Plain: Ferramenta para Desenvolvimento de Aceleradores para Overlays em FPGA na Nuvem em Tempo de Execução*

Fernando Passe[1], Lucas Bragança[1], Michael Canesche[1], Felippe Cathoud[1], José A. Nacif[1], Ricardo Ferreira[1]

[1]Universidade Federal de Viçosa, Brasil

Abstract. *FPGAs provide an energy-efficient solution to design data-flow cloud accelerators. Nevertheless, there are some challenges to widespread usage as the compiling time (minutes to hours) and low-level hardware knowledge. The READY tool recently provides a compiling time reduction to the range of microseconds, generating code for the Intel/Altera HARP 2 platform. Although READY generates code that can easily integrate with C++ code, the accelerator specification input is a text format graph. We propose here an extension named PLAIN, which includes a browser-based graphical interface. Also, we improve the design flow by adding two simulation levels and high-level feedback profiling. Furthermore, the PLAIN tool facilitates the design and evaluation of new operators.*

Resumo. *Os FPGAs oferecem eficiência energética para o desenvolvimento de aceleradores para fluxo de dados na Nuvem. Porém, existem vários desafios para popularizar seu uso. Dentre eles, podemos citar o tempo de compilação (que pode demorar horas) e conhecimento de hardware para uso adequado de linguagens de síntese de alto nível. Recentemente, a ferramenta READY possibilitou a redução do tempo de compilação e configuração para microsegundos. O ambiente foi validado na plataforma em nuvem HARP 2 da Intel/Altera. Apesar da integração com a Linguagem C++ para o desenvolvimento das aplicações, o acelerador é descrito de forma textual como um grafo. Neste trabalho é apresentado a extensão PLAIN, que inclui uma interface online gráfica para descrição dos aceleradores, a automatização do fluxo de projeto, dois níveis de simulação e um nível de execução. A ferramenta também mostra estatísticas de desempenho e permite criação de novos operadores para exploração do espaço de projeto.*

1. Introdução

A modelagem de um algoritmo com um grafo de fluxo de dados torna explícita a descrição do paralelismo. Além disso, o mapeamento do grafo diretamente no *hardware* permite eliminar que a busca e decodificação das instruções sejam realizadas a cada ciclo, economizando energia. Os dados irão fluir pela estrutura, gerando reuso e reduzindo as operações com estruturas temporárias de memória. Portanto, os grafos de fluxo oferecem eficiência energética e desempenho.

*Financiamento: FAPEMIG, CNPq, Nvidia, Funarbe. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. Apoio dos laboratórios: Intel Academic Compute Environment e Departamento de Informática da Universidade Federal de Viçosa (UFV).

O custo de projetos em *ASIC* (circuitos dedicados) é elevado para mapeamento dos grafos de fluxo. Uma alternativa é o uso de hardware reconfigurável como os FPGAs [Penha et al. 2019]. Porém, o desenvolvimento com FPGA ainda exige um conhecimento detalhado do hardware e de uma linguagem de descrição de *hardware*, normalmente Verilog ou VHDL. Além disso, o processo de síntese (ou compilação) pode levar várias horas. Duas alternativas podem ser usadas: (a) linguagens de alto nível; (b) camadas virtuais ou *Overlays*.

Como linguagens de alto nível podemos destacar HLS baseadas em C/C++ (*High Level Synthesis*) [Nane et al. 2015], OpenCL [Krommydas et al. 2016] e Maxeler [Stanojević et al. 2019]. HLS ou OpenCL tem a vantagem da programação em alto nível em C/C++. Porém, o grafo de fluxo de dados que irá implementar o hardware é gerado de forma implícita, sem muito controle do programador e pode não ser eficiente. Portanto, o programador precisa ter conhecimentos sobre o estilo de código que deve usar, diretivas e o processo de compilação HLS. A abordagem da Maxeler [Stanojević et al. 2019] utiliza a linguagem Java como uma extensão onde o programador descreve explicitamente o grafo de fluxo de operações. O código é compilado e sintetizado para FPGA. Entretanto, qualquer modificação requer que o projeto seja re-sintetizado e todo o FPGA deve ser reprogramado.

