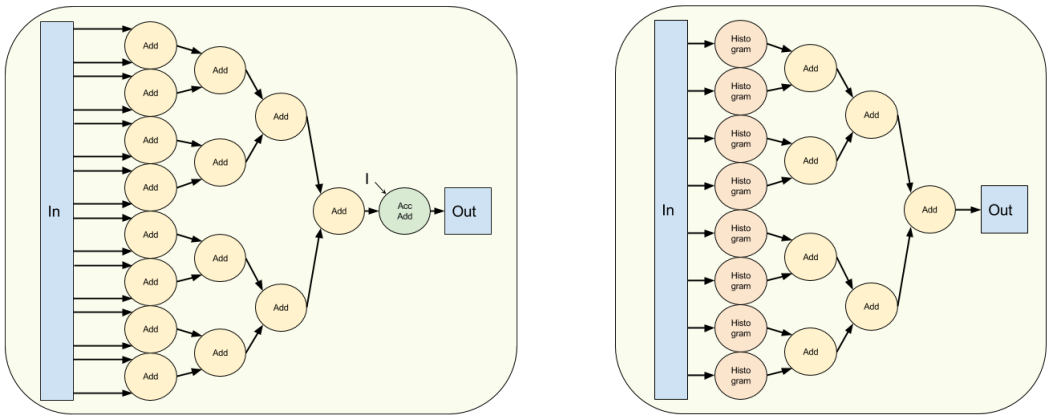
Figura 2.1: Exemplo de algoritmo com *Branches* (a) e o DFG equivalente esquemático com os operadores da biblioteca do ADD (b).

mite 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 2.2.2. 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 2.2(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 utilizados 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 2.2 (b) seguido de uma redução.



(a) (b)

Figura 2.2: Algoritmos *Reduce Add* (a) e *Histogram* (b).

2.2.2 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 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 2.3 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*.

```
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 2.3: Código que o implementa o operador MulI.

2.2.3 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 estados, sinais de relógio, *reset*, habilitação, codificação, etc. A descrição *Verilog* sintetizável é gerada com o auxílio da biblioteca Veriloggen [52]. 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.

2.2.4 Simulação, Comunicação e Execução em 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 utiliza a interface JTAG para a comunicação com os FPGAs e pode ser utilizadas 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 [47] 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 [35, 53, 54, 55, 56], 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 2.4. 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`.

O ADD pode suportar novas plataformas, como o Catapult da Microsoft [57], 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 2.5 detalha o processo de execução dos aceleradores com as duas interfaces: JTAG e XEON/FPGA com QPI.

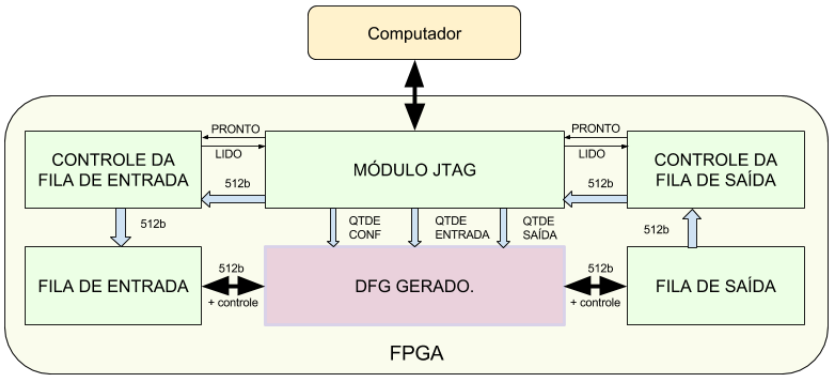
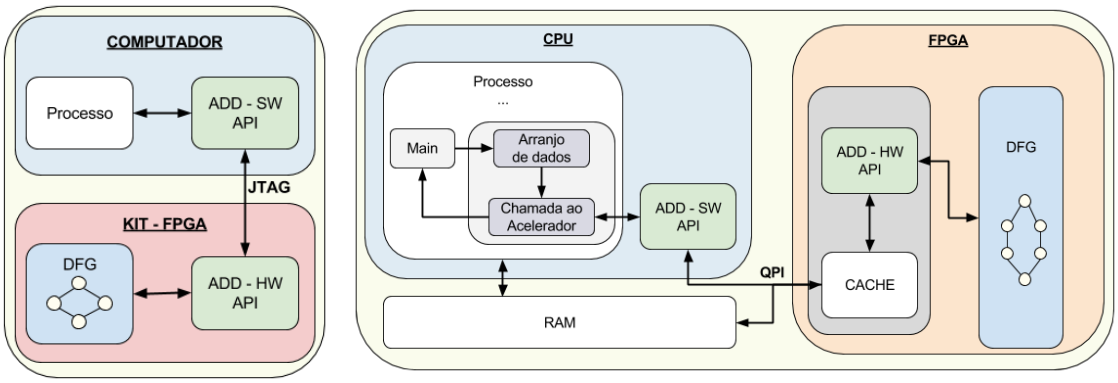


Figura 2.4: Interface para execução em FPGAs.



(a) (b)

Figura 2.5: (a) Execução de um DFG em kit de FPGA; (b) Ambiente de alto desempenho da fabricante Intel/Altera

2.2.5 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.

2.3 Experimentos e Resultados

Para a validação do funcionamento e da eficiência dos DFGs desenvolvidos com o ADD, foi utilizado um conjunto de *benchmarks* de algoritmos para processamento de sinais que podem ser vistos na Tabela 2.2. 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 observar entre parênteses o número de cópias paralelas que foram sintetizadas para cada DFG.

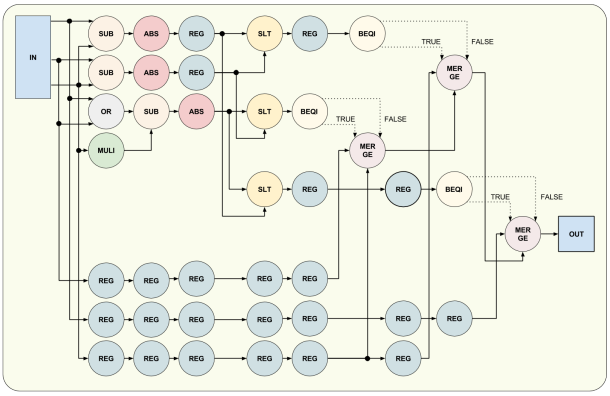
Tabela 2.2: Relação de resultados de execução dos algoritmos em CPU e FPGA.

Benchmark	**Stratix V 5SGXEA7N1F45C1**					**CPU**	**T(cpu)/ T(fpga)**
	ALMs (%)	**Registers**	**M20K**	**DSP (%)**	**T(ms)**	**T(ms)**	
Gouraud (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

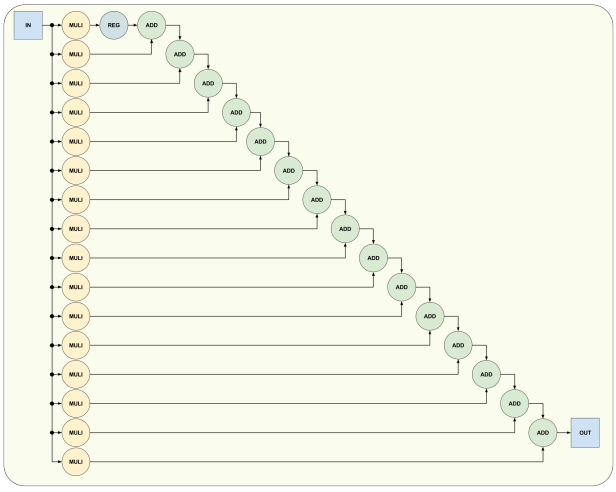
O sistema CPU-FPGA utiliza *FPGA Stratix V 5SGXEA7N1F45C1* fortemente acoplado a um processador XEON com 10-*cores*. Esta plataforma permite o acesso à memória do computador através do barramento QPI que realiza a transferência em pacotes de 64 *Bytes* a uma taxa de transferência de aproximadamente 6 GBps [58]. 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 2.2. 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 2.6.



(a)



(b)

Figura 2.6: 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 2.2 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.

2.4 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 [59]. 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 [60], 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 [61], 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 [62] 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 possibi-

lidade 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.

2.5 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.

Capítulo 3

ADD: Accelerator Design and Deploy - A Tool for FPGA High Performance Dataflow

Dataflow-based FPGA accelerators have become a promising alternative to deliver energy efficient high performance computing. However, FPGA programming is still a challenge. This paper presents Accelerator Design and Deploy (ADD), a high-level framework to specify, to simulate, and to implement dataflow accelerators for streaming applications. The framework includes an open dataflow operator library, and templates are provided to easily design new operators. The framework also provides a high-level and an accurate simulation at circuit level with short execution times. Moreover, ADD provides software and hardware APIs to simplify the integration process, extending the benefits of portability from low-cost FPGA boards to high performance datacenter FPGA platforms. Our framework supports coupling with high-level programming languages, and it has been validated on two FPGA platforms: The Intel high-performance CPU-FPGA heterogeneous computing platform and an educational FPGA kit. We show that our simple approach presents competitive performance, both in time and energy, when compared to multi-core and GPU accelerators.

3.1 Introduction

Nowadays, Field Programmable Gate Array (FPGA) accelerators have gained attention as an energy efficient platform for high performance computing. Several companies such as Intel [50, 63], Microsoft [57], and Amazon [64, 65] have built their own hardware platforms to support FPGA acceleration. Microsoft Catapult counts with thousands of FPGA boards connected by an efficient network, making

possible to perform a 250,000 machine round-trip in less than 20 microseconds [66]. Microsoft uses FPGA accelerators in a wide range of applications, e.g., network cryptography, neural networks training, Azure workload processing, and Bing web search ranking, which its throughput has been improved by 95% with only 10% more power consumption. Since 2017, Amazon Web Services cloud offers servers with tightly coupled CPU-FPGA architecture [64, 65]. In 2015, Intel has completed the USD 16.7 billion acquisition of Altera, which shows that reconfigurable platforms play an important role in the next-generation high performance computing systems.

Although reconfigurable FPGA technologies have been around since the 1980s, their utilization as a general-purpose processing platform is recent. Historically, both FPGA and ASIC developers employ Hardware Description Languages (HDLs) to implement their designs, which is usually outside the main expertise area of software developers. The lack of simple and common programming models prevent software developers from easily designing accelerators and delays a broader adoption of this technology. Recently, FPGA high-level language extensions such as OpenSPL [67] (Maxeler) and OpenCL ([23] and [68]) have been proposed to provide solutions to fill this gap. Nevertheless, these approaches still require low-level knowledge to achieve the highest energy efficiency and performance. Therefore, the developers should understand the system software and hardware layers in order to design efficient accelerators.

Another key advantage of using FPGAs in dataflow processing acceleration is the memory access efficiency. A memory access consumes two to three orders of magnitude more power than a complex addition [69]. In this direction, dataflow applications can save power by enabling computation with minimal data movement.

Dataflow does not require any instruction movement, thus making possible to save power by eliminating instruction fetch and decode during every clock cycle. Although high-level approaches like OpenCL [23] and OpenSPL [67] automatically generates a dataflow to be mapped onto an FPGA, these approaches are highly sensitive to coding style in order to effectively extract parallelism [12].

In this work, our goal is to provide directions for better dataflow designs at software and hardware levels, and to gain insight on how to use dataflow mapped onto FPGAs in HPC. The paper's contributions can be summarized as follows:

1. For the high performance programming community, we provide a framework to explicitly specify a stream-based dataflow algorithm using both graphical interface or high level text description;

2. A design flow where an algorithm could be developed and validated by easily deploying at different abstraction and physical levels: high level dataflow simulation, register transfer Level (RTL) simulation, FPGA execution mapped on low cost boards, and FPGA execution in high performance cloud CPU FPGA platforms;

3. A framework to generate the software/hardware to design new dataflow algorithms and operators, and tools to be integrated in the design chain (compilers, domain specific languages DSL, etc.) by encapsulating the details through the use of simple software and hardware APIs;

4. Even using a simple dataflow programming model, the solutions provided by the proposed tool are competitive to the state-of-the-art of GPU and multicore accelerators.

This paper extends our previous work [29] which introduces ADD (Accelerator Design and Deploy), a framework to model and generate dataflow accelerators for CPU-FPGA systems. The previous work targets an Intel prototype Xeon and Stratix V FPGA platform which was discontinued. In this paper, we include a novel unified SW/HW interface, which simplifies and encapsulates a new open standard proposed by Intel, named OPAE [70] to provide support for FPGA accelerator engines. In addition, we also develop a compatible standard interface for low cost FPGA boards, which could be used by embedded software or debugging in hardware. By extended OPAE, a fast circuit level accurate simulation could be performed by using the Intel Accelerator functional unit Simulation Environment (ASE) [70]. In this work, the experiments were validated on the new Intel cloud Xeon/Arria 10 FPGA, and in low cost FPGA boards. Moreover, the new SW/HW API presented here allows multiple heterogeneous accelerators to be mapped and shared in the same FPGA to maximize the occupancy. Three new case studies are introduced to demonstrate several aspects of the dataflow model over three specific domains: data mining (KNN algorithm), bioinformatics (gene regulatory network), and cryptography (TEA algorithm). These examples add new operator design features and methodologies.

The rest of this paper is organized as follows. Section 3.2 presents the related work. Section 3.3 introduces the proposed ADD tool. Section 3.4 discusses the performance metrics and details the case studies. Finally, the experimental results and conclusions are drawn in Sections 3.5 and 3.6, respectively.

3.2 Related Work

Nowadays, the implementation of heterogeneous accelerators mainly relies on heterogeneous computing units like CPU and GPU accelerators to achieve high efficiency of data-intensive computing, where FPGA accelerators [65, 12] have a place to provide high performance and low power consumption when specific applications are running, and hence a high energy efficiency rate can be achieved. In last decade, the big data era, research centers and numerous enterprises (such as Microsoft, Intel, Amazon, Maxeler, BlueBee, Convey Computer, and Nallatech) have presented FPGA-based approaches to provide high-performance solutions [71]. Amazon launched the Amazon EC2 F1 [64, 65] to provide FPGA computing instances as a cloud service, which intends to make it easier for programmers to deploy FPGA accelerators to address the high performance data-intensive. However, the lack of high level programming tools and the software community knowledge gaps concerning the FPGA mapping process prevents a deep understanding of how the main challenges could be faced. Microsoft offers FPGA computing as a high-level service like the Bing search tool, where high performance FPGA-based datacenters execute the users queries transparently through the Catapult project [66].

Concerning the high-level languages target the HPC software community, Maxeler provides a Java-based dataflow language named OpenSPL [72], and Xilinx [68] and Intel [23] provides OpenCL with specific **pragmas** to optimize for FPGAs. However, even the Intel FPGA SDK for OpenCL Best Practices Guide [23] advices in its introduction that in order to achieve the highest performance for FPGAs, the programmer should be familiarized with details of the underlying hardware. In addition, it also recommends that to understand how the compiler optimizations map a OpenCL application into the FPGA primitive elements.

Recently, a stencil accelerator using Intel OpenCL was presented in [73], where the authors highlight two important concerns about the compiler. First, due the Partial Reconfiguration on Arria 10 FPGAs, the fitting and routing quality for OpenCL is reduced on Arria 10. Second, in their experiment, most of best-performing kernels either failed to fit or route with the default flow, or exhibited lower operating frequency compared to the flat flow (up to 100 MHz lower). Even, a simple loopback kernel where the CPU sends data to the FPGA, and receives it back without any computation, it could requires hours to compile. Therefore, even by using a high-level approach, the programmers should have knowledge in FPGA architecture and compiler optimization. Our approach provides one to one mapping between the high-level dataflow and the generated hardware. The dataflow components enable

high concurrency, and the stream components enable communication and coordination at very-low power and area overhead [74].

Recently, a high-level programming platform for streaming applications on FPGA named ST-Accel [75] was proposed and validated on the FPGA Amazon Cloud [65, 64]. A message-passing-based host/FPGA communication model is used to avoid the coherence issue of shared memory, enables the decoupled access/execute architecture to maximize the utilization of I/O devices [75].

Since the FPGA is configurable at the bit level, the price to pay for the flexibility is the compile time, and the complexity to provide efficiency to map general purpose application. One solution is to design a reconfigurable architecture at word level, the Coarse-Grained Reconfigurable Architectures (CGRAs). However, there is no commercial CGRA available in the market, and the CGRAs also suffer from long compilation times [76].

One way is to focus on application and domain-specific accelerators: Neural network [63, 77], Bayesian learning [78], bioinformatics [79], stencil computing [73, 80], energy-efficient accelerators for graph analytics algorithms [81] , and irregular applications mapping [82]. Another way is to focus on Domain-Specific Language (DSL) which aims representing parallelism in stream-based applications, like SPar based on C++ [83].

Our work differs from the previous studies in two important aspects. First, we propose to explicitly specify the accelerator kernel as a dataflow graph, and to easily add accelerator function calls in the host code, and the data transfers. Second, our tool can be used to explore a variety of dataflow operators and to create new ones to provide support for domain specific applications. As already mentioned, the details concerning the FPGA synchronization, tools, generation, and communication are transparent for the users. Finally, the ADD tool is complementary to the existing strategies, and it can work together to find a way to bear a more diverse and inclusive design environment for FPGA accelerators and dataflow modeling, which should include approaches like OpenCL, OpenSPL, ST-Accel, and the explicit dataflow tools like ADD.

3.3 Accelerator Design and Deploy - The ADD Framework

In this section, we use simple dataflow examples to introduce the ADD framework, motivating the need of tools to easily manage streaming algorithms providing direct

visualization of both spatial and temporal parallelism. ADD offers a library of dataflow components and a graphical editor to design algorithms. By using this graphical interface, ADD stimulates the developers to be innovative, thinking 'out of the box'.

Typically-accelerated workloads have common streaming computing characteristics, which leads to very efficient hardware implementations through the exploitation of concurrency [84]. First, in section 3.3.1 we introduce the dataflow paradigm by showing examples of branching structures with arithmetic operations and *map-and-reduce* applications. Section 3.3.2 presents our approach to automatically translate the dataflow onto an FPGA design, also providing an accurate simulation with Intel ASE tool [70]. Finally, we present the FPGA compilation and execution steps in section 3.3.3.

3.3.1 High-level Dataflow Modeling

The Table 3.1 presents the 51 operators available in the current version of ADD library. More details can be found in the ADD technical documentation [85]. For easy of explanation, Figure 3.1(a) shows a simple piece of code to illustrate the paradigm shift in the programming model. First, the accelerator algorithm should be described as a dataflow, as depicted in Figure 3.1(b).

A graphical/text description formats are provided to specify the dataflow. In addition to the arithmetic operations, this example uses control flow operators to implement the nested if-else structure. The dataflow evaluates all expressions and conditional branches in parallel, processing the input stream in a pipeline fashion. Hence, this approach maximizes the throughput to one result per clock cycle. However, since the operators are synchronous, the stream branches should be balanced by inserting registers. In order to implement control flow, the multiplexer operation (Mux) should be used together with each branch (Beq). It is also important to mention that in dataflow computing there is no need to perform fetch/decode operations, since the data and control are processed in-place, thus decreasing the power required to perform the computations.

For *map-and-reduce* based applications, the ADD modeling is straightforward, as shown in Figure 3.1(c). We model the *map-and-reduce* application onto a dataflow by using replication (spatial parallelism) for the mapping, and a pipelined tree of operators (temporal parallelism) for the reducing. In this example, the dataflow receives 16 elements per clock. At each clock cycle, it is possible to perform all 15 addition operations in parallel. Moreover, the accumulator operation (Acc) executes

Tabela 3.1: A sample of ADD operators library: logic/arithmetics, In/Out, Control flow, Reduction (accumulators).

Class	Name	Operation(s)
Arithmetic	Abs	$\|a\|$
	Add	$a + b$
	Div	a/b
	Mod	$a\%b$
	Mul	$a * b$
	Sub	$a - b$
	AddI	$a + Const$
	DivI	$a/Const$
	ModI	$a\%Const$
	MulI	$a * Const$
	SubI	$a - Const$
Logical	And	$a\&b$
	Not	$!a$
	Or	$a\|b$
	Xor	$a \oplus b$
	AndI	$a\&Const$
	OrI	$a\|Const$
	XorI	$a \oplus Const$
Comparison	Max	$Max(a, b)$
	Min	$Min(a, b)$
	Slt	$(a < b)?1 : 0$
	MaxI	$Max(a, Const)$
	MinI	$Max(a, Const)$
Shift	SltI	$(a < Const)?1 : 0$
	Shl	$a << b$
	Shr	$a >> b$
	ShlI	$a << Const$
	ShrI	$a >> Const$

Class	Name	Operation(s)
Accumulation	AccAdd	$\sum(a)$
	AccMax	$Max(a)$
	AccMin	$Min(a)$
	AccMul	$\prod_{Stream} a$
	Register	a
Control Flow	Beq	$(a = b)?1 : 0$
	Bne	$(a \neq b)?1 : 0$
	BeqI	$(a = Const)?1 : 0$
	BneI	$(a \neq Const)?1 : 0$
	Merge	First-in(a, b)
	Mux	$(Ctrl)?a : b$
	Demux	$out_0 = (Ctrl)?a : -\|out_1 = (!Ctrl)?a : -$
	FGate	$(!Ctrl)?a$
	TGate	$(Ctrl)?a$
Async Flow	CpBranch	$out_0 = a \mid out_1 = a$
	CpData	$out_0 = a \mid out_1 = a$
Stream	In	$Stream$
	Out	$Stream$
Predication	KnnC	$(a > v)?v,v = a : a$
	KnnQ	$(Ctrl(0,1,2))?v : v,v = a : v$
Special Function	Queue	$Enqueue \mid Dequeue$
	Histogram	$h[*] = 0 \mid h[a]{+}{+} \mid scan(h)$
Specific App	GRN	$Custom \quad Function$

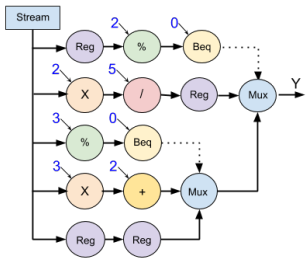
in store-and-forward fashion to sent the resulting value to the output.