A segunda abordagem evita a re-síntese e a reprogramação do FPGA com *Overlays* [Nane et al. 2015], que em geral implementam um CGRA (*Coarse-Grained Reconfigurable Arrays*) como uma camada virtual sobre o FPGA. Os CGRAs são reconfigurados a nível de instrução, simplificando o processo de mapeamento. O CGRA é configurado apenas uma única vez sobre o FPGA. Posteriormente, o compilador precisa apenas gerar código para CGRA. Entretanto faltam ferramentas para compilação e execução de aplicações CGRA mapeados em FPGAs.

Desse modo, este artigo propõe o uso combinado das duas abordagens. Primeiro, a aplicação é integrada em um código alto nível escrito em C/C++, semelhante a abordagem CUDA de GPUs. Diferente de OpenCL/HLS, o grafo de fluxo de dados é explicitamente descrito para especificação do algoritmo (ou *kernel*) que o acelerador irá executar, semelhante a abordagem da Maxeler [Stanojević et al. 2019]. Entretanto, não é necessário recompilar para FPGA a cada modificação, pois é feito o uso de um *Overlay*. Finalmente, todo o fluxo de projeto foi automatizado, sendo transparente para o programador. O grafo especificado irá executar em um FPGA na nuvem. A implementação foi realizada fazendo uma extensão da ferramenta READY [Silva et al. 2019], denominada PLAIN, que encapsula e automatiza todo fluxo, inclui uma interface gráfica para reduzir a curva de aprendizagem, dois modos de simulação que permite a exploração de novos operadores de maneiras distintas, apresenta relatório da execução de forma simplificada e ainda permite a criação de novos operadores para futuras expansões.

A estrutura deste artigo é descrita a seguir. A Seção 2 descreve o grafo de fluxo de dados. A Seção 3 apresenta a ferramenta READY com suas especificações, detalhes e funcionamento. A Seção 4 apresenta a ferramenta PLAIN e as Seções 5 e 6 realizam uma comparação com outras ferramentas. Por fim, a Seção 7 discute as principais conclusões e trabalhos futuros.

2. Grafos de Fluxo de Dados

Um grafo de fluxo de dados apresenta o paralelismo de uma forma explícita no nível de instruções e pode conter operadores com fluxo de controle [Ferreira et al. 2004, Nowatzki et al. 2017]. Um conceito importante é a temporização para saber qual elemento da sequência está sendo trabalhado em cada nível. A vazão sempre será máxima e o número de operações executadas pelo grafo por ciclo é exatamente o número de operadores do grafo, ou seja, a cada ciclo todas as operações são executadas em paralelo.

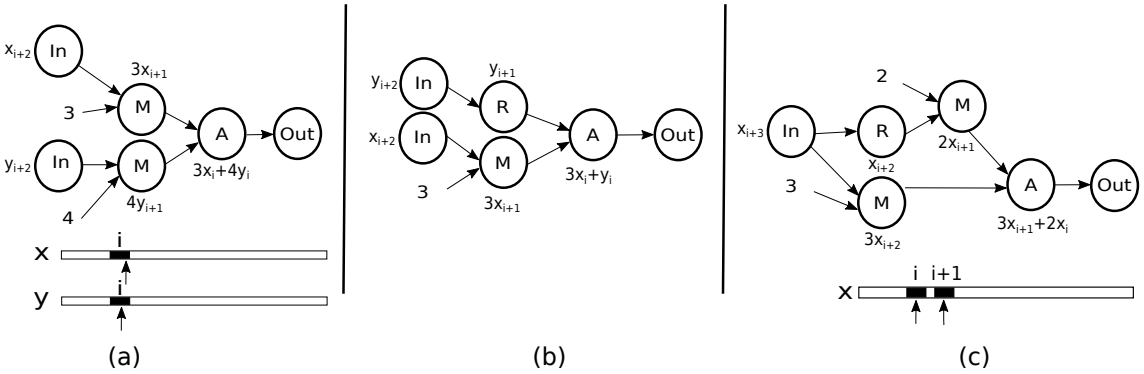


Figura 1. (a) Grafo com uma soma de vetores; (b) Soma de vetores que requer balanceamento; (c) Convolução com dois elementos consecutivos de um vetor.

A Figura 1 apresenta três exemplos. Primeiro, apresenta uma computação simples $out = 3 \cdot x_i + 4 \cdot y_i$ na Figura 1(a) onde os operadores m fazem multiplicações e o operador a faz uma adição. A cada ciclo, novos elementos de x e de y são inseridos no grafo mantendo o pipeline sempre cheio. O segundo exemplo faz a computação $out = 3 \cdot x_i + y_i$. Neste caso é necessário inserir um registrador (operador R) para equalizar os caminhos, garantindo que os elementos do vetor chegarão de forma sincronizada em a. Supondo que cada operação tem a latência de 1 ciclo, o balanceamento poderia ser automático, porém nas ferramentas apresentadas neste artigo (READY e PLAIN), o balanceamento deve ser explícito para permitir operações do tipo convolução com várias aplicações em aprendizado de máquina. O último exemplo calcula $out = 3 \cdot x_{i+1} + 2 \cdot x_i$. Como os caminhos estão desequilibrados, gerou-se computações com elementos distintos do vetor. Neste exemplo, há dois elementos consecutivos. A linguagem MAXELER [Stanojević et al. 2019] oferece um operador para implementar o elemento posterior e anterior em uma sequência.

A ferramenta PLAIN, proposta neste trabalho, permite a visualização da sequência de dados fluindo pelo grafo na fase de desenvolvimento do algoritmo. O programador pode implementar um trecho do grafo, injetar valores e verificar se o fluxo e o balanceamento estão corretos. A Seção 4.3 apresenta este modo de simulação que é útil na fase inicial do projeto.

3. Ferramenta READY

A ferramenta READY [Silva et al. 2019] permite a descrição de um ou mais grafos de fluxo e seu acoplamento com um código C++. O grafo é mapeado em tempo de execução em um *overlay* CGRA que executa como um acelerador no FPGA. O CGRA utiliza uma rede multiestágio que permite uma implementação eficiente de posicionamento e roteamento do grafo de fluxo de dados [Ferreira et al. 2011]. A ferramenta possui

também uma API para execução de múltiplos grafos [Silva et al. 2019]. Os grafos podem ser réplicas do mesmo grafo ou grafos diferentes. Semelhante a uma GPU, múltiplos grafos (ou threads) são usados para esconder a latência do CGRA e garantir a vazão máxima no mapeamento dos grafos na arquitetura física.

3.1. Especificação

A Figura 2(a) representa a fase de compilação e a Figura 2(b) a fase da execução da ferramenta READY. A nova extensão proposta neste trabalho inclui a etapa (1), onde o grafo de fluxo é descrito de forma gráfica ou textual (ver Seção 4). Na ferramenta original o grafo deve ser incluído dentro do código C++ com uma descrição textual.

O grafo descrito na ferramenta PLAIN é transformado em um formato intermediário JSON. Trabalhos futuros poderão incluir a geração do grafo por compiladores e uso de DSL (linguagens de domínio específico). Na etapa (2), o JSON é carregado como um grafo de fluxo de dados pela biblioteca construída do READY. A ferramenta PLAIN agregou mais portabilidade onde a descrição JSON também pode ser carregada em tempo de execução. Posteriormente, o mapeamento é executado com uma implementação com complexidade polinomial em tempo de execução [Silva et al. 2019], gerando a configuração para o CGRA.