Therefore, the accumulator operates in a synchronous mode based on a local store-and-forwarding protocol. Once developers model the application at the dataflow level, there is no need to implement control signals since they are embedded into ADD components. We depict a simple example of *map-and-reduce* histogram computation in Figure 3.1(d). The mapping step uses histogram operators. This operator has a local memory with N buckets to store the data locally, like scratchpad memories. At the end of the stream, each histogram scans the local memory, and then forwards the data to the reduction tree. These three examples are useful to introduce the ADD high-level dataflow modeling. Due the lack of space, complementary material available in [85] contains a C code for the examples depicted in Figure 3.1.

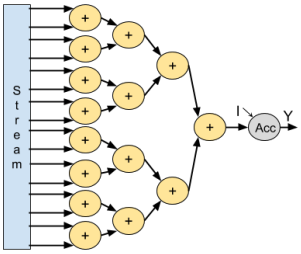
The ADD framework extends a digital circuit simulator [47, 55, 54] to support dataflow computing. The framework offers a graphical interface where the developer can simulate and debug the design. ADD also provides a simple software API to call dataflow functions in a Java/C++ code as illustrated in Figure 3.2(a). The dataflow function is then responsible to send/execute/receive data from/to the FPGA

```
...
for x in stream{
   if(x % 2 == 0){
      y = (x * 2) / 5;
   }else if(x % 3 == 0){
      y = (x * 3) + 2;
   }else{
      y = x;
   }
}
...
```

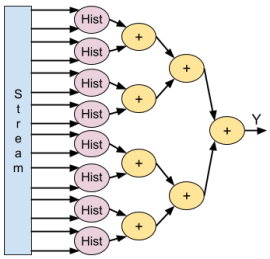
(a)



(b)



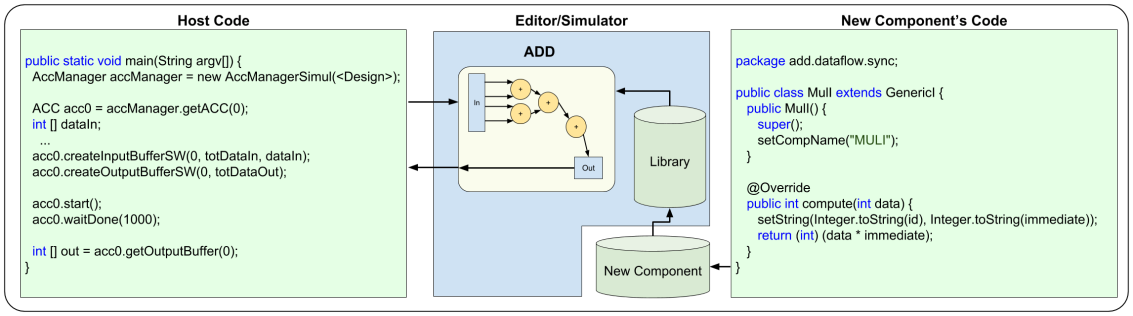
(c)



(d)

Figura 3.1: (a) A simple loop with nested branches; (b) Dataflow for the nested branches; (c) Reduction dataflow with accumulator operator; (d) Map-reduce Histogram dataflow.

as shown in Figure 3.2(b). Moreover, the ADD framework has an open library approach, which can be easily extended by creating new operators. A new component can reuse by inheritance any ADD library item written in Java. The developer should only override the *compute* method. Figure 3.2(c) depicts an example of how to write Java code to describe the behavior of a new component that performs multiplication to a constant operator. Furthermore, the ADD framework hides all complexity of synchronized hardware design, where global clock and reset signals provide timing synchronization and initialization references. In addition, ADD encapsulates the complex software/hardware interface that provides a coherent shared memory in the Intel CPU-FPGA architecture. Complementary material available in [85] contains the Intel OPAE for the correspond code shown in Figure 3.2(a), the Verilog generated code, and the component library. The repository includes a wrapper code for the component development, and the source codes for the operators depicted in Table 3.1.



(a) (b) (c)

Figura 3.2: (a) Host code to instantiate and call the accelerator(s); (b) ADD editor/simulator and the dataflow library; (c) An example of a new operator code by using inheritance.

3.3.2 FPGA Translation and Accurate Simulation

After the high-level dataflow modelling and validation, the next step is to transparently translate the code to an FPGA-based accelerator design. Each ADD component also has a Verilog description at the register transfer level (RTL), which has already been validated in an FPGA platform. The ADD framework automatically generates a set of files for the entire design, which includes the dataflow core, the input/output streams, and the host interface. This generator is based on the Veriloggen tool [52]. Each dataflow can have an arbitrary number of n inputs and m output streams. ADD generates code to an Intel HPC CPU-FPGA platform [50], which supports memory communication at 13 GB/s using mixed QPI and PCI channels. In our experiments we have limited the number of streams to $n + m < 64$ for a given design. Our previous dataflow examples, shown in Figure 3.1(b-d), have inputs varying from 1 to 16 streams with a single output stream.

Recently, Intel has introduced the Open Programmable Acceleration Engine (OPAE) [70] to simplify the integration of FPGA accelerators with software applications and environments. The OPAE SDK also provides the Accelerator functional unit Simulation Environment (ASE). This tool allows to accurately simulate both hardware (RTL) and software (C++) accelerator components by also emulating all transactions between these components. In our methodology, the ADD hardware generator creates all FPGA RTL design files.

These files are Intel ASE compatible and also include the accelerator and our SW/HW interface. Intel ASE provides an accurate simulation in few seconds without the need of FPGA synthesis and mapping steps, which can take several hours. In our framework, these abstractions are useful for debugging and tuning,

being able to rapidly make changes, recompiling, and re-executing. The host software API depicted in Figure 3.3 is the same for both simulation levels. Furthermore, even in dataflow level, the clock and reset signals, as well as operator protocols are simulated to verify the flow synchronism. The next step is mapping the design in an FPGA platform as shown in the design flow depicted in Figure 3.3.

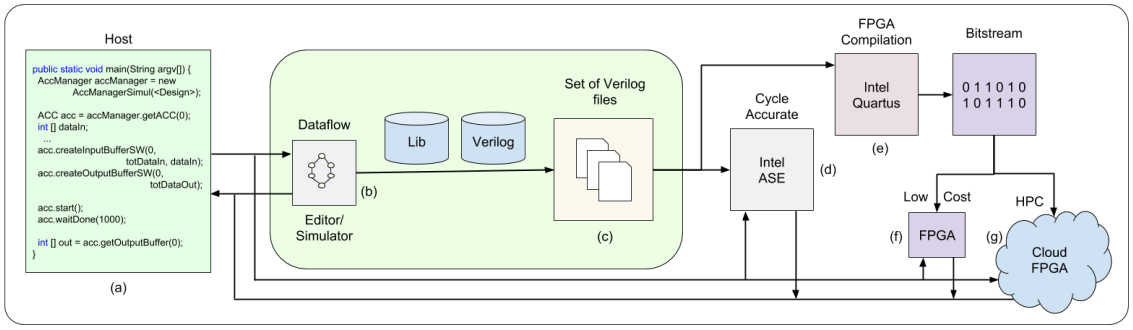


Figura 3.3: ADD Framework: (a) Software API to call the accelerator; (b) Dataflow editor; (c) Open library and high-level simulation and RTL design generation; (d) Cycle accurate simulation on Intel ASE; (e) FPGA synthesis on Quartus; (f) Low-cost FPGA platform; (g) HPC FPGA platform.

3.3.3 FPGA - Compile, Deploy and Execute

An FPGA device is itself a heterogeneous system which consists of a set of programmable logic elements (LE), programmable interconnections, embedded BRAM, and 18-bits ALUs (DSP blocks). For instance, the Intel CPU-FPGA platform [50] includes a high performance FPGA with 2,713 BRAMs (53 Mbytes in total), 1,518 DSPs (hardened single-precision floating-point units with up to 1.4 Teraflops), and 1,150,000 LEs. The main question is how to take advantage of this large amount of FPGA computing resources. Moreover, the FPGA flexibility increases the complexity to tailor the hardware for a specific application. Therefore, the FPGA has two big challenges: programmability and compile time. Our high-level approach improves the programmability by translating a dataflow in a HDL description that is later synthesized to generate an FPGA configuration bitstream. To address the FPGA compile time, we propose two targets: low-cost FPGA to map one dataflow instance, and the HPC FPGA to map multiple copies to exploit the spatial parallelism. Even if the application is already described at HDL RTL, the compilation process is very time consuming due the high complexity to perform: (1) logic

synthesis; (2) technology mapping; (3) placement and routing; (4) the bitstream generation.

In addition to multiple target FPGAs, we provide a portable SW/HW API, and the developers do not need to write any user interface code to call the accelerator at host side, and the code shown in Figure 3.3 could be reused. For a low cost FPGA, the compilation could be done in few or dozens minutes. It serves as proof of concept to execute an accelerator design in the hardware. Although, a hardware description level (HDL) code is intended for circuit synthesis as well as circuit simulation, though HDL is fully simulatable, not all constructs are synthesizable. Moreover, if one register is not correctly initialized at circuit level, the design could not work properly although the simulation was correct. Our second target option is to reach high performance, once the dataflow has been already validated in the previous levels, this final step, unfortunately still consumes hours to compile. It will be shown in next sections, in order to achieve portability and high efficiency of host/FPGA interface, we develop a multiple accelerator manager which is compatible with the two target platforms.

3.3.3.1 Low Cost FPGA Board

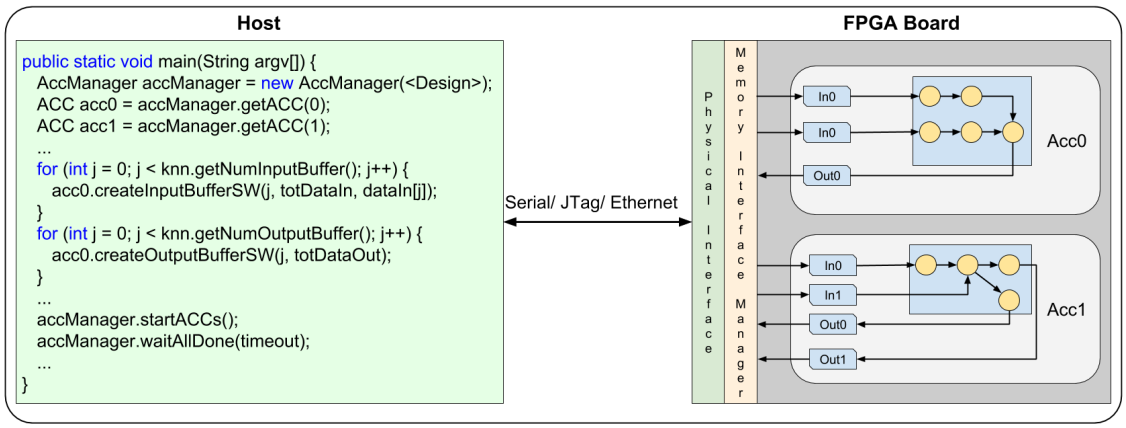


Figura 3.4: ADD and Low Cost FPGA boards. A host code example with two heterogeneous accelerators mapped in the FPGA board.

Figure 3.4 shows a block diagram of accelerator mapped onto a low cost FPGA board. At the physical level the host computer could be connected by using RS-232 serial, Serial USB, JTAG or Ethernet rj-45. Most low cost board provides all these interfaces. This approach allows a fast run time for medium size workloads as well as the execution (not simulation) in a physical FPGA. Moreover, design simulation for larger/medium workloads is prohibitive, even with an efficient simulator as the Intel

ASE/OPAE tool. In addition, low cost boards making dataflow and FPGA learning accessible for students. Further, it is an option for low cost and high performance embedded platforms and Internet-of-things devices.

As already mentioned, the ADD framework provides a SW/HW API. At the SW level, the host code should be written in Java or C++. However, other languages could easily be added like Python. Figure 3.4 depicts another code example, where there are two accelerators mapped onto the FPGA. Each accelerator unit has its own input and output FIFOs. The fifo manager controls the data communication.

3.3.3.2 Cloud FPGA Platform

Although Intel provides the Open Programmable Acceleration Engine (OPAE) [70] to abstract the hardware/software interface of the programmable accelerator resources and application software, it still requires a complex a software/hardware stack for the programming community. For instance, the developers should know implementation details of the Acceleration functional unit Link Interface (ALI), which provides functions to map the accelerator control registers into CPU memory, allocate shared buffers, configure and reset FPGA resources [63]. We propose to simplify OPAE at both sides: software API and the hardware API. Figure 3.5 shows an Intel OPAE example with one accelerator, one input stream, and one output stream. The OPAE code is more complex in comparison to the ADD solution as shown in Figure 3.3(a). The ADD provides a more intuitive API based on standard API like CUDA.

However, the compile time is still expensive, and a design will consume at least 20% of the FPGA resources, due the overhead introduced by the OPAE interface to facilitate software access and to provide transparently a coherent shared memory. The CPU-FPGA communication will first go through the FPGA cache (64KB) and then go to the host memory if a cache miss happens.

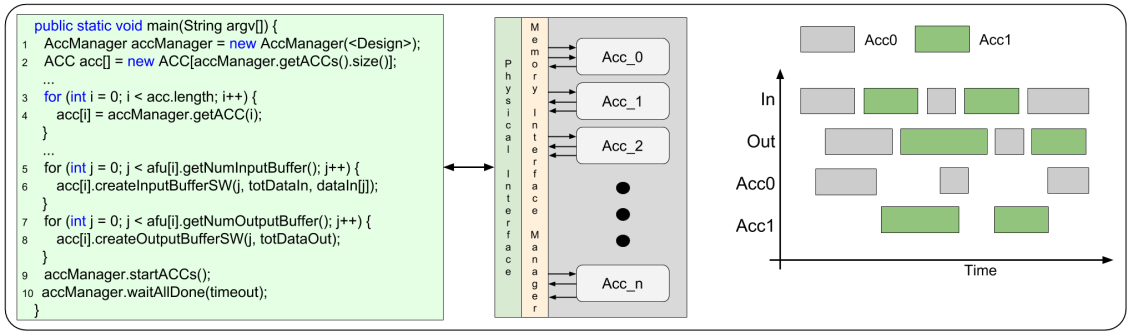
While OPAE was designed as single accelerator, we propose to integrate multiple heterogeneous accelerators in a unique design, which helps maximize FPGA occupancy. Moreover, since the performance of a single accelerator could be bound by the memory bandwidth, the multiple accelerators are mapped once, and the current accelerator can be quickly swapped without the need of the FPGA partial reconfiguration.

The software API is available in C++ or Java, and it is friendly for GPU developers like Nvidia CUDA. First, the programmer should allocate a *accelerator manager*. Then, all information regarding the accelerators available in the FPGA is sent

```
#include "opae_svc_wrapper.h"
#include "csr_mgr.h"
int main(int argc,char *argv[]){
 OPAE_SVC_WRAPPER* fpga = new OPAE_SVC_WRAPPER(<UUID>);
 assert(fpga->isOk());
 CSR_MGR* csrs = new CSR_MGR(*fpga);
 uint64_t workspace_size = (<dataIn_CL_size>*sizeof(<CacheLine>)*2)
    +sizeof(DSM);
 void* workspace = fpga->allocbuffer(workspace_size);
 assert(NULL!=workspace);
 memset(workspace,0x0,workspace_size);
 DSM* dsm = (DSM*) workspace;
 void* inputBuffer = (void*)((char*)workspace+sizeof(DSM));
 void* outputBuffer = (void*)((char*)inputBuffer+(sizeof(<CacheLine
    >)*<dataIn_CL_size>));
 memset(dsm,0x0,sizeof(DSM));
 memset(inputBuffer,0xAF,CL(<dataIn_CL_size>));
 memset(outputBuffer,0xBF,CL(<dataIn_CL_size>));
 csrs->writeCSR(ADDR_DSM_BASE,intptr_t(dsm));
 csrs->writeCSR(ADDR_INPUT_BUFFER_BASE,intptr_t(inputBuffer));
 csrs->writeCSR(ADDR_OUTPUT_BUFFER_BASE,intptr_t(outputBuffer));
 csrs->writeCSR(NUM_CL,<dataIn_CL_size>);
 //Starting FPGA
 csrs->writeCSR(FLAG_START);
 struct timespec pause;
 pause.tv_sec=(fpga->hwIsSimulated()?1:0);
 pause.tv_nsec=2500000;
 uint64_t counter=TIMEOUT_SEC;
 while(0==dsm->done){
  nanosleep(&pause,NULL);
 }
 //Releasing the buffers
 fpga->freeBuffer((void*)workspace);
 //Done
 delete csrs;
 delete fpga;
}
```

Figura 3.5: Intel OPAE API code example: one input, one output and one accelerator.

to the host code as shown in lines 1-4 for the example depicted in Figure 3.6(a).Then, a set of n_i input streams and m_i output streams could be allocated for each accelerator unit i, as shown in lines 5-8. All accelerators could be started concurrently as depicted in line 9, and work in a synchronous mode, where the host code waits for all tasks to be finished with a timeout (or not). Furthermore, it is also possible to start a specific accelerator a_i to process an input stream of l_i data length, and in addition, the host can concurrently computes a task. The host could verified if the accelerator is already finished by using a query method. More than one accelerator



(a) (b)

Figura 3.6: (a) Host code for multiple accelerators; (b) The ADD memory interface manager; (c) A concurrent execution example with two accelerators.

tasks can simultaneously execute in a synchronous mode as shown in Figure 3.6(b). The input streams, task computations, and output streams can overlap in time.

The Figure 3.7 shows a detailed view of the hardware and software layers to implement the transparent host/accelerator interface in the Intel CPU/FPGA platform [50]. The API at software level is encapsulated by object-oriented classes. Our API at hardware level encapsulates the Intel/OPAE [70] interface by using a multiple memory manager interface to communicate with the accelerator units.

Regarding the target FPGA Cloud platform, the ADD framework maps the dataflow on a Intel processor with an integrated FPGA, which provides all the capabilities of a Broadwell Xeon Processor with the added functionality of an Arria 10 FPGA in the same package [50, 63]. In this platform, instead of usual accelerator hardwares which have theirs own off-chip DDR global memory like GPU, the Intel FPGA approach employees the low latency Quick Path Interconnect (QPI) to share the DDR with the Xeon processor that opens the opportunity to accelerate problems in a different way [63]. In addition to the QPI channels, there are also two PCIe Gen3x8 interfaces. Furthermore as already mentioned, the Intel hardware provides cache coherency between the processor and accelerator.

3.4 Performance Metrics and Analysis

This section presents the signal processing case studies from our previous work, *ADD Harp1* [29], and three new case studies in order to draw insights for the elaboration of a conceptual analysis to compare CPU and FPGA performance. First, it is important to determine if the workload is sensitive to latency, bandwidth, or arithmetic operations. There are two categories: memory-bound, and compute-bound. The

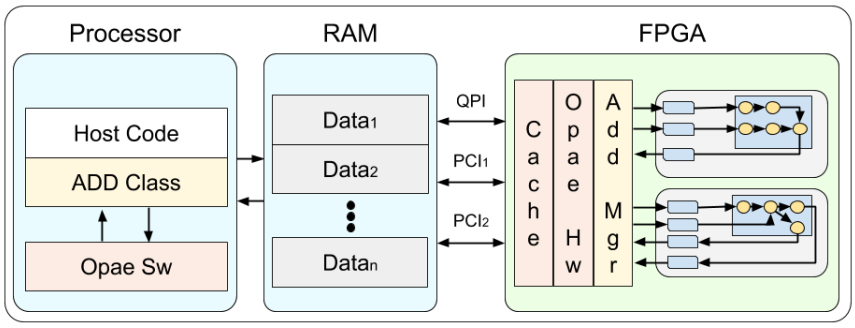


Figura 3.7: Intel Cloud CPU/FPGA Platform: a Broadwell Xeon Processor plus an Arria 10 FPGA in the same package, ADD and OPAE hardware and software layers.

processor and/or accelerator throughput can be measured by the number of operations it performs in a period of time. Nevertheless, not all operations are created equal, and it is important understanding the strengths and weaknesses at different levels. For the same reasons, regarding energy efficiency, the metric Gops/Watt is dangerous since it depends on the operation types. For instance in GPU code, a multiply-add (mac) will perform two basic operations and it could be counted as just one unit. A constant shift operator can be implemented hardwired in FPGA-logic, and no operation is performed. Furthermore, we also evaluate the temporal and spatial parallelism. The dataflow model improves the performance as the pipeline depth increases. Moreover, one can achieve significant operation throughput improvements by replicating dataflows. Doing so causes each dataflow to expose more requests in-flight at once, leading to higher saturation of the CPU/FPGA memory interface. Therefore, as a basic principle of programming, one should always be thinking of ways to minimize host-FPGA transfers and/or to reuse the data to maximize the number of computations per byte.

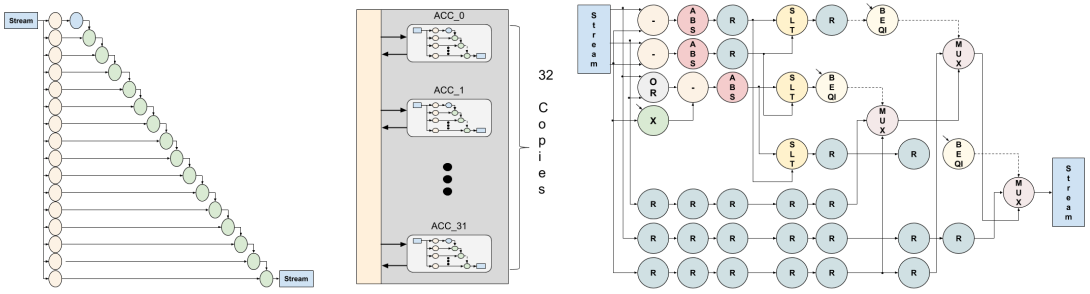
3.4.1 Signal Processing and Map-and-Reduce Benchmarks

In our previous work *ADD Harp1* [29], the target platform was the first Intel CPU/FPGA prototype where an accelerator hardware module (AHM) occupies the other processor socket in a 2-socket motherboard [58], and the peak bandwidth between the CPU memory and FPGA is 7 GB/sec for read and 5 GB/sec for write operations [58]. The software API was based on the Intel Accelerator abstraction layer SDK, which differs from our current work based on Intel OPAE SDK [70], since this first prototype was discontinued. As an initial proof of concept, six showcasing examples for this first Intel prototype platform have been compared to a CPU

implementation in our previous work [29]. For signal processing, the following benchmarks have been evaluated: 8-tap and 16-tap finite impulse response (FIR) filter, the Paeth filter (compression of PNG images), and Gouraud interpolation method. Considering map-and-reduce paradigm, a parallel sum reduction and a histogram as shown in Figure 3.1(c-d). Moreover, the previous interface [29] was limited to one input and one output FPGA cache line with 64 bytes width.