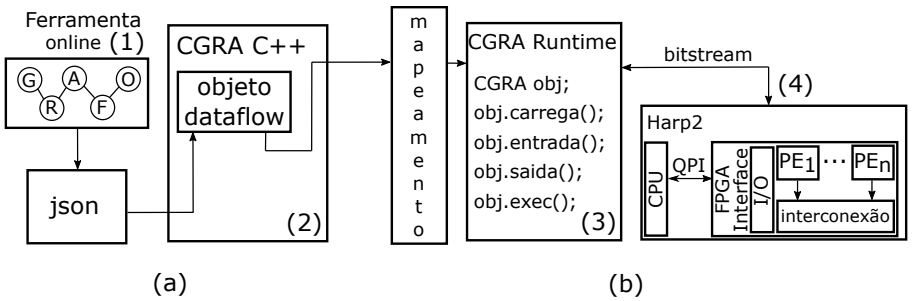


Figura 2. Etapas de (a) compilação (b) execução da ferramenta Ready

A configuração é transferida para o CGRA na etapa (3). As variáveis das entradas e saídas fazem o acoplamento entre o código C++ e o grafo. Na etapa (4) começa a execução com sobreposição do envio/recebimento dos dados e da computação no acelerador CGRA.

A ferramenta READY possibilita a execução síncrona ou assíncrona. Similar ao SDK Cuda da Nvidia [Nickolls et al. 2008], o grafo pode ser executado de maneira síncrona, bloqueando o processador até que a tarefa da aceleração esteja concluída ou executando de forma assíncrona, sobrepondo as tarefas do processador e do acelerador. Vale ressaltar que o sistema inicia enviando as palavras de configuração pelo canal de dados para configurar o acelerador. Uma vez configurado o CGRA, a execução começa assim que os primeiros dados são transmitidos da CPU para o acelerador. No caso de falha na cache, a entrada do grafo fica bloqueada aguardando os dados solicitados.

Outra contribuição da ferramenta READY é o encapsulamento da comunicação com FPGA da Intel/Altera. Primeiro, a API OPAE [Luebbers et al. 2020] da Intel/Altera foi encapsulada na camada superior do programa. OPAE é uma API desenvolvida inicialmente pela Intel/Altera e aberta para a comunidade com o objetivo de simplificar a

integração de vários tipos diferentes de dispositivos aceleradores de FPGAs. Uma nova API SW/HW foi desenvolvida sobre a plataforma de pesquisa em arquitetura heterogênea Intel/Altera (HARP 2). Esta API pode ser ampliada para outros ambientes de FPGA na nuvem.

3.2. Execução

A ferramenta READY possui suporte para o HARP 2, disponível na Nuvem da web pela Intel Research [Intel 2020], permitindo que projetos para FPGA possam ser executados *online*. Diferentemente das instâncias EC2 F1 da Amazon que são comerciais, a plataforma web do HARP 2 é voltada para centros de pesquisas. Outro diferencial do HARP é o acoplamento entre o FPGA Arria 10 de modelo 10AX115U3F45E2SGE3 com o processador Xeon E52680 2.4 GHz com 14 núcleos e 24 MBytes de cache L3 acoplado na FPGA via um barramento 2-PCI e um Intel QuickPath Interconnect (QPI) com a abordagem de memória compartilhada com cache transparente.

4. Ferramenta PLAIN

A ferramenta PLAIN foi construída para simplificar o desenvolvimento de aceleradores com grafos de fluxo de dados. Como a ferramenta executa na Web, permite ao usuário gerar toda a estrutura sem a necessidade de conhecimento de FPGA/CGRA. Portanto, reduz a curva de aprendizagem, cria um ambiente no navegador para desenvolvimento de algoritmos com execução em FPGAs na nuvem e ainda permite extensões que serão apresentadas a seguir.