The reduction depicted in Figure 3.1(c) receives one cache line (64 Bytes) as input. Considering 16 bits data width, a peak CPU-FPGA bandwidth of 7 GB/s, and 32 inputs instead of 16 shown in Figure 3.1(c), all 32 operators work in pipeline fashion every clock cycle, the peak number of operations for one copy is equals to $\frac{7GB/s}{64B} \times 32ops = 3.5Gops/s$. In previous work *ADD Harp1* [29], the reduce example is the worst case among the six examples considering the measured operation throughput for 512 MB of data was $1.9\frac{Gops}{sec}$, which is 54% of the maximum throughput. On the other hand, the 16-Tap FIR achieves the best result among the test cases. The FIR peak number of operations for 32 copies is $32 \times \frac{7GB/s}{64B} \times 31ops = 108.5\frac{Gops}{s}$. Figure 3.8(a) details one 16-tap FIR copy, and Figure 3.8(b) shows 32 copies out by one cache line to maximize the throughput. The FIR replication exploits the spatial parallelism where an entire cache data line is fetched every clock cycle. The measured operation throughput achieves $39.7\frac{Gops}{sec}$ in [29], which is only 36.6% of the maximum throughput. The novel manager interface, presented here, improves the CPU-FPGA communication to improve the achieved throughput. The remain four examples presented in [29] are between the two previous cases: Reduction and FIR. While the reduction performs 32 ops per cache line, and the 16-tap FIR performs 992 ops, the number of operations per cache line is for the histogram, the gouraud, the paeth and the 8-tap FIR are 64, 80, 136, and 480, respectively. Figure 3.8 also details the paeth dataflow where there are control flow nodes.

In this work, we improve the throughput in comparison to our previous work [29] by adding an efficient memory interface which allow us to replicate even more the dataflow copies to exploit spatial parallelism. Therefore, maximizing the application´s use of the memory bandwidth is a fundamental step in the performance tuning. If the memory interface is not properly used, other optimizations will likely have a negligible effect. Furthermore, three new case studies are introduced in next sections to illustrate the different new features of the ADD framework.



(a) (b) (c)

Figura 3.8: (a) One copy of a 16-Tap FIR dataflow; (b) 32 FIR copies for input stream (c) Paeth dataflow.

3.4.2 K-Nearest Neighbor algorithm - KNN

Big data brings a lot of challenges regarding the variety of data, where algorithms that take large sets of data and attempt to classify entries into different clusters for more specific processing. K Nearest Neighbor (KNN) is one widely applied algorithm that aims to add new entries to the previously classified dataset. Therefore, we have selected it to be implemented as a case study for our tool.

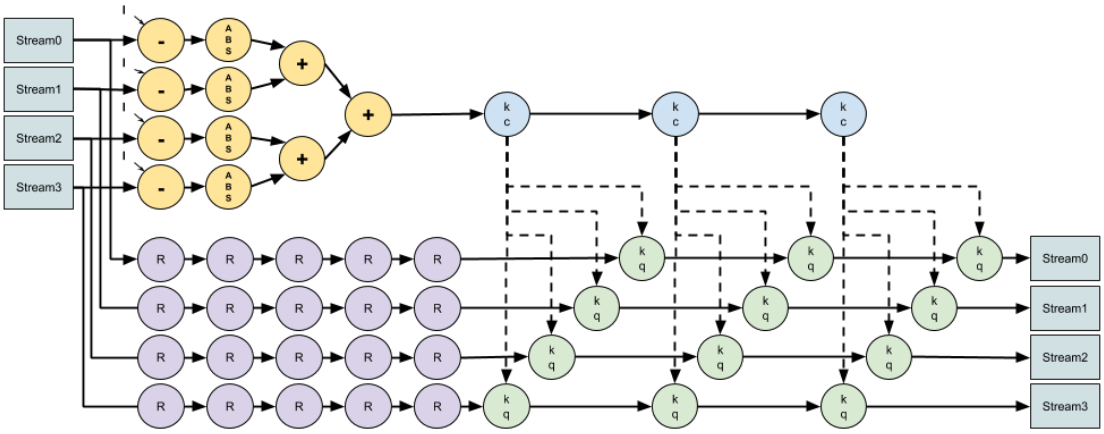


Figura 3.9: Knn algorithm implemented.

An entry is classified by a majority vote of its neighbors to be assigned to the class most common among its k nearest neighbors. Assuming a four dimensional dataset. Suppose the summing of absolute differences to be used to compute the distance function. Figure 3.9 depicts a dataflow to implement the KNN algorithm for a four dimension data set and considering $k = 3$. First, the distance is computed by using a map-and-reduce tree as shown in Figure 3.9, and then the values are

forwarded to a dataflow chain, which consists of three stages of two new store-and-forward operators k_c and k_q. The KNN dataflow is generic and it can easily be extended to any value of k.

The operator k_c belongs to a class of store-and-forward predication operators that performs two functions: $v = f(in,v)$ and $out = g(in,out)$, where in, out, and v are the input value, output value, and the local internal value, respectively. For k_c operator employed in the KNN dataflow, $v = Min(in,v)$ and $out = Max(in,v)$. Therefore, the three stages of the k_c behaves as a priority queue to keep in the three minimum values and drop out greater values. The closest elements are stored in the priority queue. In addition, the predication operator as the k_c has a second output to propagate the predicate value to conditionally execute a predicated target operator.

On the other hand, the operator k_q belongs to the predicated instructions, which a conditionally controlled by the predication values. The k_q execution is controlled by the predicate value, and it chooses to drop, enqueue, or forward the input value. There are a four chain of three k_q components to keep track of the closest neighbors found so far for our four dimension elements of this case study.

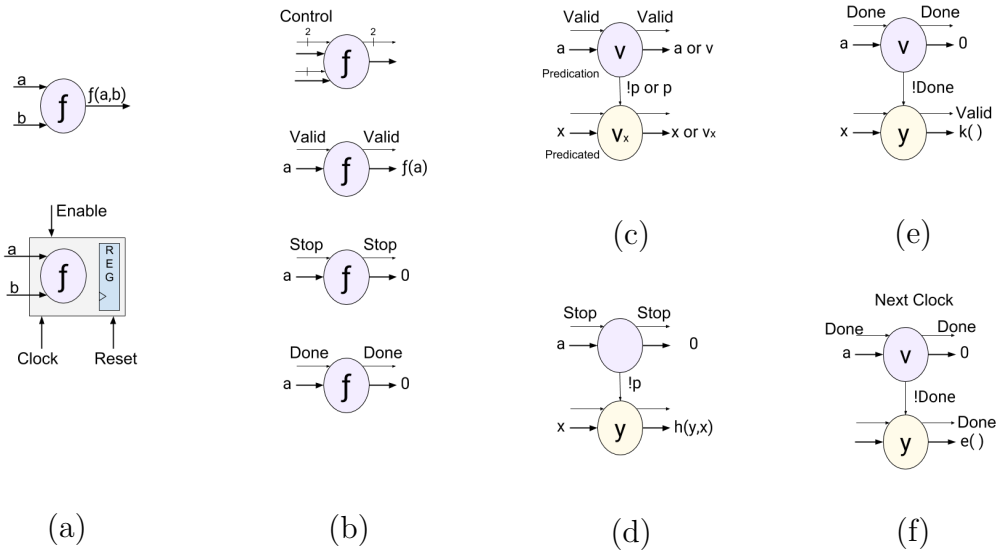


Figura 3.10: (a) Dataflow node and hiding control signals; (b) 2-bit control flow protocol; (c) Valid predication; (d) Stop predication case; (e) Done predication case ; (f) Done predication case next cycle.

As mentioned before, the high-level approach presented here hides protocol control signals: clock, reset, enable, and dataflow controls. All signals are automatically generated at the circuit level. Figure 3.10(a) shows a binary operator which implements a function $f(a,b)$ at dataflow level, and its synchronous behavior with

clock, reset, and enable signals at register transfer level. In addition, each operator has an internal register to implement the pipeline execution. Moreover, the enable signal provides tolerance to cache misses. Different from GPU architecture which has a larger global memory in addition to local memory, the Intel CPU-FPGA has only a local 64 KB cache inside the FPGA, and when a data request generates a cache miss, the enable signal will stall the dataflow pipeline until the data is fetched from memory and then the execution can be continued.

Furthermore, the ADD dataflow provides an implicit control flow to manage the beginning and the end of stream computation. Two extra bits are embedded together to the dataflow data edges as depicted in Figure 3.10(b). Most operators work with three control state (valid, stop, done). The *valid* state means that the input data is valid to be computed, and the control value is also propagated to the output as shown in Figure 3.10(b). The *stop* state could be used for a cache miss or a multiple cycle local computation, where an operator could has an internal pipeline with the latency greater than one clock cycle. Finally, the *done* state means that the computation is finished.

Besides traditional control flow methods, as in processor instruction sets in our ADD dataflow library, the predication is an architectural feature that provides an alternative to conditional branch instructions. The KNN dataflow is an example how predication could be employed to simplify the final implementation.

Figure 3.10(c-e) shows how the predication and predicate operators are orchestrated. Considering that the predication operator receives a valid state in its inputs. The operator will perform the computation, and a predication function $p(a,v)$ will be computed to return bypass (false) or valid (true) state. In case of stop input state as shown in Figure 3.10(d), the predication operator drops out its output and the predicate bypass value is send to the predicated operator.

Finally, if the done state arrives in the predication operator, its output is dropped out, and the predicate value is set to done as shown in Figure 3.10(e). However, the predicated operator could have two behaviors. In the current clock cycle, a $k(x,y)$ function is computed, and a valid control signal is propagated. For the next clock cycle the done signal will be propagated, and a $end(x,y)$ function will be performed.

3.4.3 Tiny Encryption Algorithm - TEA

The tiny encryption algorithm (TEA) [86] is light weight cryptographic algorithm with a smaller code and memory footprint. Figure 3.11 depicts the kernel for the en-

crypt, which processes a pair of 32 bits data words (in_0,in_1), and four cryptographic keys. The decrypt code has a quite similar code. In addition, the dataflow for the loop body is also shown in Figure 3.11. For each input pair, thirty two iterations are performed, where each iteration depends on the previous value of the iterator. Hence, this data dependency constraint prevents to apply loop unrolling to overlap the iterations. Since the loop body computes 17 operations, and the loop has 32 iterations, for each input pair 544 operations are performed.

```
1 void encrypt(uint32_t* IN, uint32_t* k) {
2   uint32_t sum = 0, delta = 0x9e3779b9;
3   for (int i = 0; i < 32; i++) {
4     sum += delta;
5     IN[0] += ((IN[1] << 4) + k[0]) ^ (IN[1] + sum) ^ ((IN[1] >> 5)
          + k[1]);
6     IN[1] += ((IN[0] << 4) + k[2]) ^ (IN[0] + sum) ^ ((IN[0] >> 5)
          + k[3]);
7   }
8 }
```

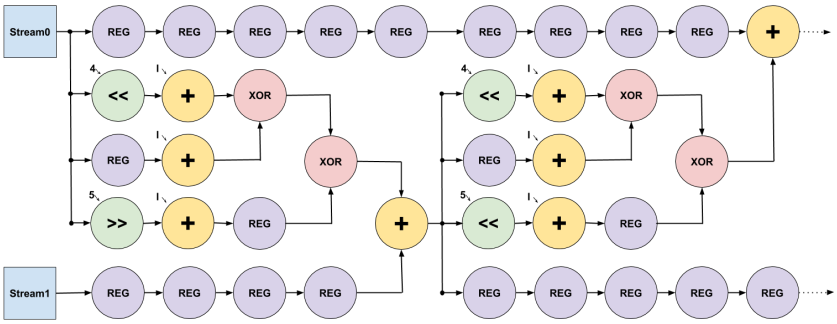


Figura 3.11: TEA cryptography code and the corresponding dataflow for the loop body (lines 4-8).

The decrypt code has the same amount of operations. As study case for a deeper pipeline design, we propose a three-stage application. The encrypted data is send to the FPGA, the data is decrypted inside the FPGA (1^{st} stage), a compute task is done (2^{nd} stage), and finally the result is encrypted before to be sent back to the CPU (3^{rd} stage). We propose to fully unroll the decrypt loop (32 times), followed by 1-D convolution (a four-tap FIR) as a task example, and then also to fully unroll the encrypt function (32x). The dataflow consists of a deeper pipeline with 1102 operators, and 1410 registers to balance the paths. The input data is pair numbers of 32 bits width. Figure 3.12 shows a simplified view of the deeper dataflow.

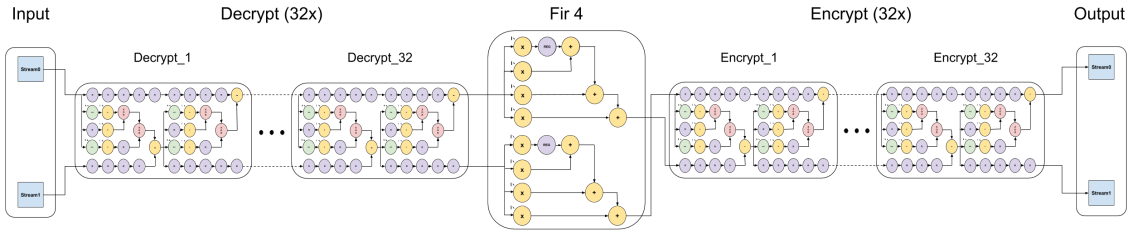


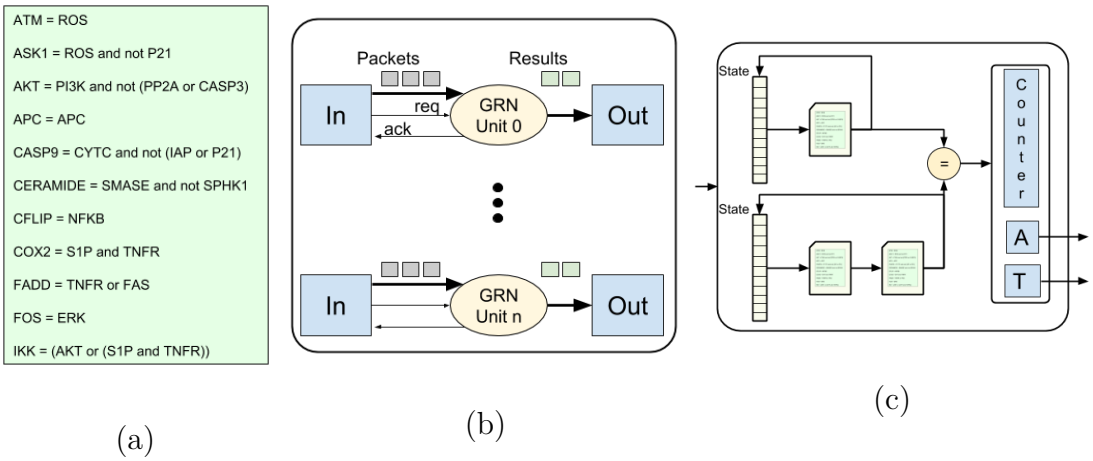
Figura 3.12: A Deeper Dataflow: Decrypt, compute the convolution 4-tap FIR, and Encrypt.

Nowadays, FPGA accelerators have been widely used to implement deeper neural networks [63]. Data compression and cryptography are also two fields suitable for deeper FPGA pipelines due several proprieties. For the TEA algorithm, the constant shift operations will be efficiently implemented as simple wires inside the FPGA. The dataflow is described by an open structural format as a graph (components and connections), and it could be automatically generated without use the graphical interface for regular design like this example with more than thousand components. Furthermore, one could take advantages of FPGA large amount of resources (registers, DSP, BRAM, and hundred thousands of logic elements) to instantiate more than on copy by doing design replication.

3.4.4 Gene Regulatory Networks - GRN

Systems biology poses computational challenges that surpass the capabilities of even the largest and fastest computer platforms. A particular challenge is simulating the dynamics of Gene Regulatory Networks (GRNs), which could be used for a variety of pathological settings, including genetic tumor syndromes, cancer, and obesity. Previous work [79, 87] propose specific FPGA accelerators to speed up GRN computations in addition to traditional CPUs. We have chosen this case study in particular because it allows us to present three different aspects of the ADD framework.

First, the basic step in GRN computation is the evaluation of a set of n Boolean expressions. Figure 3.13(a) depicts a subset of expressions used in the Colitis-Associated Colon cancer (CAC) GRN evaluated in [79]. The CAC GRN has 70 nodes, and the expression evaluation consumes 272 *x86* assembler instructions. Suppose a 20-core processor that could execute 4 instructions per clock cycle per core at 3 Ghz. Hence, it could achieve the peak performance of $20 \times 4 \times 3 = 240 \frac{Gops}{sec}$, and since a expression consists of 272 instruction, the peak expression throughput will



(a) (b) (c)

Figura 3.13: (a) A subset of Boolean expression for the CAC Gene Regulatory Network; (b) A set of GRN asynchronous accelerator; (c) GRN accelerator unit.

be $888\frac{Mexpressions}{sec}$. An FPGA can easily evaluate a set of 128 Boolean expressions per clock cycle at 200 Mhz. Therefore, the FPGA peak performance could achieve $25600\frac{Nexpressions}{sec}$, which represents a theoretical speed-up of about 30. Furthermore, the real scenario is even more favorable to the FPGA approaches, for instance, the CAC GRN computation is 174.2 times faster than a 20-core multiprocessor in [79].

The second feature is the introduction of asynchronous operators. The GRN algorithm presented in [79] computes the transient size and the steady cycle for each network state. These computations could be performed in parallel by using more than one accelerator unit. However, the state computations are irregular in the sense that each computation has a variable-length clock cycles. We propose to use an elastic dataflow approach provided by the ADD tool in an asynchronous fashion through the use of *ready* and *acknowledge* signals as depicted in Figure 3.13(b) for a set of n accelerator units attached to the ADD interface memory manager.

The third point deals with accelerator design reuse and portability. The accelerator GRN generator presented in [79] has a memory interface customized to get the maximum performance for the first Intel CPU-FPGA platform [58], which was discontinued. Therefore, we create an accelerator unit by reusing the GRN kernel from [79] as shown a simplified diagram in Figure 3.13(c). For the CPU interface, we use the ADD portable interface memory manager. Moreover, since this unit has a variable data width of n bits (70 bits for the CAC GRN [79]), we also introduce an asynchronous packet-based link. First, the n-bits is sent in data packet of 32 bits width. The computations are performed inside the GRN operator, and finally, the results are sent back to the CPU. These three tasks (send data, compute, and send results) are asynchronously executed and also overlapped in time.

3.5 Experimental Results

In this section, we perform experiments to corroborate the effectiveness of the proposed accelerator design and deploy tool (ADD). First, the simulation and the compile time are evaluated in Section 3.5.1. Section 3.5.2 presents the FPGA resources consumed by each case study as well as the compile time. Finally, a performance evaluation and analysis is conducted in Section 3.5.3.

3.5.1 Simulation and Compile Time Comparison

Considering the design cycle, Table 3.2 shows all steps provided by the ADD framework by using two dataflow algorithms as examples: KNN and TEA. Our KNN instance has a small dataflow with only 54 nodes as shown in Column *DFG size*. Nodes are the dataflow operators, which also include registers to balance the pipeline as shown in Figure 3.9. The TEA dataflow has a deeper pipeline with 645 stages and 2452 nodes (including registers) as shown in Figure 3.12. Columns *Proposed ADD* shows the simulation execution time at dataflow level and the translation time to generate the accelerator design plus the hardware interface in Verilog. Columns *Intel ASE* depicts the time to compile the ADD generated Verilog design to be simulated in the ASE/OPAE SDK. Moreover, the time to recompile in case of minor changes is shown in next column, and column *simul. time* shows the time for a cycle accurate simulation, which includes the CPU and memory transactions (including cache misses) to communicate with the FPGA. The FPGAs columns display the time for a full compilation to generate the FPGA bitstream in Intel/Altera Quartus tools, as well the number of adaptive logic module (ALM) required to implement the final design. FPGA RAM blocks and DSP blocks are not depicted. The ALM size can be used as a reference to show the complexity of the placement and routing steps performed during the FPGA compilation.

Tabela 3.2: Design and Validation from the dataflow to the FPGA.

Example	Node Size	Proposed ADD		Intel ASE			FPGA Compile Time			
		Simul. Time (s)	Verilog Generation (s)	Compile Time (s)	Recompile Time (s)	Simul. Time (s)	Low cost Cyclone IV		HPC Arria 10	
							hh:mm	ALMs	hh:mm	ALMs
Knn	54	2.25	1.192	28.11	20.53	2.46	0:03	7,945	1:25	90,721
TEA	2,452	11.79	5.238	23.12	20.87	4.58	1:33	72,674	1:57	120,309

The last columns in Table 3.2 corroborate that a significant FPGA productivity bottleneck is the compile time, mainly due to the complexity of placement and routing of all internal resources. Our proposal includes the low cost FPGA, where one or more dataflow kernels as well as new operators could be quickly validated

during the early stages of the design process. For instance, the KNN kernel was compiled in three minutes for a low cost board in comparison to 1 hour and 25 minutes to target a high performance Arria 10 FPGA due the OPAE layer overhead. Moreover, even the cycle accurate simulation in ASE tool requires few seconds/minutes to compile, and the simulation time depends on the input size. Hence, the two abstract levels (dataflow and ASE) and the low cost FPGA approach give to the developers feedbacks within minutes, which it is very useful for the next design iteration. In addition, even the dataflow simulation which includes signals and synchronized protocols, it hides details from the user/programmer using dataflow of abstraction level and real data stimulus, and mitigates the number of (re)compilation cycles. However, reducing the FPGA compile time is still a critical issue to enabling mainstream usage [12].

3.5.2 FPGA Resource Usage and Compilation

Table 3.3 shows the total number of dataflow copies, the number of accelerator units, the FPGA resource usage, and the FPGA compile time for the evaluated dataflow implementations. In addition to the ADD resource usage, Table 3.3 depicts the resource usage for Intel OpenCL SDK [23]. Most accelerator data access begins in CPU memory, and most applications tend to be limited by memory bandwidth. Considering FPGA designs, a lack of one kind of resource can prevent an accelerator duplication. Moreover, an increasing on the critical path can mitigate the clock frequency, and hence, it can drastically reduce the overall performance.

Tabela 3.3: Dataflow copies, Accelerator units, FPGA resource usage, and Compile Time.

Benchmark	**FPGA Resources**						**Compile Time**	
	ALM		**Ram Block**		**DSP**			
	ADD	**OpenCL**	**ADD**	**OpenCL**	**ADD**	**OpenCL**	**ADD**	**OpenCL**
Loop Back	81,789 (19%)	98,248 (23%)	449 (17%)	405 (15%)	0 (0%)	0 (0%)	1:40	6:02
Paeth	101,396 (24%)	107,027 (25%)	889 (33%)	435 (16%)	32 (2%)	0(0%)	2:19	7:00
Gouraud	99,981 (23%)	101,448 (24%)	889 (33%)	397 (15%)	0 (0%)	0(0%)	2:09	6:04
Fir16-Tap	103,555 (24%)	106,157 (25%)	663 (24%)	409 (15%)	1,024 (67%)	8(1%)	2:42	7:16
TEA	251,503 (59%)	126,384 (29%)	1,289 (48%)	434 (16%)	0 (0%)	64(4%)	5:33	9:16
KNN	131,197 (31%)	113,733 (26%)	1,167 (43%)	469 (18%)	0 (0%)	0(0%)	2:40	7:35
GRN	133,154 (31%)	138,485 (32%)	2,119 (78%)	493 (19%)	0 (0%)	0(0%)	2:13	9:37

First line in Table 3.3 depicts the *loopback* example to given us a baseline of the compiler and OPAE resource usage overhead. The *loopback* example just sends data to the FPGA and sends back to the CPU. As we can see, the lower bound for the compile time is around one hour and forty minutes. At least 20% of ALMs and BRAMs are used to build the minimum infrastructure to include one accelerator

unit. While ADD is implemented as nested structure inside the OPAE layer that provides the memory interface, the OpenCL could generate a customized memory interface. The ADD consumes more Ram Blocks than OpenCL, since the ADD memory interface consumes Ram Blocks in addition to the OPAE Ram Blocks. The following three lines shows the results for three accelerators to compare to our previous work ADD Harp1 [29]. First, we increase the number of accelerator units to improve the performance with a little impact on the resource usage for the *paeth* and the *gouraud* dataflows, where each accelerator unit has 8 dataflow copy. The accelerator has four units each, and therefore the total number of dataflow copies is 32 for these benchmarks.

The unique exception is the 16-tap FIR since it consumes a larger number of DSP resources, and therefore, there is a lack of DSP resources to scale it to 4 units. While OpenCL uses ALMs to perform the multiplications. Finally, the last four lines show the case studies to exploit the resource usage to maximize the performance, and also to exploit the design trade-off when choosing the number of accelerator units and dataflow copies per unit. The maximum number of input streams was fixed in 16. The *TEA* has a deeper pipeline with more than two thousands components per dataflow copy, which is expensive for the FPGA compiler to scale it. The *TEA* and the *KNN* was evaluated with one dataflow per unit. Finally, the *GRN* which has a single input stream per dataflow has evaluated with 12 units. All dataflow operates in synchronous mode, except the *GRN* that uses the asynchronous mode. Finally, the OpenCL full compilation time is two to three times greater than the Verilog generated by the ADD approach.

3.5.3 Performance

Table 3.4 presents the performance evaluation considering the previous work ADD Harp1 [29] and our proposed ADD implementation for the novel Intel Xeon - Arria 10 FPGA. The two first columns depict the benchmark name and the input data size in GBytes.

First three lines depict dataflow results to compare to our previous implementation in the Intel Xeon - Stratix V FPGA prototype. The last column depicts the speedup factor. For the *Gouraud* and the *Paeth*, we improve the execution time by a factor of 3.2 and 3.9, respectively. These results could be explain as we increase by a factor of four the number of dataflow copies thanks to the new multiple accelerator unit interface. Moreover, the FIR achieves a better speedup of 3.3 times even it only doubles the number of dataflow copies. The FIR has a deeper pipeline,

Tabela 3.4: Execution Time, Dataflow Replication and the number of operations per cycle considering the Intel Xeon-StratixV FPGA prototype and the Intel Cloud Xeon-Arria 10 FPGA.

Benchmark	Data Size GB	ADD Harp1		Proposed Work					Speedup
		Execution Time (ms)	Dataflow Copies	Execution Time (ms)	Dataflow Copies	Theoretical Bound	GOPs	Clock MHz	
Paeth	0.24	80.9	8	20.5	32	28.7	26.6	200	3.9
Gouraud	3.58	1202.9	8	376.4	32	16.9	12.8	200	3.2
Fir16-Tap	1.91	790.7	32	240.6	64	209.2	136.2	200	3.3
Knn k3	9.53	-	-	927.2	4	43.9	35.9	400	-
TEA	0.48	-	-	210.9	4	1319.6	237.3	200	-
GRN	1.00	-	-	9,337.11	12	-	-	200	-

and it achieves a high operation throughput. The remain lines depict benchmarks which have been only evaluated in the novel Intel Xeon-FPGA platform, since the previous platform used in [29] does not support OPAE layers.

In Table 3.4, column *Theoretical Bound* is the maximum number of parallel operations per second by assuming the peak memory bandwidth. Therefore, the *Theoretical Bound* is equals to $\frac{Ops_{dfg}}{Size_{in}} \times Memory_{band}$, where the Ops_{dfg} is the total number of dataflow operations, the $Size_{in}$ is the dataflow input size in Bytes, and the $Memory_{band}$ is the peak memory bandwidth value of 13.5 GB/s for the QPI/PCI channels interface provides by Intel/OPAE. Column *G*OPs is the measured value of the number of operations computed per second during the experiments. Here, our goal is twofold: 1) Evaluate the effectiveness of the memory interface layers; 2) Estimate how far the result is from the optimal.

The memory interface has a complex number of layers which includes OPAE layers (request queues, cache manager, QPI and PCI channels, etc.) and our interface memory manager unit. Considering multiple input stream, our interface manager should quickly multiplex between various concurrent streams since there is only one single OPAE input stream channel. For three cases (*Gouraud, Paeth* and *Knn*), we are quite close to the upper bound. However, for two cases (*TEA* and *FIR*), there is still room for improvements. Both cases have deeper pipelines, where stalls have a great impact, and it may occurs due cache misses or stream multiplexing.

The *GRN* uses an asynchronous operation mode and it is bound by the number of hardware resources. In addition for the GRN, the *Theoretical Bound* and the *GOPs* are not depicted in Table 3.4 since the operations are fine grained (Bit Boolean operations), and the network state computation has a variable length execution time. On average as shown in [79], a state computation spends around 80 simulation steps. Each step evaluates the set of the Boolean equations three times, where each set has 272 Boolean operations. Therefore, for the GRN with 8 copies that computes 2^{27}

states, the total number of operations is 2^{27} states $\times$ 3 $\times$ 272 Boolean equations $\times$ 80 steps = 8.76 Tera operations. Hence, the GOPs/s is around $732\frac{Gops}{sec}$.

Tabela 3.5: Performance evaluation on different target platforms: single-core CPU, multi-core-CPU, GPU, and the proposed CPU-Arria 10 FPGA by using the ADD framework.

Benchmark	Instructions Giga - 10^9	Execution Time (ms)					Speedup			
		CPU	OpenMp	GPU	OpenCL	ADD	CPU	OpenMp	GPU	OpenCL
Paeth	0.29	30.56	24.05	5.53	23.526	20.5	1.49	1.17	0.27	1.15
Gouraud	5.25	514.07	137.10	169.59	183.695	376.4	1.37	0.36	0.45	0.49
Fir16-Tap	7.91	930.64	338.26	360.79	63,394.4	240.6	3.87	1.40	1.49	251.0
Knn k3	75.9	6,785.59	3,285.35	1,632.21	14,507.501	927.2	7.32	3.54	1.76	15.65
TEA	117.9	20,048.42	5,012.11	35.55	3 02.305	210.9	95.06	23.77	0.17	1.43
GRN	20,791.24	3,119,110.00	459,424.00	79,162.00	119243.378	9,337.11	334.05	49.20	8,48	12.78

It is important to highlight that a FPGA design is reconfigurable, and it could be continually improved even after the device fabrication in comparison the CPU/GPU for which the decisions are taken at device design time. Therefore, the FPGA compiler, the OPAE interface, our memory interface unit, the dataflow operators, and the dataflow algorithm could be even more improved to get close to the upper bounds. The same remark is also valid for FPGA OpenCL approach. Regarding the clock frequency, all generated designs have been successful compiled to work at 200 MHz, except the *Knn* accelerator, which is able to work properly at 400 MHz.

Since, the target Arria 10 FPGA can work at 400 Mhz, future improvement in the placement and routing process during the FPGA compilation, as well as in the OPAE and the ADD interface memory manager could speedup all designs presented in Table 3.4 by a factor of two. Moreover, a code optimized for a specific GPU or CPU architecture has short life cycle, and FPGA solution allows the tool developers to design in-line processing directly from data interfaces, in addition to provide highly parallel customizable architecture.

Table 3.5 shows the results for the comparison of the proposed high-level dataflow tool with the high performance accelerators on a multi-core processor by using OpenMP directives, and on a Nvidia GPU. The processor is an Intel Xeon CPU E5-2630v3, clock frequency 2.40GHz, 64 GB DDR, 8 cores (16 threads), 20 MB L3, and 85 Watts. The GPU is Tesla K40c, 2880 CUDA Cores, clock frequency 745 MHz, and 150-235 Watts. As baseline, Table 3.5 also depicts the results for a single-core execution. Column *instructions* shows the number of executed x86 instruction for the baseline single core execution in Giga instructions. This metric allows us to compute the number of Gops/s performed by a single core. For instance, the *Paeth* execution has a measured throughput of $9.49\frac{Gops}{sec}$.

Since the case studies, except the *GRN*, are data-intensive kernels with a small code footprint, the CPU and the GPU compilers are able to generate highly optimized code. The following columns show the execution time in milliseconds (ms) for the evaluated target platforms. Column Xeon CPU X86 is a single core execution, and column OpenMP is a eight core execution. The last three columns depict the speed-up factor relative to the execution with proposed ADD accelerator. A ratio greater than one means that it is better to our approach while a ratio less than one means a slowdown. The ADD results outperform the single core CPU for all study cases. It is important to highlight that even the FPGA clock (200 Mhz) is more than 10 slower regarding the OpenMP with eight cores (16 threads), the multi-core is faster only for the gouraud application. However, considering that the average power consumption (estimated by using Intel tools) is around 22 Watts in comparison to the 85 Watts for the eight cores CPU, the ADD is still more than 1.4 times energy efficient. Considering the GPU accelerator, the GPU K40 consumes between 150 Watts to 235 Watts, even the GPU outperforms ADD for three case studies, ADD is still more energy efficient. The best scenario for the GPU is the *TEA*, where the GPU is 5.9 times faster. However the GPU consumes 6.8x-10.9x more energy.

We also compare the ADD results to the high-level OpenCL Intel SDK for FPGA. As already mentioned, the OpenCL generates implicitly the dataflow and the quality of the generated accelerator is quite sensitive to the code style and primitives. The ADD outperforms the OpenCL in 5 of the 6 case studies. Gouraud is the only exception, and it is the most simple evaluated code. For Paeth and TEA case studies, the OpenCL and ADD achieve almost the same performance. For KNN and GRN which has conditional branch, and in addition, the GRN has variable length loops (*while structure*) and function call, the ADD outperforms the OpenCL by one order of magnitude. Finally, the FIR case study is an example of how the performance is sensitive to the code style and primitives. As demonstrated in detail in [88], a FIR implementation code could be up to 800x slower due the code style. The TEA case study includes a "good"FIR code inside, and the performance is similar to ADD. In contrast, a more naive code as implemented for the FIR case study results in a slowdown of 250x. Other implementations for the FIR algorithm in OpenCL are available in the complementary material to exemplify the trade-off on the code style to tune the FIR implementation [85]. Hence, our approach is simple and it is straightforward to explicitly map the dataflow on the target architecture for data intensive stream-based kernels.

Regarding the GPU GRN case study shown in Table 3.5, it is not yet fair

to compare to the GPU, since an optimized implementation should be developed and tunned. However, different from the other case studies for which the GPU vectorization is straightforward, the GRN has an unbalanced compute tasks, and it causes GPU thread divergences that degrades the GPU performance. In comparison to the FPGA customized GRN generator presented in [79], the ADD is twice slower. Nevertheless, it was a proof of concept of the asynchronous mode, and the ADD memory interface is not bit width optimized for the target GRN word. Therefore, there is room to improve it. Moreover, even the current GRN accelerator is more than 50 times faster the 16 thread OpenMP implementation. Finally, to summarize the outcome of the high-level dataflow tool proposed here to transparently target FPGA accelerator devices, our tool is up to 18.8 and 193.2 times more energy efficient than GPU and multi-core evaluated platform, respectively.

3.6 Conclusion

As with the early days of GPU-based accelerators in high performance computing (HPC) data centers, one question is how to convince HPC resource users to shift to a new FPGA/dataflow paradigm. One way is to provide higher level tools that deliver high performance results. In this work, we present an Accelerator Design and Deploy (ADD) framework to simplify the dataflow-based algorithm design for FPGA by providing programmability and usability required to be competitive. This work contributes to pave the way for FPGAs to become a force in HPC. Huge companies like Amazon, Microsoft, Intel, and research centers have already created FPGA cloud infrastructures to study the potential of FPGAs for energy-efficient scientific computing. Our results have been validated on an high performance Intel Xeon-FPGA Arria 10 platform placed at Germany Paderborn University where the first phase of the ten million Euro Noctua cluster project is underway.

While OpenCL and OpenSPL approaches are based on high level languages which are highly sensitive to coding style and constructs to implicitly extract a structured model, and then generates a dataflow design without mapping between high level code and dataflow structure, our approach provides a simple and, at same time, powerful and explicit expanded view of dataflow design, which can lead to improvements, and enable new application developments and computing platforms. Although, FPGA and dataflow approaches can provide power efficient accelerator, how responding to the current opportunities will require more "out-of-the-box"designs at algorithm high levels and architecture low levels. Moreover,

dataflow does not require neither instruction fetch nor decode like CPU/GPU platform, where these steps are performed every clock cycle.

Furthermore, big data workloads have common streaming computing characteristics, which leads to very efficient hardware implementations through exploitation of concurrency. Finally, considering the performance and the energy efficiency in comparison to the state-of-the-art accelerators based on GPU and multi-core CPU, our experimental results shows the potential of the proposed approach, that can be further investigated and improved in future studies.

Capítulo 4

Kmeans

O algoritmo *K-means* é um método utilizado para aprendizado não supervisionado no agrupamento de dados. Este trabalho apresenta um gerador de código de domínio específico para o *K-means* capaz de gerar código para GPUs e FPGAs. Para aumentar a eficiência, o código é parametrizável e especializado para GPUs da Nvidia e para a plataforma HARP v.2 da Intel/Altera. Entretanto, o gerador é modular e pode ser estendido para outras plataformas de FPGA e GPU. Outra contribuição deste trabalho é simplificação do uso de FPGAs de alto desempenho para programadores, pois não requer nenhum conhecimento de hardware por parte do usuário para prover um acelerador de alto desempenho no nível de software. O gerador também simplifica a programação para GPU. Para os experimentos realizados, em comparação com o tempo de execução em uma CPU Intel XEON, a solução proposta em GPU acelerou em até 55 vezes e o FPGA obteve uma aceleração de até 13,8 vezes. Em relação à eficiência energética, as execuções em FPGA foram até 10 vezes mais eficientes que as GPUs avaliadas. Os resultados foram validados com duas GPUs: K40 e 1080ti.

4.1 Introdução

Atualmente, as GPUs (*Graphics Processing Unit*) e os FPGAs (*Field-Programmable Gate Array*) são duas tendências em aceleradores para desempenho com eficiência energética em comparação aos processadores de uso geral [89]. Neste contexto, a eficiência pode ser maior ao trabalhar com domínios específicos, conforme destacado por David Patterson e John Hennessy na aula do prêmio Turing de 2017 [1]. Apesar dos FPGAs oferecerem maior potencial de eficiência energética, o maior desafio ainda é a sua programação que exige conhecimentos específicos de hardware, mesmo com os ambientes de alto nível como OpenCL [90].

Uma demanda atual é a extração de conhecimento e identificação de grupos

de grandes volumes de dados. Este artigo aborda a implementação do algoritmo *K-means* [91] de aprendizado não supervisionado nas plataformas de GPU e FPGA. O algoritmo *K-means* possui potencial para a paralelização de operações com reúso de dados, que é uma característica apropriada para as arquiteturas alvo. A primeira contribuição deste trabalho é a apresentação de um gerador de código parametrizável para GPU para a execução do *K-means*, que otimiza o reúso dos dados. A segunda contribuição é a geração automática e transparente de hardware e software para execução do algoritmo na nova plataforma CPU-FPGA HARP v.2 de alto desempenho da Intel.

A estrutura do artigo é descrita, a seguir. A Seção 4.2 apresenta brevemente o algoritmo *K-Means*, a Seção 4.3 descreve a plataforma HARP v.2 e a Seção 4.4 apresenta a arquitetura desenvolvida para execução em FPGA. Na Seção 4.5, o gerador para GPU é apresentado. Os experimentos e resultados são discutidos na Seção 4.6, os trabalhos relacionados são discutidos na Seção 4.7, e por fim, a Seção 4.8 apresenta as principais conclusões.

4.2 K-Means

O *K-means* é um algoritmo de aprendizado não supervisionado capaz de particionar um conjunto de pontos em k grupos. O algoritmo é utilizado em larga escala em aplicações de mineração de dados [92]. A cada iteração, todos os pontos são avaliados. Várias iterações são executadas até que convirja. Este processo exige esforço computacional para grandes volumes de dados [93].

A Figura 4.1 apresenta um exemplo com pontos de duas dimensões, x e y, dos quais deseja-se classificar em dois grupos, i.e., $k = 2$. Cada grupo terá um centroide C_i e cada ponto é classificado no grupo do centroide mais próximo. A Figura 4.1(a) mostra o ponto p a ser classificado no grupo G_0 por estar mais perto do centroide C_0. A etapa de classificação é executada para todos os pontos. A próxima etapa é o cálculo da nova posição dos centroides. Essa nova posição é a que minimiza a média do quadrado das distâncias de todos os pontos do grupo. A Figura 4.1(b) mostra o reposicionamento dos centroides após a execução de uma iteração. Também é destacado o reposicionamento do centroide C_0 e os pontos classificados nos grupos G_0 e G_1 na etapa anterior. O algoritmo repete as duas etapas (classificação e reposicionamento) até que convirja ou atinja um limite de iterações.

O algoritmo pode ser descrito com o paradigma de *map-reduce*. A primeira

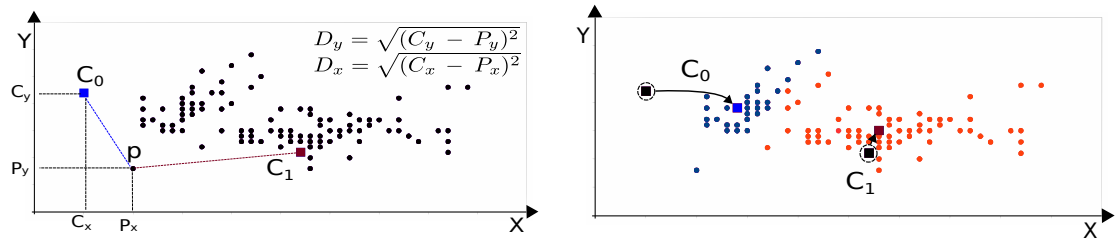
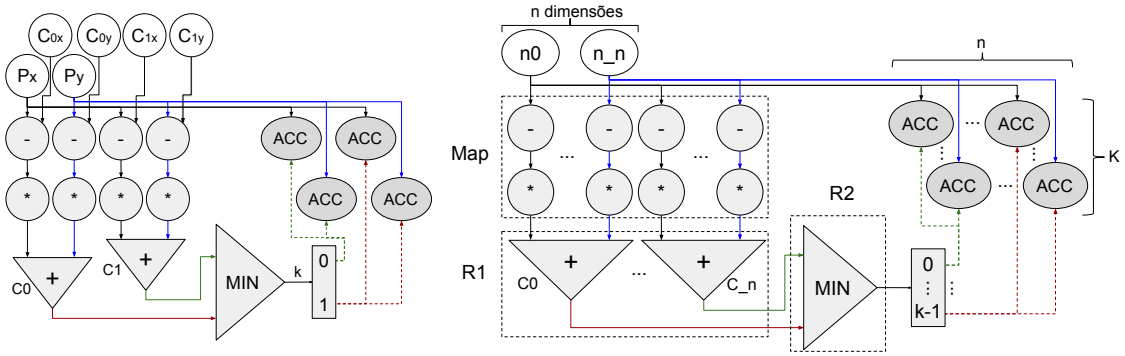


Figura 4.1: Exemplo de agrupamento pelo *K-means*.