A Figura 3 mostra a implementação de filtro de impulso finitos (FIR4) onde seu paralelismo é explicitado com o uso da ferramenta. Este grafo calcula uma convolução $out = x_i + 2 * x_{i+1} + 3 * x_{i+2} + 4 * x_{i+3}$, onde o índice i representa o i-ésimo elemento da sequência.

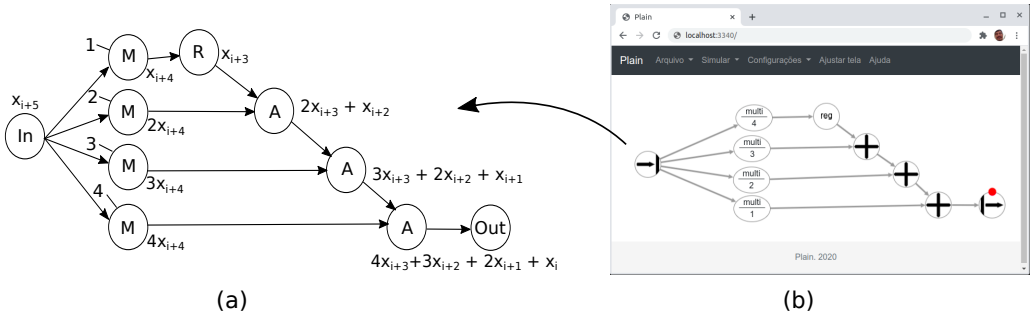


Figura 3. (a) Grafo de fluxo FIR4; (b) Representação do FIR4 na ferramenta PLAIN.

Além das facilidades de edição gráfica dentro de um navegador, sem a necessidade de instalação de ferramentas para FPGA, a ferramenta também permite ao usuário a criação dos seus próprios operadores. Este recurso é importante para definição de quais os operadores o CGRA deve implementar. Apesar de os primeiros CGRA terem sidos definidos há mais de duas décadas [Wijtvliet et al. 2016], ainda não existe um consenso sobre o conjunto de operadores. Por exemplo, para códigos x86 é conhecido que 90% do código é mapeado em apenas 25 *opcode* ou operações básicas [Mutigwe and Aghdasi 2013]. No modelo de computação espacial e com as novas cargas de trabalho de aprendizado de

máquina, quais são os operadores que um CGRA deve conter? Pensando neste aspecto, a ferramenta PLAIN possibilita a criação e validação de novos operadores que será detalhada na Seção 4.2. O usuário pode fornecer uma descrição comportamental do novo operador em C++ para validação da execução e/ou descrição comportamental/estrutural em Verilog para acoplamento em um novo CGRA. Esta modificação é implementada com inclusão de metadados na descrição JSON com os trechos de códigos.

4.1. Descrição do Grafo

A estrutura utilizada para criação do modelo do grafo foi o tipo JSON (JavaScript Object Notation) [JSON 2020], sendo escolhida por ser uma estrutura aberta permitindo que sejam adaptado para o funcionamento em diversos sistemas. A partir desta estrutura foram definidos um conjunto de nodos, divididos em tipos lógicos (*and*, *or*, *mux* e *not*) e tipos aritméticos (*sub*, *subi*, *add*, *addi*, *mult* e *multi*). Além disso, possui nodos de entrada (*in*) e saída (*out*). Esse subconjunto de operadores predefinidos possibilita ao usuário realizar uma superposição dos mesmos para construção de um operador mais complexo.

Para o desenvolvimento da interface foi utilizado o Cytoscape JS [Franz et al. 2016] que é uma biblioteca JavaScript que permite a visualização/manipulação de grafos. Ela funciona no navegador e pode ser associado com ferramentas como o Node JS para permitir o processamento dos dados em um servidor. Além das funcionalidades de edição (adição e remoção de arestas), o Cytoscape JS oferece vários algoritmos de teoria de grafos que podem ser utilizados para análise de métricas como histograma de grau de entrada e saídas, algoritmos de centralidade, *betweeness, pagerank*, dentre outros. Além disso, a ferramenta PLAIN utiliza o *Webpack* para que tudo funcione só no navegador sem que seja necessário o uso de servidores ou algum *plugin* extra, independente do sistema operacional.