etapa de classificação, pode ser definida por um mapeamento seguido de duas reduções. A Figura 4.2(a) mostra o fluxo de operações para o exemplo com duas dimensões ($n = 2$) e dois centroides ($k = 2$). O mapeamento é responsável pelo cálculo da distância em cada dimensão. A métrica de distância utilizada no *K-means* foi a distância euclidiana, ou seja, através do cálculo de raiz quadrada do quadrado da diferença para cada dimensão. Para a dimensão x em relação ao centroide i, será $D_{i_x} = \sqrt{(P_x - C_{i_x})^2}$. Esta operação básica é executada para cada ponto e todas as suas dimensões em relação a todos os centroides. Como é uma métrica de distância, uma simplificação pode ser feita com a retirada do cálculo da raiz quadrada com o intuito de reduzir a complexidade [94]. O segundo passo é uma redução para somar as distâncias de todas as dimensões do ponto em relação ao centroide. No exemplo, para os centroides C_0 e C_1, o cálculo será $D_0 = D_{0_x} + D_{0_y}$ e $D_1 = D_{1_x} + D_{1_y}$, respectivamente. O terceiro passo é a redução para encontrar o centroide mais próximo de p, ou seja, $Min(D_0,D_1)$. A função Min retorna o número do centroide mais próximo de p. Esta é a fase de classificação do *K-means*. Formalmente, para k centroides e n dimensões, o ponto será classificado como $Min(D_0, \ldots, D_{n-1})$ onde a distância para o centroide C_i será $D_i = \sum_{j=0}^{n-1}(p_j - C_{i_j})^2$. Assim, tem-se kn subtrações e multiplicações durante o mapeamento, a redução de soma totaliza $k(n-1)$ adições e a redução de mínimo com $k-1$ comparações. A cada iteração para m pontos, tem-se $m[2kn + k(n-1) + k - 1]$ operações. A primeira fase do algoritmo é apropriada para execução em GPU e FPGA que são capazes de executar todas as operações em paralelo, dependendo do código, da arquitetura e dos valores de m,k e n. Outro aspecto favorável às GPUs e aos FPGAs é o reúso do dados que é definido pelo número de operações realizadas para cada dado lido da memória, que é igual $\frac{m[2kn+k(n-1)+k-1]}{mn} = 3k - \frac{1}{n}$, ou seja, quanto maior for o valor de k, o cenário se torna mais favorável às GPUs e aos FPGAs.

A etapa de reposicionamento é responsável pelo cálculo dos novos valores para



(a) (b)

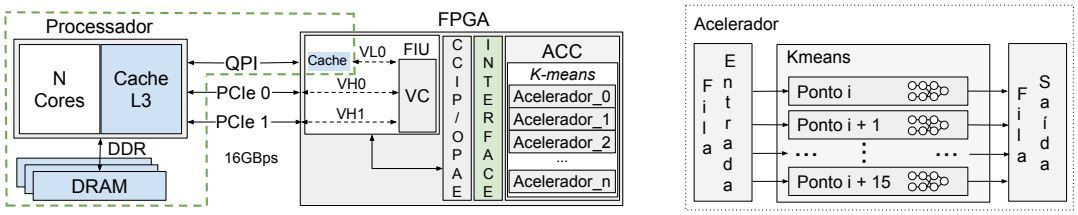
Figura 4.2: (a) Grafo de operações com $k = 2$ e $n = 2$; (b) Grafo Genérico.

cada uma das dimensões dos centroides. Para cada ponto classificado pelo centroide C_i, os valores de cada dimensão são acumulados. Por isto há vários acumuladores (Acc) na Figura 4.2. Se t pontos forem classificados para o centroide C_i, a nova posição na dimensão j para o centroide C_i será calculado por $C_{i_j} = \frac{\sum_{j=1}^{t} p_j}{t}$.

4.3 Plataforma de Alto Desempenho com FPGA HARP v.2

Recentemente, várias plataformas para computação de alto desempenho com FPGAs foram apresentadas por grandes empresas como Intel-Altera [50], Amazon [64] e Microsoft [66]. A plataforma utilizada neste trabalho é o HARP v.2 da Intel-Altera, que possui um FPGA fortemente acoplado através da memória compartilhada com um processador Xeon E5-2600 v4. A Intel provê interfaces em *software* e *hardware* que mantêm a coerência de dados entre o processador e o FPGA.

A Figura 4.3(a) ilustra os principais pontos da arquitetura da plataforma HARP v.2. A Intel provê a API OPAE (*Open Programmable Acceleration Engine*) que traz os objetos necessários para a comunicação do aplicativo com os aceleradores desenvolvidos para a plataforma. Entretanto, é necessário que o usuário possua conhecimentos específicos em FPGA e da documentação do OPAE. A ligação física entre memória e o acelerador é feita através do barramento Intel QPI (*Quick Path Interconnect*) mais dois barramentos PCI-e, nos quais são possíveis taxas de transmissão de dados na ordem de 16GB/s. O FPGA mantém uma *cache* interna de 64 Kbytes. A coerência com as memórias é feita de forma transparente. A CPU e o FPGA trabalham de forma assíncrona e independente.



(a) (b)

Figura 4.3: (a) Plataforma HARP v.2 da Intel (b) Arquitetura do acelerador.

Diferente da arquitetura de uma GPU que possui memória dedicada, o FPGA do HARP v.2 terá sempre que ler dados das memórias conectadas a CPU (*cache* L3 e RAM) através dos barramentos PCI e QPI, uma vez que a cache implementada dentro do FPGA é muito pequena, limitada a 64 Kbytes.

4.4 Arquitetura do Acelerador K-Means no FPGA

A arquitetura do gerador é o mapeamento direto do grafo de fluxo de dados do algoritmo de forma a explorar o potencial de paralelismo espacial e temporal (*pipeline*). Além do grafo, as interfaces de entrada e saída são adicionadas, ilustradas na Figura 4.3(b), que acoplam da plataforma de hardware com o código em execução no processador implementado para plataforma HARP v.2 da Intel. A interface pode ser estendida para outras plataformas. As interfaces são baseadas na unidade de hardware apresentada em [38]. Como no HARP v.2, a granularidade da comunicação é limitada a uma linha de cache de 64 bytes, caso um ponto necessite menos de 64 Bytes, a arquitetura explora o paralelismo espacial com replicações, o que possibilita o processamento de vários pontos em paralelo para maximização da vazão de dados. Ademais, para mitigar a latência da comunicação, a interface possibilita o uso de filas e mais de uma instância do acelerador com execuções simultâneas.

Toda a arquitetura é gerada automaticamente de forma parametrizada em função do número de dimensões do conjunto de dados e do número de centroides. O gerador foi desenvolvido em Python com auxílio da biblioteca Veriloggen [52]. O hardware é gerado em Verilog e sintetizado com a ferramenta Intel Quartus para FPGAs. O gerador encapsula toda a complexidade de interface, sincronismo, compilação e síntese para FPGAs e a comunicação do hardware/software. O usuário precisa apenas instanciar os dados a serem processados e chamar ao acelerador a partir de software.

4.5 Gerador de código para GPU

Nesta seção é apresentado o gerador de código para o algoritmo *K-means* para GPU Nvidia em CUDA. O gerador é parametrizável em função do número de dimensões n e do número de centroide k. Cada *thread* é responsável pela classificação de um ponto e executa os seguintes passos: o cálculo da distância em relação a todos os centroides, a redução de soma e a redução para a determinação centroide mais próximo. Para evitar divergência com condicionais em GPU, foram utilizadas condicionais simples. A etapa de reposicionamento também é realizada na GPU.

```
__global__ void kmeans(int*in,
int *c, int *nC,
int *total, const int n) {
  int i;
  i = (blockIdx.x * blockDim.x +
      threadIdx.x) * DIM;
  // leitura do Ponto
  int pd0 = in[i + 0];
  int pd1 = in[i + 1];
  // Map para distancia
  int k0d0 = pd0 - c[0]; // C0_0
  int k0d1 = pd1 - c[1]; // C0_1
  int k1d0 = pd0 - c[2]; // C1_0
  int k1d1 = pd1 - c[3]; // C1_1
  //Quadrado Distancia
  k0d0 *= k0d0; k0d1 *= k0d1;
  k1d0 *= k1d0; k1d1 *= k1d1;
  // Reducao de Soma para Distancia
  k0d0=  k0d0+k0d1;// Distancia de C0
  k1d0=  k1d0+k1d1;// Distancia de C1
  // Reducao de Minimo
  int minId;
  minId = (k1d0 < k0d0) ? 1 : 0;
  // Inclusao do Ponto para
     Reposicionamento
  atomicAdd(&(nC[DIM*minId+0]),pd0);
  atomicAdd(&(nC[DIM*minId+1]),pd1);
  atomicAdd(&(total[minId]), 1);
}
```

Figura 4.4: Código simplificado para *K-means* gerado para GPU com $k = 2$ e $n = 2$ realizando a redução de reposicionamento com operações atômicas.

Ao iniciar o cálculo concorrente de milhares de *threads*, a GPU mascara a latência da leitura na memória e das dependências de dados no nível de instruções de todos os passos da etapa de classificação, incluindo o mapeamento e as duas reduções. O gargalo é a etapa final de reposicionamento que envolve a redução com um acumulador para cada dimensão de cada centroide. Duas versões foram propostas para a fase de reposicionamento dos centroides. Uma baseada nas operações atômicas que possuem suporte em hardware nas novas GPUs (a partir da geração Kepler) e outra baseada em memória compartilhada, com segmentação a partir de blocos e bancos distintos de memória compartilhada em função dos valores de n e k. A Figura 4.4 ilustra o código gerado para o exemplo com duas dimensões e dois centroides na versão mais simples baseada em operações atômicas.

4.6 Experimentos e Resultados

Para avaliar os geradores propostos foi utilizada a base de dados *US Census 1990* [95]. Esta base de dados também foi utilizada em outros trabalhos com FPGA e GPU [96, 97, 93]. O tamanho da palavra utilizado para cada dimensão foi de 16 bits e a base tem 2 milhões de pontos. Para avaliar a variação do desempenho em função do número de centroides e do número de dimensões, utilizamos combinações valores de n e k variando na faixa 2, 4, 8, 16 e 32.

Os resultados foram comparados com a execução em uma CPU Intel Xeon E5-2630 v3 @ 2.40GHz com uma implementação do *K-means* em C, semelhante ao código gerado para GPU, compilado com GCC 4.8.4, com a opção -O3, executando como uma *thread*. O tempo de execução foi medido com a utilização de *chrono::high_resolution_clock*. Além disso, três outras implementações foram utilizadas como referência. A primeira é a implementação popular para execução do *K-means* encontrada no pacote *Scikit-learn* [98]. A segunda é a implementação em OpenMP com 16 *threads* disponibilizada pelo pacote Rodinia, versão 3.1 [99, 100]. A terceira é a implementação em GPU/CUDA também do pacote Rodinia, versão 3.1, executada na GPU K40. O código CPU, utilizado como referência, foi em média 6,5 vezes mais rápido que a implementação *Scikit-learn*. A versão CPU foi em média 3,42 e 6,83 vezes mais lenta que as implementações Rodinia em OpenMP e GPU/CUDA, respectivamente. A plataforma Intel HARP v.2 possui um FPGA Arria 10 modelo 10AX115U3F45E2SGE3. Duas GPUs foram utilizadas, uma Nvidia Tesla K40c com 12GB de memória RAM e uma frequência de relógio de 745 MHz e uma Nvidia Pascal 1080ti com 11 GB e uma frequência de relógio de 1.584 GHz. Todos os experimentos foram conduzidos com 10 repetições e são apresentados os resultados médios.

A Figura 4.5 mostra os resultados do ganho de aceleração agrupados pelo número de dimensões. Pode-se perceber que o ganho de aceleração das GPUs e do FPGA em relação a CPU para conjunto de pontos com a mesma quantidade de dimensões aumenta em função de k, o motivo é o reúso dos dados de $3k-\frac{1}{n}$ operações por byte. Para cada grupo avaliado, as três primeiras barras são os resultados do *scikit-learn*, Rodinia OpenMP e GPU. As três últimas barras são os resultados dos geradores de código propostos para o *K-means*: FPGA, K40 e 1080ti. Os tempos de execução foram normalizados em relação ao tempo de execução da CPU, utilizado como referência. Quanto maior o valor, melhor é o resultado. A linha em negrito com o valor 1 simboliza o tempo de execução na CPU. As GPUs K40 e 1080ti chegam a ser até 54 e 67 vezes mais rápida que a CPU. O FPGA foi até 13,85 mais

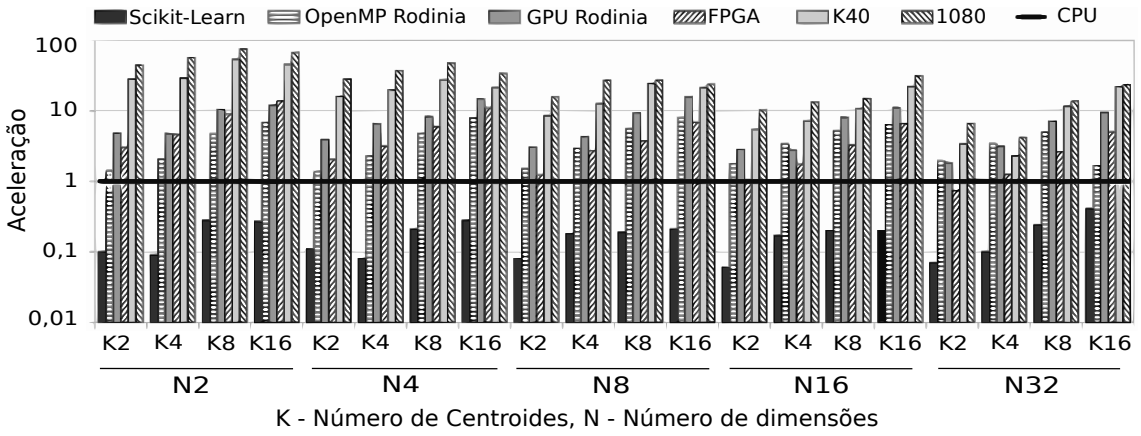


Figura 4.5: Acelerações das GPUs K40 e 1080 e do FPGA normalizadas em relação ao tempo de execução com um *thread* em CPU. Escala logarítmica.

rápido. Pode-se observar que a implementação padrão *scikit-learn* é, em média, 63 vezes mais lenta que o FPGA, 245,5 e 250 mais lenta que as GPUs K40 e 1080ti, respectivamente. Em relação a implementação Rodinia para GPU, os geradores para a K40 e a 1080ti foram, em média, de 3 e 4 vezes mais rápidos, respectivamente. Em relação a versão Rodinia OpenMP com 16 *threads*, os geradores com a K40 e a 1080ti foram, em média, de 6 e 8 vezes mais rápidos.

O maior limitador da plataforma Intel HARP v.2 é a leitura de dados. Como o FPGA não tem memória local, todas as leituras são feitas da memória da CPU a cada iteração. Além disso, para $k = 2$ com 32 dimensões, o conjunto de dados é 16 vezes maior que para 2 dimensões. Na GPU, os dados são transferidos uma vez pelo barramento PCI e permanecem na memória global da GPU até o fim de todas as iterações. A memória global das GPUs tem uma vazão de 200-400 GB/s, o que é significativamente maior que transferência CPU-FPGA, que fica em torno de 16 GB/s. O tempo de execução inclui todas as transferências de dados. É importante ressaltar que o HARP v.2 é um protótipo que pode não refletir o desempenho de sistemas futuros da Intel. Recém lançado em maio de 2018, o processador Intel Xeon 6138P traz integrado um FPGA Arria 10 [101] e pode atingir taxas de transferência de 160 GB/s, 10 vezes mais que o HARP v.2.

O potencial de eficiência energética do FPGA é outro ponto importante. A plataforma HARP v.2 tem um consumo médio de 22,351 Watts aferidos durante as execuções com a ferramenta disponibilizada pela API OPAE da Intel. Como referência para comparação foi utilizada uma estimativa de potência aferida para GPU K40 disponibilizada em [102] de 134 Watts. A GPU 1080ti tem o consumo de 300 Watts. Com base nestes dados, a Figura 4.6 apresenta a eficiência energética

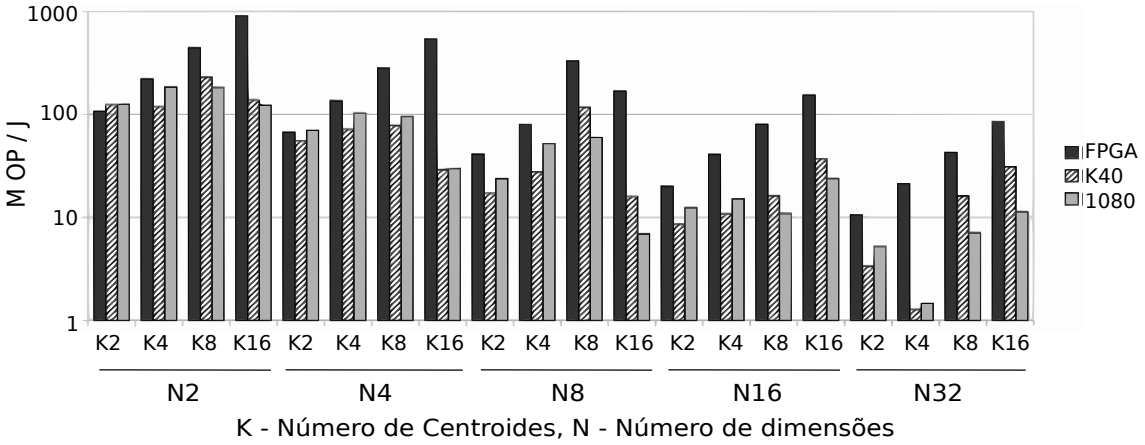


Figura 4.6: Eficiência energética para a execução dos *kernels* em GPU e em FPGA.

como a quantidade de milhões de operações (MOPs) por Joule obtidos em cada experimento para uma iteração nas GPUs e no FPGA. Para um mesmo valor de n, quanto maior o reúso, mais eficiente é a solução. Pode-se observar que a solução com FPGA é promissora, mesmo sendo um protótipo da Intel com limitações de leitura de memória. Este fato também corrobora o interesse das grandes empresas nas soluções com FPGA em nuvem, como por exemplo a Microsoft [66] e a Amazon [64].

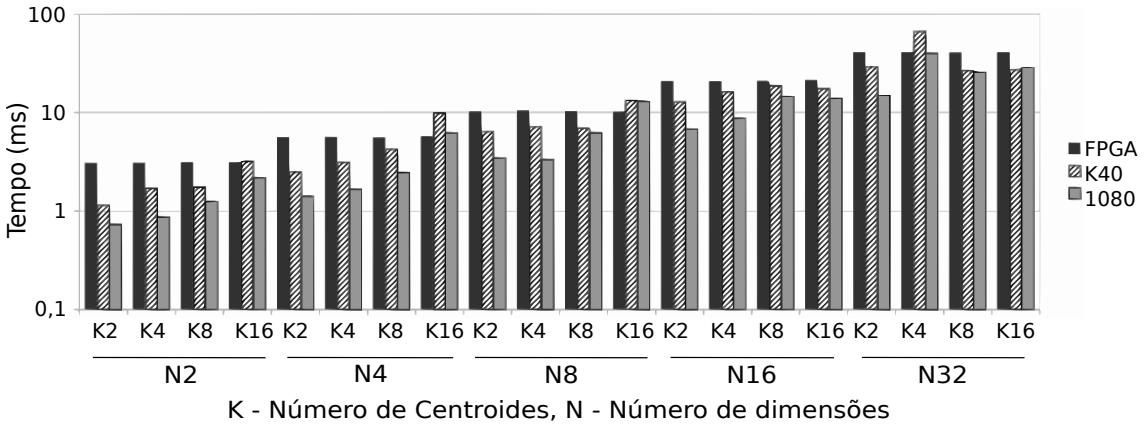


Figura 4.7: Tempos de execução da chamada dos aceleradores.

Como já mencionado, o HARP v.2 é ainda um protótipo. Ao realizar a chamada do acelerador, a sobrecarga de tempo da API Intel domina o tempo de execução. A Figura 4.7 exibe o tempo de execução da chamada do *kernel* do acelerador para cálculo de uma iteração no FPGA incluindo a transferência dos dados, mas não considera o tempo gasto pela API de software da Intel para a realização da inicialização entre as chamadas sucessivas do acelerador (entre as iterações). Em

média, o tempo da execução, incluindo a transferência de dados, é de apenas 30% do tempo da iteração. A sobrecarga das APIs é aproximadamente de 70% do tempo da iteração. Para GPU, a sobrecarga das chamadas é pequena, onde o tempo da execução e da transferência é da ordem de 95% ou mais. Ao desconsiderar a sobrecarga da API, o FPGA foi mais rápido que a GPU K40 em três casos, apesar de ter uma taxa de leitura de memória 13,75 vezes menor e uma frequência de relógio de 200 MHz em comparação com os 745 MHz da K40. Já a GPU 1080ti é mais rápida pois possui uma frequência de relógio de 1,58 GHz e uma taxa de leitura de memória de 485 GB/s.

Vale ressaltar uma diferença da Figura 4.7, na qual o tempo não varia em função de k para um dado n, do comportamento dos resultados anteriores. Nas Figuras 4.5 e 4.6, a eficiência energética e a aceleração aumentam em função de k. Na Figura 4.5, a medida é relativa ao tempo da CPU, portanto apesar do tempo de execução do FPGA não variar para mesmo n, o tempo da CPU varia, aumentando em função de k pois não tira proveito do paralelismo de calcular todas as distâncias ao mesmo tempo. No caso da Figura 4.6, como número de operações aumenta em função de k, a eficiência aumenta, uma vez que o tempo de execução não se altera para um dado n. Ao dobrar o valor de n dobra-se a quantidade de dados. Por exemplo, para 2 milhões de pontos e $n = 2$, tem-se 8 MB de dados e para $n = 4$ teremos 16 MB. Outra observação é a redução da quantidade de cópias do algoritmo por linha de cache com o aumento da quantidade de dimensões. A linha de *cache* do HARP v.2 é de 64 Bytes. Com $n = 2$, cada ponto consome 4 bytes e pode-se instanciar 16 cópias que processam 16 pontos em paralelo. Para $n = 4$, cada ponto terá 8 bytes e tem-se 8 cópias do processo por linha de *cache*.

Ao analisar o aumento de k para um n fixo, pode-se observar que com mais centroides, há um maior reúso dos dados. Para $k = 2$ e $n = 2$, a Figura 4.8(a) mostra que existem 11 operações por ponto e com $k = 4$, na Figura 4.8(b), são 23 operações por ponto. Desse modo, o mesmo ponto será comparado com 4 centroides simultaneamente, o que explora o paralelismo e aumenta o reúso.

A Figura 4.9 apresenta a quantidade de recursos do FPGA gastos em função dos valores de k e n, que é um aspecto favorável aos FPGAs. São apresentadas apenas a quantidade de ALMs (blocos lógicos) e de DSPs (unidades lógica-aritméticas), pois o número de módulos de memória não varia, uma vez que são utilizados apenas pela interface com a CPU. As ALMs são utilizadas para construir a interface, controle, operações de soma e redução de mínimo. Em média, 18-20% das ALMs são utilizads pela interface da Intel e apenas 2-6% das ALMs são gastas pelo gerador para implementar o controle e as operações de soma e redução. O maior valor de k

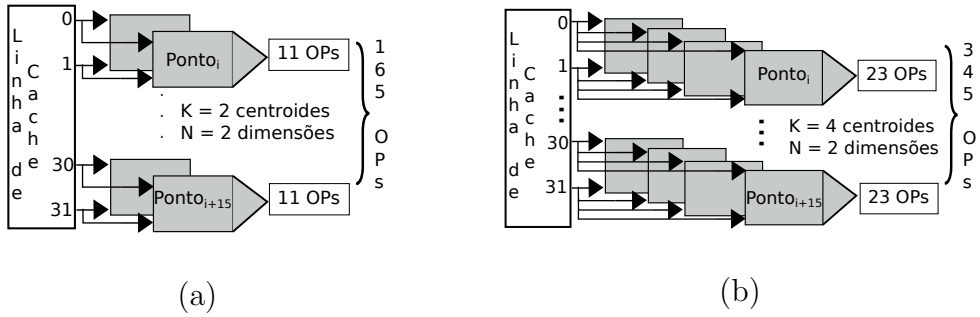


Figura 4.8: Impacto do valor de k: (a) 165 Operações por 64 bytes; (b) 345 ops.

é apenas limitado pela quantidade de DSPs disponíveis no FPGA utilizado. Entretanto, existe uma perspectiva favorável aos FPGAs, pois o novo FPGA Stratix X da Intel possui 6760 DSPs [20] em comparação com os 1518 do Arria 10 do HARP v.2. Além do aumento da quantidade de DSP, a flexibilidade do FPGA permite que futuras extensões do gerador façam uso de aritmética dedicada com ALMs (que são abundantes) para o cálculo do quadrado das distâncias ao invés dos DSPs como multiplicadores. Portanto, com vazão de dados, mais DSPs, o desempenho do FPGA para o *K-means* pode aumentar significativamente.

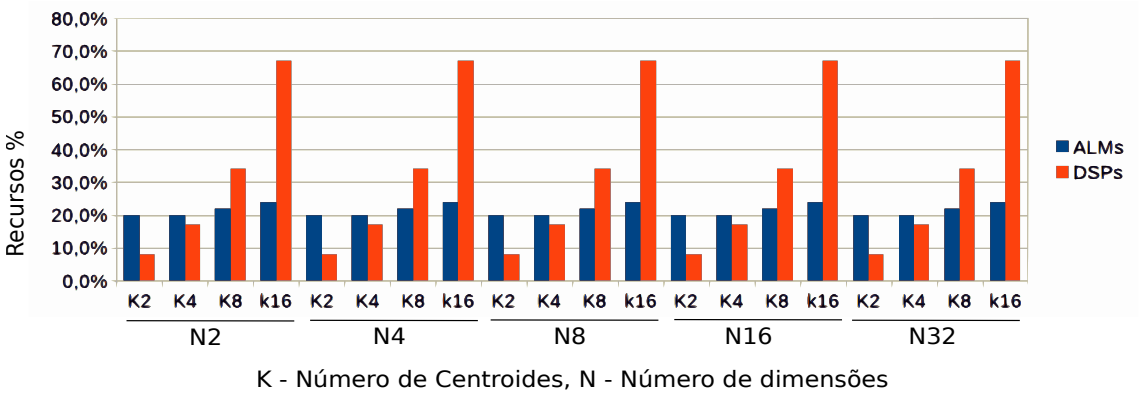


Figura 4.9: Recursos do FPGA utilizados pelos aceleradores.

4.7 Trabalhos Relacionados

Esta seção apresenta um estudo das implementações do *K-means* em FPGAs e GPUs. Um gerador parametrizável em função do número de dimensões e centroides para FPGA foi proposto em [103]. Entretanto, o artigo não apresenta resultados de execução, apenas estimativas e simulação. É importante ressaltar que uma descrição que pode ser simulada em FPGA, não necessariamente irá executar corretamente.

Além disso, o tempo de sincronização, chamada, leitura e escrita de dados pode não estar incluso. Uma estimativa para execução do *K-means* em FPGA é apresentada em [104], também sem resultados concretos com execução. Outra implementação em uma plataforma CPU/FPGA Xilinx é apresentada em [89]. A aceleração é estimada em função do potencial de aceleração do FPGA e da CPU, inclusive a comunicação, o que mostra um potencial de aceleração de até 150x para o *K-means*, porém o sistema não foi avaliado em tempo de execução. Uma abordagem em alto nível para FPGA com a utilização da linguagem OpenCL é apresentada em [90], na qual a implementação utiliza uma função para a classificação e outra para o reposicionamento dos centroides. A classificação utiliza a distância de Manhattan no lugar da distância Euclidiana. Esta proposta apresenta melhora de desempenho apenas para valores grandes de k, maiores que 128. Em [105], um acelerador para FPGA foi implementado, onde os parâmetros k,n e número de iterações podem ser alterados em tempo de execução. Porém o formato dos dados é fixo (64 bits), o valor máximo está limitado a apenas 16K pontos e foi apenas prototipado em uma placa de baixa custo com cálculo paralelo de apenas 2 pontos por vez, o que restringe seu uso em problemas reais. Para $k = 8$ e $n = 4$, o desempenho é de apenas 6M operações/s a 50 MHz, o gerador proposto neste trabalho executa 9,7G operações/s à 200 MHz, ou seja, 250 vezes mais rápido que o acelerador proposto por [105] e não está limitado a 16K entradas.

Como não existe uma padronização, é difícil realizar uma comparação entre as implementações. A comparação será feita através de bases de dados de mesmo tamanho e considerando o tempo de uma iteração no qual o número de operações realizadas é equivalente. A arquitetura FPGA proposta em [96] é baseada no paradigma *Map-Reduce* onde a avaliação foi feita com *benchmarks* de apenas 2 e 4 dimensões para $k = 4$. Ao comparar com os resultados apresentados na Seção 4.6, o gerador proposto neste trabalho apresenta tempos de execução duas vezes mais rápidos que resultados de [96] para $n = 2$ e 4. Ademais, o gerador é genérico e não foi dedicado a dois valores de n. Uma arquitetura com elementos de processamento para o *K-means* em FPGA foi proposta em [106], que é pelo menos duas vezes mais lenta que o gerador proposto neste trabalho para $k = 3$ e $n = 10$. O trabalho apresentado em [93] é o mais próximo da abordagem proposta em termos da plataforma de FPGA. Os resultados também são avaliados em tempo de execução. O FPGA é o protótipo da HARP v.1, já descontinuado. O FPGA é utilizado para calcular a etapa de classificação e a CPU calcula o reposicionamento. Apenas $k = 8$,$n = 2$ e $k = 4$,$n = 4$ são avaliados para os quais a abordagem proposta neste trabalho em FPGA é 20 vezes ou mais rápida que o resultado apresentado em [93]. Uma

biblioteca para GPU é apresentada em [107], porém os resultados são pelo menos 10 vezes mais lentos da solução em FPGA e na GPU K40 apresentada nesse trabalho. Em relação a implementação proposta para GPU 1080ti, os resultados do trabalho proposto em [107] foram 50 vezes mais lentos. O maior problema é que parte da classificação e reposicionamento são implementados na CPU. Uma comparação entre as soluções com CPU, GPU e FPGA foi apresentada em [108], onde a solução com FPGA foi a mais rápida executando na Nuvem da Amazon [64]. A solução apresentada por este trabalho em FPGA é genérica com resultados equivalentes em desempenho considerando $k = 8$ e $n = 5$ aos apresentados para uma abordagem específica em [108]. Como já apresentado na Figura 4.5, a implementação Rodinia para GPU [100] foi de 3 à 4 vezes mais lenta que o gerador proposto neste trabalho.

4.8 Conclusão

Este trabalho apresenta um gerador parametrizável do algoritmo *K-means* para FPGAs e GPUs. O desempenho é maximizado ao se especializar em um domínio específico e explorar o paralelismo espacial e temporal. Apesar de ser específico, existe uma demanda por agrupamento de dados, onde o *K-means* e suas variações têm muitas aplicações. Os resultados experimentais mostraram ganho em tempo de execução em comparação a CPU de até 54,55 e 13,85 vezes para as GPUs k40 e 1080ti e a plataforma HARP v.2, respectivamente. O gerador executou com uma frequência de 200 MHz no FPGA, mas pode ser ajustado para executar a 400 MHz, que dobrará seu desempenho. Apesar do desempenho do FPGA ter sido limitado pelas baixas taxas de transmissão de 16 GB/s na leitura de memória do protótipo HARP v.2 da Intel, existe um potencial para os novas plataformas FPGAs da Intel e da Xilinx onde o desempenho da transferência de dados deve melhorar significativamente.

Como trabalhos futuros, há dois aspectos importantes do algoritmo *K-means* que podem ser explorados: a determinação da quantidade de centroides e a seleção de um subconjunto relevante de atributos. Essas duas funcionalidades podem ser adicionadas de forma eficiente às implementações aqui propostas, tanto em GPU quanto FPGA, uma vez aumentarão ainda mais o reúso de dados. Outra possibilidade é a computação aproximada que vem sendo explorada em vários algoritmos de aprendizado de máquina. Pode-se esperar que a flexibilidade do FPGAs tenha um nicho melhor neste domínio que as GPUs e as CPUs. Para a nova geração *Volta* de GPUs da Nvidia [109], existe ainda um potencial a ser explorado pelo *K-means* com os operadores TPU que são capazes de multiplicar e acumular matrizes com

estruturas sistólicas. Com relação às CPUs, o *K-means* pode explorar paralelismo a nível dos múltiplos núcleos e também o paralelismo no nível de instrução com as extensões vetoriais AVX.

Capítulo 5

Conclusão

Os resultados apresentados nesta dissertação mostram que o uso de FPGAs em alto desempenho é promissor, e um dos principais desafios é o desenvolvimento de ferramentas. Uma nova ferramenta foi desenvolvida neste trabalho para projeto de algoritmos com a modelagem por grafos de fluxo de dados tendo como alvo a geração automática do hardware e software para execução em uma plataforma com um FPGA integrado a um processador com múltiplos núcleos através de memória compartilhada.

O projeto da Intel denominado HARP tem se mostrado uma boa iniciativa para a computação de alto desempenho devido a possibilidade de construção de aceleradores específicos para aplicações a nível de hardware. A possibilidade da utilização de diferentes comprimentos de palavras de dados para o processamento e o aumento de DSPs inseridos são vantagem que permitem a criação de circuitos maiores em menor espaço. Com o lançamento da segunda versão do protótipo HARP, conseguiu-se um aumento na taxa de transferência de dados, o que colocou o HARP em uma posição mais competitiva que pode ainda ser explorada com o produto lançado integrado ao novo processador Intel Xeon 6138P no qual as taxas de transferência saltam para 160GBps, cerca de dez vezes maior que o conseguido pelo protótipo.

O ADD permite um meio termo entre a construção de *dataflows* (DFG) em linguagens de descrição de hardware e as linguagens de alto nível como OpenCL. Esta característica permite ao desenvolvedor um maior controle do circuito gerado e seu funcionamento além de possibilitar a simulação e depuração de modo visual. A primeira versão da ferramenta trouxe como base o desenvolvimento com operadores síncronos em um banco de operadores já construídos. Esses componentes podem ser aprimorados e novos desenvolvidos com base nos já existentes. O ADD trouxe

também uma interface gráfica que permite a edição e simulação e uma API que permite a execução dos aceleradores no HARP e em placas com FPGA didáticas de modo semelhante, sem exigir o conhecimento prévio dos ambientes que forem utilizados pelo usuário.

Com o desenvolvimento da nova versão, melhorias foram acrescentadas no funcionamento dos componentes com o acréscimo de protocolos de comunicação entre os operadores o que possibilitou a execução de novos algoritmos como o *Knn* e o *genetic regulatory network* (GRN). Melhorias na criação da interface de *hardware* e *software* possibilitaram a criação e execução de diferentes DFGs em um mesmo FPGA tanto em placas didáticas quanto no HARP. Além dos acréscimos citados, o suporte a operadores assíncronos foi incluído como opção de desenvolvimento de DFGs.

Em continuação na linha dos DFGs, foi proposto geradores automático para o algoritmo *K-means* para a execução no HARP v.2 e em GPU com o objetivo de comparação das execuções. A geração de código para GPU foi feita com base na execução da classificação de cada ponto e acúmulo dos valores para o cálculo dos centroides em *threads* separadas. O dos cálculo de centroides e as condições de parada do algoritmo ficaram por conta da CPU. Para o HARP, o circuito gerado realiza apenas a classificação dos dados, cabendo a CPU o acúmulo e cálculo dos novos valores dos centroides. Conforme a análise dos resultados dos experimentos, foi possível observar as possibilidades promissoras da utilização de FPGAs para a execução desse algoritmo tanto quanto o poder de execução da GPU para o mesmo.

Com a criação de novos protótipos e o desenvolvimento de novos FPGAs, a utilização de FPGAs em aceleradores tende a ser maior. Os FPGAs atuais possuem uma quantidade pequena de memória interna (alguns mega Bytes), com o aumento da quantidade de memória interna ou o acréscimo de bancos de memória para o acesso direto do FPGA, como no *Catapult* da *Microsoft*, pode-se aumentar conseguir melhores desempenhos e um aumento na utilização de FPGAs em aceleradores.

Para trabalhos futuros com aceleradores em FPGA, uma gama de possíbilidades podem ser exploradas como a criação de CGRAs para aplicações específicas nas quais DFGs podem ser mapeados e executados sem a necessidade de uma nova síntese para cada novo circuito, o que reduz o tempo de desenvolvimento. A possibilidade da utilização de computação aproximada que tem sido utilizada para o treinamento de redes neurais pode ser explorada em FPGA por possibilitar uma menor utilização de recursos e palavras de dados menores. A utilização de bancos de memória internas para a criação de acumuladores é uma opção para a execução do *K-means* completo internamente no FPGA.

Referências Bibliográficas

[1] ACM in *ACM Award 2017* *https://amturing.acm.org/*, 2018.

[2] G. E. Moore, "Cramming More Components Onto Integrated Circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp. 114 ff.," *IEEE solid-state circuits society newsletter*, vol. 20, no. 3, pp. 33–35, 2006.

[3] ACM, "Turing (lecture at isca 2018)." `https://www.acm.org/hennessy-patterson-turing-lecture`, 2018. Accessed: 2018-06-17.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitative Approach*. Elsevier, 2011.

[5] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of Ion-Implanted MOSFET's With Very Small Physical Dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[6] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, ACM, 1967.

[7] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[8] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[9] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA Tensor Core Programmability, Performance & Precision," *arXiv preprint arXiv:1803.04014*, 2018.

[10] Microsoft, "Clouds, Catapults and Life After the End of Moore's Law with Dr. Doug Burger." `https://www.microsoft.com/en-us/research/blog/`

clouds-catapults-life-end-moores-law-dr-doug-burger/, 2018. Accessed: 2018-06-17.

[11] J. Mora, A. Otero, E. de la Torre, and T. Riesgo, "Fast and Compact Evolvable Systolic Arrays on Dynamically Reconfigurable FPGAs," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2015 10th International Symposium on*, pp. 1–7, IEEE, 2015.

[12] G. Stitt, "Are Field-Programmable Gate Arrays Ready for the Mainstream?," *IEEE Micro*, vol. 31, no. 6, pp. pp.58–63, 2011.

[13] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, *et al.*, "Firesim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pp. 29–42, IEEE Press, 2018.

[14] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, *et al.*, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," 2018.

[15] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable Computing Architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, 2015.

[16] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms," in *Proceedings of the 53rd Annual Design Automation Conference*, p. 109, ACM, 2016.

[17] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, *et al.*, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, 2015.

[18] S. Hauck and A. DeHon, *Reconfigurable Computing: the Theory and Practice of FPGA-Based Computation*, vol. 1. Morgan Kaufmann, 2010.

[19] I. Corp, "Device Overview," 2018. Accessed: 2018-07-14.

[20] Stratix in *Device Overview https://www.altera.com/products/fpga/stratix-series/*, 2018.

[21] Alpha, "Alpha Data ADM-PCIE-7V3 Datasheet." http://www.alpha-data.com/pdfs/adm-pcie-7v3.pdf, 2016. Accessed: 2018-06-14.

[22] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, *et al.*, "A Reconfigurable Computing System Based on a Cache-Coherent Fabric," in *ReConFig*, pp. 80–85, 2011.

[23] Intel, "fpga sdk for opencl."

[24] Xilinx, "SDSoC Development Environment Help." https://www.xilinx.com/html_docs/xilinx2018_1/sdsoc_doc/index.html, 2018. Accessed: 2018-07-16.

[25] OpenSPL, "The Open Spatial Programming Language." http://www.openspl.org/, 2018. Accessed: 2018-07-16.

[26] D. Koch, F. Hannig, and D. Ziener, *FPGAs for Software Programmers*. Springer, 2016.

[27] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[28] UFV, Conselho Técnico de Pós Graduação da Universidade Federal de Viçosa, "Normas de redação de teses e dissertações." http://www.dpi.ufv.br/arquivos/ppgcc/doc/PPG-2015-normascorrigidas.pdf, 2018. Accessed: 2018-06-14.

[29] J. Penha, L. Bragança, D. Almeida, J. Nacif, and R. Ferreira, "ADD - Uma Ferramenta de Projeto de Aceleradores com DataFlow para Alto Desempenho," *XVIII Simpósio em Sistemas Computacionais de Alto Desempenho - WSCAD*, 2017.

[30] WSCAD, "Simpósio em Sistemas Computacionais de Alto Desempenho." http://wscad.sbc.org.br/current/index.html, 2018. Accessed: 2018-06-15.

[31] I. Corp, "Intel® QuickAssist Technology Accelerator Abstraction Layer QuickPath Interconnect-FPGA Release 4.1.7 Software Release Notes," *Intel Corporation*, 2015.

[32] CCPE, "Concurrency and Computation Practice and Experience." https://onlinelibrary.wiley.com/journal/15320634, 2018. Accessed: 2018-06-15.

[33] I. Corp, "Simplify Software Integration for FPGA Accelerators with OPAE," *Intel Corporation*, 2017.

[34] OpenMP, "The OpenMP API Specification for Parallel Programming." `https://www.openmp.org/`, 2018. Accessed: 2018-06-15.

[35] J. C. Penha, G. Fontes, and R. Ferreira, "MIPSFPGA - Um Simulador MIPS Incremental com Validação em FPGA," *International Journal in Computer Architecture Education (IJCAE)*, vol. 5, no. 1, pp. 19–25, 2016.

[36] IJCAE, "International Journal of Computer Architecture Education." `http://www2.sbc.org.br/ceacpad/ijcae/index.html`, 2018. Accessed: 2018-06-14.

[37] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: the Hardware/Software Interface*. Morgan Kaufmann, 2013.

[38] L. Bragança, F. Alves, J. C. Penha, G. Coimbra, R. Ferreira, and J. A. M. Nacif, "Simplifying HW/SW Integration to Deploy Multiple Accelerators for CPU-FPGA Heterogeneous Platforms," in *IEEE SAMOS*, 2018.

[39] I. Corp, "Intel FPGA Basic Building Blocks (BBB)," *Intel Corporation*, 2018. Accessed: 2018-07-14.

[40] SAMOS, "International Conference on Embendded Computer Systems: Architectures, Modeling and Simulation." `https://samos-conference.com/`, 2018. Accessed: 2018-06-14.

[41] G. Andrade, W. de Carvalho, R. Utsch, P. Caldeira, A. Alburquerque, F. Ferracioli, L. Rocha, M. Frank, D. Guedes, and R. Ferreira, "ParallelME: A Parallel Mobile Engine to Explore Heterogeneity in Mobile Computing Architectures," in *European Conference on Parallel Processing*, pp. 447–459, Springer, 2016.

[42] A. Munshi, "The opencl specification," in *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pp. 1–314, IEEE, 2009.

[43] OpenCL, "OpenCL™ Zone – Accelerate Your Applications." `http://developer.amd.com/tools-and-sdks/opencl-zone/`, 2017. Accessed: 2017-08-09.

[44] I. Corp, "Intel FPGA - Acceleratiing The Smart and Connected World." `https://www.altera.com/`, 2017. Accessed: 2017-08-09.

[45] Xilinx, "Xilinx – All Programmable." `https://www.xilinx.com/`, 2017. Accessed: 2017-08-09.

[46] E. Kim, K. Kim, and H. P. In, "A Multi-View API Impact Analysis for OpenSPL Platform," in *Advanced Communication Technology (ICACT), 2010 The 12th International Conference on*, vol. 1, pp. 686–691, IEEE, 2010.

[47] N. Hendrich, "A Java-Based Framework for Simulation and Teaching: HADES-the Hamburg Design System," in *Microelectronics Education*, pp. 285–288, Springer, 2000.

[48] A. C. Ling, U. Aydonat, S. O'Connell, D. Capalija, and G. R. Chiu, "Creating High Performance Applications with Intel's FPGA OpenCL™ SDK," in *Proceedings of the 5th International Workshop on OpenCL*, p. 11, ACM, 2017.

[49] J. Winans, "On Dataflow Computing With OpenSPL," 2015.