A tela principal da ferramenta dispõe de um espaço de desenho (área de trabalho) onde o usuário faz a edição do grafo. Além disso é possível realizar a simulação, criação de operadores, execução no FPGA e visualização de métricas após a execução.

4.2. Novos Operadores

É possível estender a ferramenta com a inclusão de novos operadores ou a importação de módulos que podem ser utilizados para criar grafos maiores. A Figura 4 mostra um exemplo de desenvolvimento de um novo operador. Primeiro, o novo operador é criado definindo um símbolo e suas funcionalidades (entradas, saídas, operação ou metadado em C++ com a descrição comportamental). As operações simples como soma, multiplicação ou pequenas combinações pode ser diretamente inseridas na descrição. Uma operação mais complexa pode ser descrita em C++ ou Verilog. Segundo, se o operador foi inicialmente definido em C++/Verilog pode ser acoplado ao código C++ da aplicação e simulado.

Como já foi demonstrado na ferramenta READY para aplicação K-Means [Silva et al. 2019], ao criar um operador complexo, o desempenho do CGRA passa de 25,6 Gops/s para 100 Gops/s. Pois cada operador complexo pode implementar cerca de 20 operações básicas. A Figura 4(a) ilustra um exemplo de um novo operador FIR4 que foi usado em cascata para aplicar três convoluções em sequência seguido de dois operadores de uso geral. A Figura 4(b) mostra a implementação do operador na arquitetura *overlay*. Além do desempenho, operadores com maior granularidade economizam

recursos de interconexão. A implementação atual do *overlay* possibilita no máximo 256 conexões entre os operadores. Ao usar um operador de granularidade maior, as conexões internas não fazem uso da rede, sobram mais recursos para a interconexões dos outros operadores. Ao mesmo tempo, os operadores simples e universais deixam a ferramenta flexível para descrição de um domínio maior de aplicações.

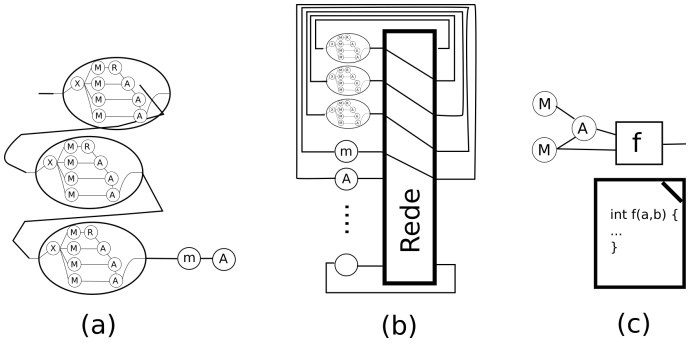


Figura 4. (a) Exemplo de um novo operador; (b) Uso do operador no *Overlay*; (c) Exemplo de um novo operador f com metadado para sua funcionalidade.

O mesmo princípio foi utilizado pela Nvidia com os operadores *Tensor cores*. Cada operador tensor encapsula mais de 100 multiplicações e somas em pipeline. Por isto, uma GPU V100 com 640 Tensores (1,5 Ghz) faz $640 \times 1,5G * 100 = 96$ Tera ops/s.

Como novos operadores é uma área em estudo, a ferramenta PLAIN permite descrever o comportamento do operador com um código C++ através da interface do navegador. Figura 4(c) mostra um exemplo de novo operador com descrição por código.

4.3. Simulação

A ferramenta PLAIN possibilita dois níveis de simulação. Primeiro, o grafo pode ser simulado com a injeção de pequenas sequências de valores pela interface gráfica. Se um novo operador for criado, este recurso também pode ser usado pois o comportamento do operador estará descrito com operadores já existentes (subgrafo) ou através de um código C++.