[50] P. Gupta, "Accelerating Datacenter Workloads," in *Field Programmable Logic and Applications, Keynote - Slides available at www.fpl2016.org*, 2016.

[51] P. K. Gupta, "Xeon+ FPGA Platform for the Data Center," in *Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, vol. 119, 2015.

[52] S. Takamaeda-Yamazaki, "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL," in *Applied Reconfigurable Computing*, Apr 2015.

[53] R. Ferreira, J. Nacif, S. Magalhaes, T. de Almeida, and R. Pacifico, "Be a Simulator Developer and Go Beyond in Computing Engineering," in *Frontiers in Education Conference (FIE)*, IEEE, 2015.

[54] R. Ferreira, J. M. Cardoso, A. Toledo, and H. C. Neto, "Data-Driven Regular Reconfigurable Arrays: Design Space Exploration and Mapping," Int. Conf. on Embedded Computer Systems Architectures, Modeling and Simulation SAMOS, 2005.

[55] R. Ferreira, J. M. Cardoso, and H. C. Neto, "An Environment for Exploring Data-Driven Architectures," Int. Conference on Field-Programmable Logic and Applications (FPL), 2004.

[56] P. Marwedel, K. Cong, and S. Schwenk, "RaVi: Interactive Visualization of Information System Dynamics Using a Java-Based Schematic Editor and Simulator," 2002.

[57] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, *et al.*, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on, pp. pp.13–24,, IEEE, 2014.

[58] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms," Design Automation Conference (DAC), ACM/EEE, 2016.

[59] R. Stewart, D. Bhowmik, A. Wallace, and G. Michaelson, "Profile Guided Dataflow Transformation for FPGAs and CPUs," *Journal of Signal Processing Systems*, vol. 87, no. 1, pp. 3–20, 2017.

[60] L. Barford, S. S. Bhattacharyya, and Y. Liu, "Data Flow Algorithms for Processors With Vector Extensions: Handling Actors With Internal State," in *Signal and Information Processing (GlobalSIP), 2014 IEEE Global Conference on*, pp. 20–24, IEEE, 2014.

[61] Z. Ul-Abdin and M. Yang, "A Radar Signal Processing Case Study for Dataflow Programming of Manycores," *Journal of Signal Processing Systems*, vol. 87, no. 1, pp. 49–62, 2017.

[62] F. Palumbo, T. Fanni, C. Sau, and P. Meloni, "Power-Awarness in Coarse-Grained Reconfigurable Multi-Functional Architectures: a Dataflow Based Strategy," *Journal of Signal Processing Systems*, vol. 87, no. 1, pp. 81–106, 2017.

[63] P. Colangelo, E. Luebbers, R. Huang, M. Margala, and K. Nealis, "Application of Convolutional Neural Networks on Intel/ Xeon Processor with Integrated FPGA," 2017 IEEE High Performance Extreme Computing Conference (HPEC), 2017.

[64] A. W. Services, "EC2: Elastic Compute Cloud." `http://aws.amazon.com/ec2/`.

[65] L. D. Tucci, M. Rabozzi, L. Stornaiuolo, and M. D. Santambrogio, "The Role of CAD Frameworks in Heterogeneous FPGA-Based Cloud Systems," 2017 IEEE International Conference on Computer Design (ICCD), 2017.

[66] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A Cloud-Scale Acceleration Architecture," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016.

[67] T. Becker, O. Mencer, and G. Gaydadjiev, *Spatial Programming with OpenSPL*, pp. pp.81–95. Springer International Publishing, 2016.

[68] Xilinx, "Cloud Acceleration for RTL, C/C++, and OpenCL." `https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`.

[69] W. J. Dally, "Challenges for Future Ccomputing Systems," 2015.

[70] E. LUEBBERS, "OPAE." `https://01.org/OPAE`, 2017. Intel, 01.org, Nov.

[71] C. Wang, W. Lou, L. Gong, L. Jin, L. Tan, Y. Hu, X. Li, and X. Zhou, "Reconfigurable Hardware Accelerators: Opportunities, Trends, and Challenges," *CoRR*, vol. abs/1712.04771, 2017.

[72] A. Iordache, G. Pierre, P. Sanders, J. G. de F. Coutinho, and M. Stillwell, "High Performance in the Cloud with FPGA Groups," Proceedings of the 9th International Conference on Utility and Cloud Computing, (New York, NY, USA), ACM, 2016.

[73] H. R. Zohouri, A. Podobas, and S. Matsuoka, "Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL," Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2018.

[74] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-Dataflow Acceleration," Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA, ACM, 2017.

[75] Z. Ruan, T. He, B. Li, P. Zhou, and J. Cong, "ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA," The 26th IEEE International Symposium on Field-Programmable Custom Computing Machines, ACM, 2018.

[76] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A Reconfigurable Architecture For Parallel Paterns," Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. pp.389–402,, ACM, 2017.

[77] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, "A Customizable Matrix Multiplication Framework for the Intel HARPv2 Xeon+FPGA Platform: A Deep Learning Case Study," Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2018.

[78] B. Darvish Rouhani, M. Ghasemzadeh, and F. Koushanfar, "Causalearn: Automated Framework for Scalable Streaming-Based Causal Bayesian Learning Using FPGAs," Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2018.

[79] L. B. da Silva, D. Almeida, J. A. M. Nacif, I. Sanchez-Osorio, C. A. Hernandez-Martinez, and R. Ferreira, "Exploring the Dynamics of Large-Scale Gene Regulatory Networks Using Hardware Acceleration on a Heterogeneous CPU-FPGA Platform," 2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2017.

[80] G. Stitt, A. Gupta, M. N. Emas, D. Wilson, and A. Baylis, "Scalable Window Generation for the Intel Broadwell+Arria 10 and High-Bandwidth FPGA Systems," Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2018.

[81] T. Oguntebi and K. Olukotun, "Graphops: A Dataflow Library for Graph Analytics Acceleration," Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. pp.111–117,, ACM, 2016.

[82] Z. Li, L. Liu, Y. Deng, S. Yin, Y. Wang, and S. Wei, "Aggressive Pipelining of Irregular Applications on Reconfigurable Hardware," Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. pp.575–586,, ACM, 2017.

[83] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes, "SPar: A DSL for High-Level and Productive Stream Parallelism," *Parallel Processing Letters*, vol. v.27,, no. n.01,, pp. pp.1740005,, 2017.

[84] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright, "Pushing the Limits of Accelerator Efficiency While Retaining Programmability," 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016.

[85] J. Penha, "Suplementary Material." `https://bitbucket.org/jeronimopenha/add_benchmarks/src/master/`, 2018.

[86] V. Venugopal and D. M. Shila, "High Throughput Implementations of Cryptography Algorithms on GPU and FPGA," 2013 IEEE International Instrumentation and Measurement Technology Conference (I2MTC), 2013.

[87] M. Purandare, R. Polig, and C. Hagleitner, "Accelerated Analysis of Boolean Gene Regulatory Networks," 2017 27th International Conference on Field Programmable Logic and Applications (FPL), 2017.

[88] Q. Jia and H. Zhou, "Tuning Stencil Codes in OpenCL for FPGAs," in *Computer Design (ICCD), 2016 IEEE 34th International Conference on*, pp. pp.249–256,, IEEE, 2016.

[89] K. Neshatpour *et al.*, "Energy-Efficient Acceleration of Big Data Analytics Applications Using FPGA," in *IEEE International Conference on Big Data*, 2015.

[90] Q. Y. Tang and M. A. Khalid, "Acceleration of K-Means Algorithm Using Altera SDK for OpenCL," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 10, no. 1, p. 6, 2016.

[91] E. W. Forgy, "Cluster Analysis of Multivariate Data: Efficiency Versus Interpretability of Classifications," *Biometrics*, vol. 21, pp. 768–769, 1965.

[92] Y.-M. Choi and H. K.-H. So, "Map-Reduce Processing of K-Means Algorithm with FPGA-Accelerated Computer Cluster," in *IEEE International Conference ASAP*, 2014.

[93] T. S. Abdelrahman, "Accelerating K-Means Clustering on a Tightly-Coupled Processor-FPGA Heterogeneous System," in *IEEE International Conference ASAP*, 2016.

[94] M. J. Zaki and W. Meira Jr, *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, 2014.

[95] D. Dheeru and E. Karra Taniskidou in *UCI Machine Learning Repository http://archive.ics.uci.edu/ml*, 2017.

[96] Z. Li, J. Jin, and L. Wang, "High-Performance K-Means Implementation Based on a Simplified Map-Reduce Architecture," *arXiv preprint arXiv:1610.05601*, 2016.

[97] R. Kaplan, L. Yavits, and R. Ginosar, "Prins: Processing-in-Storage Acceleration of Machine Learning," *IEEE Transactions on Nanotechnology*, 2018.

[98] F. Pedregosa, G. Varoquaux, A. Gramfort, Michel, *et al.*, "Scikit-Learn: Machine Learning in Python," *Journal of machine learning research*, vol. 12, 2011.

[99] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE Int. Symposium on Workload Characterization*, 2009.

[100] Rodinia in *Version 3.1* `http://lava.cs.virginia.edu/Rodinia/download_links.htm`, 2016.

[101] Intel in *Introducing the Intel Xeon Scalable processor with integrated Arria 10 FPGA https://itpeernetwork.intel.com/intel-processors-fpga-better-together/*, 2018.

[102] M. Ferro *et al.*, "Analysis of GPU Power Consumption Using Internal Sensors," in *Workshop em Desempenho de Sistemas Computacionais e de Comunicacao*, 2017.

[103] A. Amaricai, "Design Trade-offs in Configurable FPGA Architectures for K-Means Clustering," *Studies in Informatics and Control*, vol. 26, no. 1, pp. 43–48, 2017.

[104] K. Neshatpour *et al.*, "Big Biomedical Image Processing Hardware Acceleration: A Case Study for K-Means and Image Filtering," in *Int. Symposium on Circuits and Systems*, 2016.

[105] L. A. Maciel, M. A. Souza, and H. C. de Freitas, "Projeto e Avaliação de uma Arquitetura do Algoritmo de Clusterização k-means em vhdl e fpga," *WSCAD*, 2017.

[106] D. Lee, A. Althoff, D. Richmond, and R. Kastner, "A Streaming Clustering Approach Using a Heterogeneous System for Big Data Analysis," in *IEEE/ACM ICCAD*, 2017.

[107] J. Bhimani, M. Leeser, and N. Mi, "Accelerating K-Means Clustering with Parallel Implementations and GPU Computing," in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, 2015.

[108] K. Huang, "K-Means Parallelism on FPGA," Master's thesis, Northeastern University, Boston, USA, 2017.

[109] N. P. Jouppi, , *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *International Symposium on Computer Architecture (ISCA)*, 2017.

Apêndice A

MIPSFPGA - Um Simulador MIPS Incremental com Validação em FPGA

MIPSFPGA - Um Simulador MIPS Incremental com Validação em FPGA

Jeronimo Costa Penha, Geraldo Fontes, Ricardo Ferreira
Universidade Federal de Viçosa, Viçosa - MG CEP 36570-900
Departamento de Infomática
jeronimopenha@gmail.com, gsfj03@gmail.com, ricardo@ufv.br

Abstract

Este artigo apresenta inovações no ensino da arquitetura do processador MIPS com apoio de um simulador gráfico. O ambiente proposto, denominado MIPSFPGA (Mips Incremental Processor Simulator and Fpga Prototyping), além da interface gráfica para visualização do caminho de dados das diversas implementações do MIPS com e sem pipeline, o MIPSFPGA inclui vários recursos adicionais. Primeiro, o projeto pode ser editado graficamente para explorar outras implementações. Todos os projetos do livro Organização de Computadores de Patterson e Hennessy são disponibilizados. O projeto pode ser exportado e prototipado em FPGA. As implementações podem ser depuradas durante a simulação e a prototipação. Finalmente, a metodologia é incremental, permitindo começar com uma simples apresentação dos exemplos até a personalização e derivação de novas implementações e extensões.

1. Introdução

O MIPS é um processador que representa um marco de inovação na introdução da arquitetura RISC e na consolidação de ideias simples e regulares: poucas instruções, decodificação direta, registradores gerais, pipeline, load/store como unidade de execução. O livro "*Computer Organization and Design*" [13] de David A. Patterson e John L. Hennessy é uma das referências mais usadas e está na lista dos 10 mais vendidos da Amazon na área de *Hardware*. Apesar de vários simuladores já terem sido propostos para apoio ao ensino com o MIPS [3], este trabalho apresenta uma nova abordagem para uma ferramenta de simulação e apoio ao ensino de arquiteturas RISC com foco no processador MIPS. A referência básica são os exemplos do livro de Patterson&Hennessy [13].

Dentre os diversos simuladores propostos para a arquitetura MIPS, pode-se destacar o SPIM [11] e o MARS [15]. O SPIM é um simulador para o conjunto de instruções MIPS32. É capaz de executar programas escritos em *assembly*, mostrar o conteúdo do banco de registradores e da memória. O SPIM é multiplataforma (Windows, Linux Mac-OS). Apesar de ser indicado pelo livro de Patterson&Hennessy [13] como material de apoio, este simulador não detalha a implementação do processador e não mostra o caminho de dados do processador [11]. O simulador MARS é o simulador MIPS mais popular, desenvolvido em Java, semelhante ao SPIM, o MARS trabalha no nível *assembly*. É bem documentado e pode ser estendido, como por exemplo uma versão recente apresentada [1] para a simulação do MIPS Vetorial. Assim como o simulador SPIM, o MARS não detalha a implementação [15].

Para compreender os detalhes de implementação do caminho de dados do MIPS, vários simuladores foram propostos como por exemplo o ViSiMIPS [10], RAVI [12], WEBMIPS [2] e DIMIPSS [5]. Estes simuladores mostram com uma interface gráfica simples e didática a visualização da execução de um trecho de código MIPS. O usuário pode acompanhar como os encaminhamentos foram executados, visualizar com cores as instruções passo a passo no pipeline. Porém nenhum deles permite que o estudante modifique a implementação como por exemplo adicionando o tratamento de uma nova instrução ou modificando a implementação com a inclusão de multiplexadores e registradores.

A abordagem proposta aqui também é baseada em um simulador com interface gráfica para visualização do caminho de dados. Além disso, nossa proposta segue a sequência didática do livro de Patterson&Hennessy [13], na qual os autores propõem o ensino MIPS em partes e de forma incremental. O ambiente proposto é o MIPSFPGA - *Mips Incremental Processor Simulator and Fpga Prototyping* - que é baseado no simulador Hades [9] como camada básica. Todas as implementações do livro estão disponibilizadas e o estudante pode fazer alterações no projeto no nível RTL (*Register Transpfer Level*). O MIPSFPGA contribui com novas possibilidades: 1) alteração do projeto para o desenvolvimento de novas versões ou a resolução de exercícios propostos pelo livro; 2) exportar o projeto para implementação física em FPGA; 3) Depuração do

conteúdo dos registradores e memória na simulação e na prototipação; 4) Desenvolvimento de novos componentes ampliando a biblioteca do simulador.

2. Simulador Incremental

O ambiente MIPSFPGA proposto aqui é composto por um conjunto de projetos e componentes que executam no simulador Hades. O Hades é implementado em Java para simulação de circuitos digitais em diversos níveis e foi selecionado pela sua portablidade, flexibilidade para a criação de novos componentes, extensão, múltiplas finalidades, além de possuir uma ampla biblioteca de componentes RTL [9].

2.1. HADES

Proposto em 1998 por Norman Hendrich, o simulador Hades [9] é um simulador de eventos com interface gráfica. Sua finalidade inicial é a simulação de circuitos digitais em vários níveis (portas, RTL, processadores, etc.). Todos os componentes e conexões (fios) são objetos e graças a uma definição simples, o ambiente é bem genérico e eficiente, podendo ser utilizado para outras áreas como processamento de imagens, computação paralela, sistemas embarcados, etc [8]. O Hades é um simulador dinâmico no qual a edição do projeto pode ser feita mesmo com circuito em modo de simulação. A biblioteca de componentes do Hades possui mais de 500 componentes. Além disso, o usuário pode desenvolver seus próprios componentes, fazendo reuso dos componentes da biblioteca. O desenvolvimento das extensões tem uma rápida curva de aprendizado. Várias extensões já foram propostas para o Hades como por exemplo, a implementação de fluxo de dados [7, 6].

Para a simulação e o ensino do MIPS, existe também o simulador RaVi [12] que é baseado no Hades. Porém o RaVi é voltado para a visualização, tem sua implementação encapsulada e não está disponível para extensões. Nossa proposta é criar projetos e componentes de forma colaborativa permitindo o reuso e extensões. As abordagens [14, 8] também são baseadas no ensino de conceitos onde os estudantes são estimulados a desenvolver seus próprios componentes.

2.2. Sequência de Implementações MIPS

O MIPSFPGA é formado por um conjunto de exemplos de projetos de MIPS seguindo a sequência do livro Patterson e Hennessy [13]. O livro apresenta os projetos de forma incremental. Para auxiliar o ensino, os projetos do MIPSFPGA são numerados seguindo a sequência de figuras do livro. Foram desenvolvidos novos componentes no Hades para que o projeto no MIPSFPGA fosse visualmente o mais próximo da ilustração original do livro. Foram implementados os caminhos de dados do MIPS para 4^a e 5^a edições do livro texto [13].

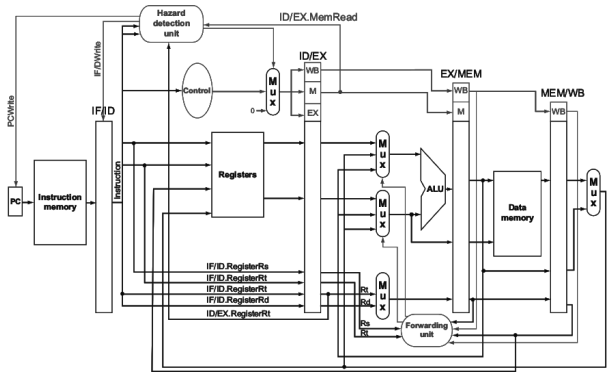


Figura 1. MIPS Pipeline Figura 4.60 Original do Livro [13]

Ao carregar o projeto no MIPSFPGA, o estudante tem acesso a uma figura interativa do livro. A cada passo da simulação (ciclo de clock), os fios e barramentos podem ter seus valores monitorados. A Figura 1 foi extraída do livro e ilustra o processador MIPS *Pipeline* com a *Forward Unit* e a *Hazard Unit*. A Figura 2 apresenta a tela do simulador para a mesma implementação, na qual pode-se observar a forte semelhança. Diferentemente dos outros simuladores ViSiMIPS [10], RAVI [12], WEBMIPS [3], DIMIPSS [5], a figura é dinâmica e pode ser editada. Assim como nos outros simuladores é possível ver o conteúdo dos registradores, memória, avançar ciclo à ciclo, ver detalhes do encaminhamento etc. Mas somente o MIPSFPGA permite ao estudante editar graficamente o projeto e derivar uma nova implementação com mais instruções. Algumas figuras do livro apresentam simplificações, nos projetos do MIPSFPGA pode-se também esconder alguns fios ou barramentos. A metodologia adotada foi colocar explicitamente todos os barramentos. Apenas alguns sinais como *clock*, reset e habilitações não são exibidos para simplificar a visualização.

3. Exercícios

Com o reuso do componente para questões múltipla escolha criado em [8], vários exercícios do livro foram remodelados para adquirirem o formato de múltipla escolha e estão disponibilizados com o MIPSFPGA. Novos exercícios foram propostos também. Este componente permite que os professores e/ou estudantes criem exercícios de múltipla escolha em um arquivo de texto simples e que estes exercícios sejam resolvidos pelo aluno durante a simulação através do componente. As questões são visualizadas dentro do simu-

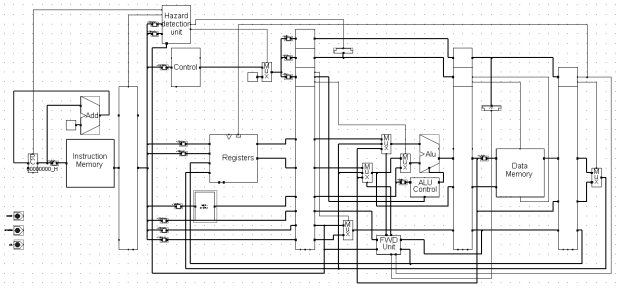


Figura 2. Projeto da Figura 4.60 do MIPS na interface do MIPSFPGA

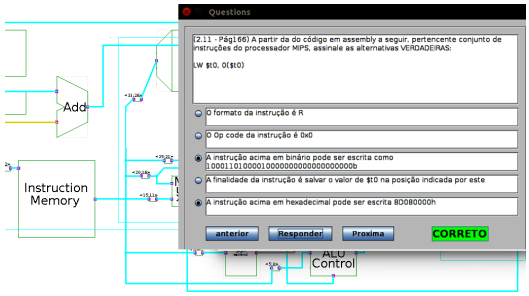


Figura 3. Componente para questões múltipla escolha.

lador na mesma janela. O aluno tem questões para refletir, solucionar e pode, ao mesmo tempo, simular o circuito para conferência das respostas.

O formato para criar questões é bem simples com marcadores. Cada enunciado de questão é incluído entre "<q>" e "</q>". Após estes marcadores, são inseridas alternativas de respostas com os marcadores "<a1>" e "</a1>" para alternativa 1, "<a2>" e "</a2>" para alternativa 2, etc. Para cada alternativa, seguem as respostas. As questões são carregadas uma a uma. O estudante pode navegar para questão anterior ou posterior e verificar se respondeu corretamente. Por se tratar de um componente desenvolvido em linguagem Java, novos componentes com outros recursos para elaboração e visualização de exercícios podem ser futuramente incluídos. As questões seguem um estilo "verdadeiro/falso". Pode haver mais de uma alternativa correta por exercício. Isto permite mostrar que podem existir várias soluções equivalentes para um mesmo exercício.

A Figura 3 mostra o simulador com o componente de questões acionado com uma questão a ser respondida e a Figura 4 exibe um exemplo de arquivo fonte no qual estão descritas as questões.

```
<q>(2.11 - Pág166) A partir da do código em assembly a seguir, pertencente conjunto de
instruções do processador MIPS, assinale as alternativas VERDADEIRAS:

LW $t0, 0($t0)</q>
<a1>O formato da instrução é R f</a1>
<a2>O Op code da instrução é 0x0 f</a2>
<a3>A instrução acima em binário pode ser escrita como
10001101000010000000000000000000b v</a3>
<a4>A finalidade da instrução é salvar o valor de $t0 na posição indicada por este f</a4>
<a5>A instrução acima em hexadecimal pode ser escrita 8D080000h v</a5>

<q>(2.11 - Pág166) Após a execução do algoritmo abaixo, assinale a(s) alternativa(s)
verdadeira(s):
                        ADDI R1,R4,4
                        ADD R2,R4, R0
                        SW R2, 0(R1)
                        LW R1, 0(R1)
                        ADD R3, R2, R1
</q>
<a1>A segunda instrução em hexadecimal é 801020h v</a1>
<a2>O valor final da posição 8 da memória RAM será de 4h f</a2>
<a3>O resgistrador R0 contém 10h no instante inicial f</a3>
<a4>Ao fim da execução, o registrador R3 terá o valor 8h armazenado v</a4>
<a5>A soma de R4 com o valor 4 faz com que a instrução LW leia a posição 8 da memória. f</a5>
```

Figura 4. Exemplo do arquivo de questões múltipla escolha.

4. Desenvolvimento de Novas Implementações

Vários recursos do ambiente utilizado podem ser explorados para a criação de novas implementações do circuito do MIPS. Como já mencionado, o MIPSFPGA oferece uma biblioteca adicional com componentes desenvolvidos para esta função além da ampla biblioteca de componentes do simulador Hades. Com uma rápida curva de aprendizado, é fácil criar novos componentes com reuso dos componentes já desenvolvidos.

4.1. Ampliando o ISA

O livro texto implementa apenas um subconjunto das instruções MIPS para fins didáticos. A ampliação do ISA (*Instruction Set Architecture*) com inclusão de novas instruções é tema de vários exercícios. Ao estender um projeto, uma ou mais partes podem ser alteradas. Como o MIPSFPGA é um editor/simulador, novos fios e componentes podem ser adicionados através da interface gráfica. Suponha que novos sinais são adicionados ao estágio de busca (*Fetch*) em uma implementação com pipeline como a ilustrada na Figura 2. Neste caso, novos sinais devem ser propagados pela barreira. A implementação da barreira é feita como um conjunto de registradores. Muitas vezes, os estudantes não sabem o que tem na caixa preta da barreira. Ao editar a barreira e acrescentar uma nova funcionalidade, o estudante passa a compreender melhor a implementação com estágios pipeline e como estes sinais são propagados e armazenados.

Nos projetos do MIPSFPGA, as barreiras foram implementadas como sub-projetos hierárquicos e podem ser facilmente editadas e estendidas. Ao editar a barreira, a parte interna é visualizada como ilustrado na Figura 5 para a barreira IF_ID (entre os estágios *Fetch* e *Decode*).

A primeira implementação do livro é o *Single* MIPS (sem *pipeline*). Neste projeto, a unidade de controle é

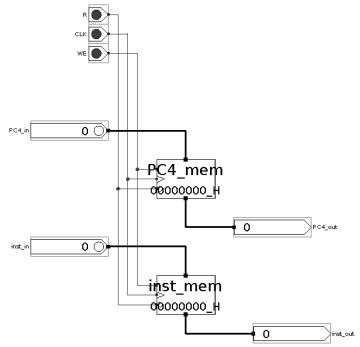


Figura 5. Detalhamento da barreira pipeline IF_ID da Figura 2 com a propagação dos registros IR e PC.

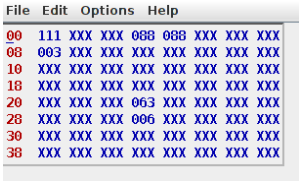


Figura 6. Edição da unidade de controle para o processador Single Mips.

```
...
//opcode 35 = LW
  case LWopcode:
     vector_Signals = new StdLogicVector(8,23); // Hex 0x23 = 0 00 1 0111
 // Regdst RT=0, OP=00 +, ALUsrc = IM = 1, MW=0, MR=1, RegW=1, MemtoReg=1
     value_Branch = new StdLogic1164(False); // bit signal
     value_IFflush = new StdLogic1164(False);  // bit signal
      break;
 //opcode 8 = ADDI
  case opcodeADDI:
    vector_Signals = new StdLogicVector(8,18); // Hex 0x18 = 0 00 1 0001
  // Regdst RT=0, OP=00 +, ALUsrc = IM = 1, MW=0, MR=0, RegW=1, MemtoReg=0
    value_Branch = new StdLogic1164(False);
    value_IFflush = new StdLogic1164(False);
    break;
....
```

Figura 7. Código parcial da Unidade de Controle em Java para as versões *Pipeline*

uma tabela com o mapeamento do código da instrução (*opcode*) para os sinais internos de controle do MIPS. A implementação do MIPSFPGA é baseada na tabela como um módulo de memória. Os módulos de memória do Hades já possuem interface gráfica com edição, leitura e escrita em arquivo,. Portanto, o projeto e suas alterações são simplificados. Suponha que deseja-se incluir uma nova instrução que não necessita modificar o caminho de dados com sinais extras. Neste caso, basta editar o componente memória ROM e programar os sinais de controle para o *opcode* desejado. A Figura 6 mostra a janela de edição da Unidade de Controle do processador *Single* Mips. Observe que apenas 6 *opcodes* estão programados para as instruções descritas no livro texto. Por exemplo, o *opcode* da instrução *load* é 35 (decimal) ou 23 (hexadecimal) e foi programando para gerar os sinais 063 (hexadecimal) ou 00 0110 0011 em binário para controle dos recursos internos (Alu, multiplexadores, banco de registros, memória). Na Figura 6, XX significa indefinido. É importante destacar que o MIPSFPGA não foi proposto para programar em *assembly* como os simuladores MARS [15] e SPIM [11].

Os componentes unidade de controle e memória de instruções permitem a edição em binário e hexadecimal. Portanto os estudantes são estimulados ao exercício de codificar que é um passo fundamental para se projetar processadores. Para verificação do funcionamento, os valores dos sinais internos, registradores e memória de dados são facilmente visualizados com interface gráfica.

Se não existir o componente pronto na biblioteca, pode-se derivar um novo componente de algum outro já existente. Os componentes são implementados em Java. Por exemplo, a unidade de controle das implementações pipeline tem seu comportamento descrito em Java. A Figura 7 ilustra uma parte do código da unidade de controle com destaque para a decodificação dos *opcodes* das instruções de *Load* e Addi.

Pode-se observar que a programação está em alto nível com o mapeamento do *opcode* nos sinais de controle. Mesmo para estudantes que não estão habituados a Java, pequenas alterações são possíveis de ser realizadas para inclusão ou exclusão de instruções.

O componente pode ser recompilado e/ou um novo componente pode ser criado. O componente tem associado um arquivo "símbolo" com sua representação gráfica que será visualizada no editor. Pode ser uma simples caixa com os nomes dos sinais, ou então algo mais elaborado. O arquivo que contém o símbolo também é descrito com componentes básicos (linhas, formas, I/O) e sua descrição é simples e direta. A representação pode também ser gráfica com animações, como por exemplo, a máquina de estados que mostra uma animação durante a simulação.

4.2. Novas Funcionalidades

Similar aos demais simuladores de caminho de dados que mostram o código com mnemônicos em *assembly* (*load*, *store*,etc.). O MIPSFPGA inclui componentes de visualização que servem para facilitar a compreensão. No caso específico do *pipeline*, a visualização passo a passo da propagação das instruções através dos estágios está presente em muitos simuladores MIPS com interface gráfica. Através do tipo *String*, foram criados alguns componentes que decodificam os sinais binários do caminho de dados e exibem de forma legível com mnemônicos, por exemplo

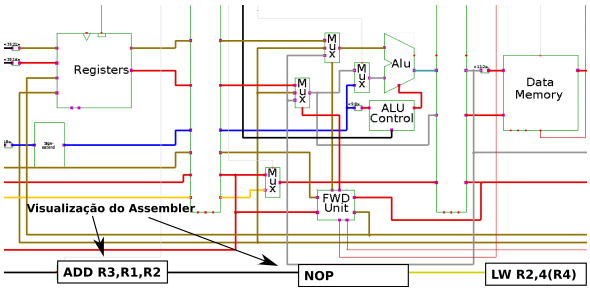


Figura 8. Exibição do deslocamento das instruções no MIPS *Pipeline*.

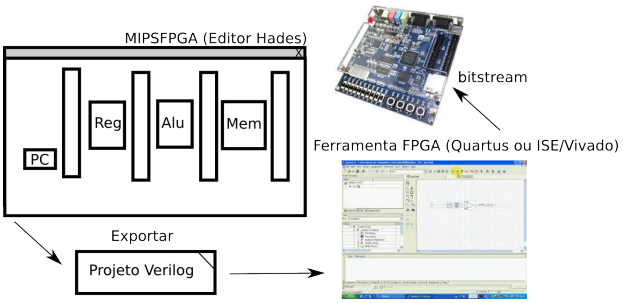


Figura 9. Fluxo para exportação da implementação para prototipação em FPGA

"add r1,r2,r3"ao invés de $0000\ldots10$. A Figura 8 mostra os componentes que exibem a instrução executada em cada *pipeline* no processador Mips *Pipeline* com *Forward Unit* e *Hazard Detection Unit*.

5. Síntese em Kits FPGA

Os programas de simulação são ferramentas importantes para auxílio no aprendizado, porém a implementação física do projeto com a prototipação em FPGA, permite ao estudante ter uma comprovação do funcionamento. Existem diversos Kits educacionais para ensino de circuitos digitais com FPGA e acessórios (chaves, leds, *displays*, etc). Entretanto, o ensino e uso de linguagem de descrição de *hardware*, como VHDL e Verilog, é mais complexo para a maior parte da comunidade de programadores que estão habituados aos paradigmas de linguagem de programação imperativas como C, C++, Java. Principalmente estudantes de Ciência da Computação ou Sistemas de Informação.

Além disso, os ambientes de projeto com FPGA são ferramentas complexas com muitas janelas e menus, como o Quartus da Altera ou o ISE/Vivado da Xilinx. O nosso objetivo é simplificar o projeto para ampliar a divulgação inicial. A proposta do MIPSFPGA é exportar de forma transparente a implementação do projeto com a geração automática do código em Verilog. A princípio, o estudante pode sintetizar um MIPS sem ter ainda conhecimento de Verilog ou VHDL. O projeto é então carregado na ferramenta de síntese FPGA que será usada apenas para carregar o *bitstream* no FPGA como ilustrado na Figura 9.

Como o MIPSFPGA segue uma metodologia incremental, permite aos estudantes e professores acrescentar novas extensões. Para desenvolver um novo projeto que necessite de novos componentes que não estejam disponíveis nas bibliotecas do Hades e MIPSFPGA, primeiro é necessário criar o componente. O mais simples é fazer o reuso ou seguir o padrão de desenvolvimento descrito no manual do Hades [9]. O segundo passo é testar o projeto no ambiente de simulação. Como já mencionado, o componente é descrito em Java, com a qual implementa-se a descrição comportamental deste através da orientação a eventos. Ou seja, toda vez que os sinais de entrada são alterados, o componente é chamado para tratar a entrada e escalonar as saídas correspondentes de forma reativa. O terceiro passo é a criação de um arquivo em Verilog que implemente a funcionalidade e interface do componente. Finalmente, o gerador do MIPSFPGA pode ser chamado para gerar os arquivos verilog do novo projeto. O projeto é então carregado em uma ferramenta de FPGA que fará a síntese e a geração do *bitstream* para prototipação em FPGA. Trabalhos futuros irão incluir a possibilidade de desenvolver o componente em Verilog e importar na simulação do Hades.

5.1. Entrada e Saída para Depuração

Para simplificar a verificação da implementação no FPGA, já que o subconjunto de instruções apresentado no livro é bem reduzido, o MIPSFPGA propõe o uso de componentes de visualização durante a simulação e também na prototipação que podem ser encontrados na maioria dos kits.

A Figura 10 mostra o diagrama de uma implementação na qual pode-se ver o processo de simulação e prototipação. Pode-se observar três tipos de componentes. Primeiro, o caminho de dados e controle que é o projeto do processador. No simulador é descrito com os componentes da biblioteca, os fios e barramentos podem ser visualizados passo a passo. Porém, quando é prototipado no FPGA, como ilustra a Figura 10, o estudante passa a não ter acesso as partes internas para verificar o funcionamento, pois tudo está programado dentro do FPGA.

O segundo tipo de componentes são os recursos de entrada e saída presentes na placa. Como a maioria das placas disponibiliza chaves e *displays*, o MIPSFPGA propõe uma abordagem simples para permitir a depuração. A ideia é priorizar a exibição do banco de registradores, do ciclo

atual do *clock*, do contador de programa PC e do conteúdo da memória. Basicamente, usa-se de forma compartilhada os *displays* numéricos para exibir os valores. Como muitas placas tem pelo menos quatro botões, a proposta é cada um selecionar uma opção: registradores, pc, memória ou *clock*. Se a memória ou registradores forem selecionados com o botão, precisamos de um paramêtro adicional para saber qual posição será visualizada. A solução é usar as chaves que irão selecionar qual o endereço de memória ou registro. Por exemplo, como ilustra na Figura 10, se as chaves tem o valor 2 e o botão de registradores estiver pressionado, o *display* irá exibir o conteúdo do registrador 2. Assim, o estudante pode ter acesso a visualização do conteúdo de qualquer registrador ou posição de memória de uma forma simples e direta.

Como a simulação e depuração são realizadas passo a passo, o *clock* também é controlado pela interface proposta. O estudante pode observar os valores dos registradores e da memória, a cada ciclo de *clock*. Para a escolha de quantos pulsos de *clock* serão executados, o aluno seleciona um valor que serão injetados através das chaves. O quarto botão, ao ser pressionado indica, através do *display*, o valor do contador de programa (PC).

Para que o ambiente seja amigável, antes de prototipar, o estudante tem no simulador o mesmo ambiente de verificação da placa de FPGA. O terceiro tipo de componentes ilustrado na Figura 10 são os componentes do kit FPGA executando no simulador MIPSFPGA. Estes componentes não são sintetizados, apenas parte deles, pois são componentes que já existem na placa e são externos ao FPGA. Pode-se visualizar os quatro seletores (botões) para selecionar: registradores, *clock*, PC e memória. As chaves possuem uma interface mais simples, basta digitar o valor, o *display* e o contador de ciclos de *clock* já executados.

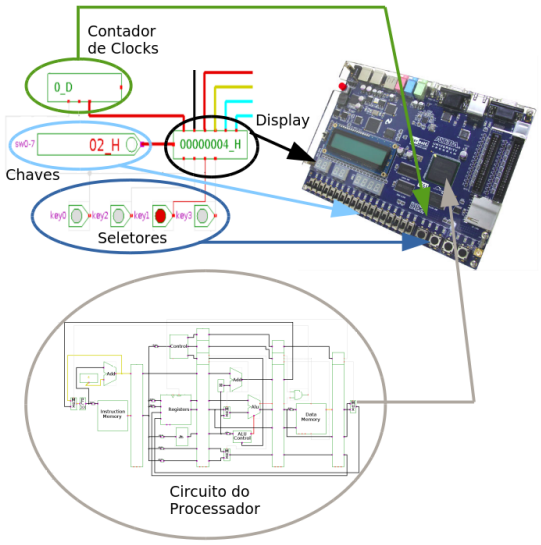


Figura 10. Componentes de depuração no simulador e no kit de FPGA DE-2 Altera

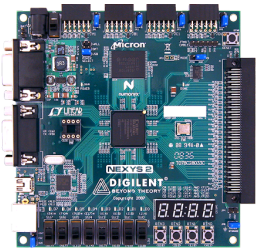


Figura 11. Kit Digilent Nexys 2. [4]

5.2. Kits para Validação

Como existem dois grandes fabricantes de FPGA, para a validação da proposta foram selecionados dois Kits, um da Altera e outro da Xilinx. Os kits são Altera DE2 *Development and Education Board*; e Digilent Nexys 2 com FPGA Xilinx. São kits populares disponíveis em várias universidades. Colaboradores podem fazer o reuso dos componentes da biblioteca do MIPSFGPA para incluir outros kits de prototipação de FPGA.

O Kit Nexys 2, que é ilustrado na Figura 11, é um kit desenvolvido pela Digilent com um FPGA Spartan 3E-500 FG320 da Xilinx. O kit possui quatro *displays* de sete segmentos, quatro botões e sete chaves, além de conectores, memória RAM dentre outros. Como o MIPS é de 32 bits, os valores a serem exibidos precisariam de oito *displays*. Como a placa posui apenas quatro *displays*, a cada segundo metade dos bits são exibidos, ora os dezesseis menos significativos e ora os mais significativos), convertidos em hexadecimal. Este recurso é implementado no processo de geração do Verilog para o kit Nexsys2 de forma transparente.

O Kit Altera DE2, que pode ser visto na Figura 10, é um kit desenvolvido pela Altera que traz o FPGA Ciclone II com 35K LEs. Este kit possui mais recursos que o primeiro kit com oito *displays* de sete segmentos (o que permite exibir números de 32 bits diretamente), quatro botões e dezoito chaves, além de *display* LCD, Memória RAM, leitor de cartões SD dentre outros. Extensões podem ser propostas para visualização mais amigável com o uso da saída VGA e/ou *display* LCD.

Como já mencionado, os quatro botões foram utilizados para selecionar a opção do depurador a ser executada: o botão 3 é responsável por disparar os pulsos de *clock* em uma quantidade escolhida através das chaves. O botão 2 exibe nos *displays* o valor atual contido na saída de PC. O botão 1 exibe nos *displays* o conteúdo do registrador

endereçado pelo valor das chaves e, por fim, o botão 0 exibe nos *displays* o conteúdo da posição de memória endereçada pelo valor das chaves. Quando nenhum botão for pressionado, os *displays* exibirão o valor especificado pelas posições das chaves.

5.3. Gerador de Código em Verilog

O Hades foi concebido com compatibilidade de simulação com VHDL. Um gerador simples de VHDL é incluso no Hades, porém o VHDL gerado não é compatível com as ferramentas da Altera e Xilinx. O MIPSFPGA adotou a linguagem Verilog para exportar o projeto para a prototipação. Contudo, é possível usar a mesma metodologia para exportar em VHDL. Cada componente da biblioteca possui dois arquivos. Um arquivo em Java que descreve o comportamento na simulação e um arquivo em Verilog, que é usado na síntese do projeto.

O projeto no Hades é descrito por um arquivo texto estrutural composto por duas partes: componentes e sinais. A parte de componentes tem a lista de componentes do projeto. Cada componente tem seu tipo, nome da instância e paramêtros para simulação (ex: número de bits, valor inicial, etc). A parte de sinais descreve a conexão entre os componentes. Um *parser* foi implementado para converter o formato do Hades para um arquivo estrutural Verilog. Como já mencionado, o *parser* realiza algumas funcionalidades específica para tornar transparente a geração de componentes que são externos ao FPGA (chaves, *display*, etc). O *parser* foi implementado em Java e tem código aberto para extensões, como por exemplo, inclusão de novos kits.

5.4. Conclusão

Este artigo apresenta uma nova abordagem baseada em simulação para auxiliar no ensino do processador do MIPS. Diferente dos outros simuladores, o MIPSFPGA inova ao usar uma metodologia incremental, ser simples e genérico para ser facilmente estendido. Ademais, a ferramenta inclui síntese e validação em FPGA, apoio para ensino com exercícios integrado no ambiente, exemplos na sequência do livro texto [13] como correspondência um para um nas ilustrações e depuração com kits FPGAs. A ferramenta tem sido utilizado em sala de aula em nível de simulação. A síntese e validação em FPGA está em fase de teste e será aplicada em sala de aula. Para trabalhos futuros, pretende-se acrescentar novas formas para a visualização dos dados, modelagem de novos componentes em Verilog tanto para prototipação como para simulação no Hades com acesso a um FPGA remoto.

6. Agradecimentos

Este projeto foi financiado pelas agências FAPEMIG (Universal 2014), CNPq e CAPES e a empresas Intel/Brazil e Gapso.

Referências

[1] F. A. Alves, D. Almeida, L. Bragança, A. B. Gomes, R. S. Ferreira, and J. Nacif. Ensinando arquiteturas vetoriais utilizando um simulador de instruçoes MIPS. *International Journal of Computer Architecture Education*, 2016.

[2] E. Z. Bem and L. Petelczyc. Minimips: a simulation project for the computer architecture laboratory. *ACM SIGCSE Bulletin*, 35(1):64–68, 2003.

[3] I. Branovic, R. Giorgi, and E. Martinelli. Webmips: a new web-based mips simulation environment for computer architecture education. In *Workshop on Computer architecture education*. ACM, 2004.

[4] DIGILENT. Nexys 2 spartan-3e fpga trainer board.

[5] A. F. Felix, C. V. Pousa, and M. Carvalho. Dimipss: Um simulador didático e interativo do MIPS. In *Workshop sobre Educação em Arquitetura de Computadores*, pages 49–52, 2006.

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

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

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

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

[10] M. Kabir, M. Bari, and A. Haque. Visimips: Visual simulator of MIPS32 pipelined processor. In *Computer Science & Education (ICCSE)*. IEEE, 2011.

[11] J. R. Larus. *Spim s20: A mips r2000 simulator*. Center for Parallel Optimization, Computer Sciences Department, University of Wisconsin, 1990.

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

[13] D. A. Patterson and J. L. Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2013.

[14] A. Torres and A. Brito. Ferramenta de auxílio no ensino de organização e arquitetura de computadores: extensão ptolemy para fins educacionais. *Int. Journal of Computer Architecture Education*, 2012.

[15] K. Vollmar and P. Sanderson. Mars: an education-oriented MIPS assembly language simulator. In *ACM SIGCSE Bulletin*, volume 38, pages 239–243. ACM, 2006.