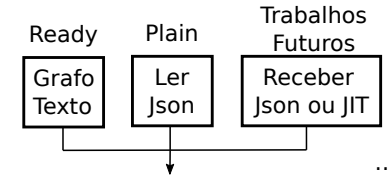
O segundo nível de simulação emula o funcionamento do sistema como um todo, incluindo FPGA. O grafo será alimentado com os dados da aplicação completa descrita em C++. Todo o processo de mapeamento será simulado a nível funcional incluindo as interfaces de I/O e a execução no *overlay*. Entretanto, devido ao nível de detalhamento, a simulação demanda mais tempo para executar.

4.4. Execução

O diferencial das ferramentas PLAIN e READY é o mapeamento do grafo no CGRA em tempo de execução, ou seja de forma dinâmica, além de possibilitar a reconfiguração e execução do acelerador também de forma dinâmica. Este recurso permite a futura criação de compiladores Just-in-Time (JIT). O PLAIN acrescenta o retorno de informações para o usuário sobre a execução: número de ciclos, número de bytes transmitidos, tempo total, dentre outros. Os dados são referentes a execução no FPGA incluindo a transferência de dados e configuração do CGRA. Ou seja, da mesma forma

que o usuário executa a nível de simulação, ele pode executar com um volume maior de dados no modo de execução.

A Figura 5 mostra dois momentos distintos da execução. No primeiro trecho o grafo é mapeado no objeto CGRA. O grafo pode estar descrito na ferramenta PLAIN ou gerado por um compilador JIT. No segundo trecho do código em C++, o objeto CGRA carrega o grafo já mapeado com o método *loadCGRAprogram*. Além disso é necessário acoplar os dados do processador com as entradas do grafo. Neste exemplo os vetores (a, b, c) são inicializados no processador. Depois com os métodos *SetCGRAInputStream* e *SetCGRAOutputStream*, os vetores de dados são acoplados ao acelerador. Vale destacar a facilidade de associar os dados da sua aplicação C++ para enviar/receber dados no acelerador. Finalmente, o método *syncExecute* realiza duas operações. Primeiro, o CGRA é configurado com a transmissão do bitstream para o *overlay* no FPGA. Depois automaticamente, o CGRA começa a requisitar, processar e enviar os resultados através da memória compartilhada entre o processador e o acelerador.




```
map = df->mapped();                 ...
...                                 cgra.loadCgraProgram(map);
Cgra cgra;
int a[1024], b[1024], c[1024];      cgra.setCgraInputStreamByID(df->id, df->in0.id, a, len);
int len = 1024 * sizeof(short);     cgra.setCgraInputStreamByID(df->id, df->in1.id, b, len);
for (int i = 0; i < len; ++i)       cgra.setCgraOutputStreamByID(df->id, df->out.id, c, len);
   a[i] = 0; b[i] = 0; c[i] = 0;
...                                 cgra.syncExecute(0); // Execução
                                    ...
```

Figura 5. Etapa do mapeamento com um exemplo de um trecho do código em C++ gerado automaticamente para executar o CGRA.

5. Resultados

A ferramenta PLAIN amplia e simplifica a ferramenta READY. O resultado final traz vantagens no processo de desenvolvimento de soluções de aceleradores em hardware modelados com grafos, pois todo o processo de elaboração e síntese foi abstraído em um processo automatizado com edição gráfica em um navegador e execução em hardware com CGRA *overlay* já carregado no FPGA. Reduzindo assim os esforços necessários para elaboração de soluções de aceleradores em hardware acoplados a processadores de alto desempenho.

A ferramenta PLAIN exibe um relatório de execução. A Figura 6 ilustra o relatório do exemplo da aplicação *Kmeans* e a Tabela 1 mostra o desempenho da execução de 4 exemplos em comparação com a execução em um Processador XEON com 1 e com 8 *threads* respectivamente. O CGRA pode executar até 128 operações por ciclo, explorando paralelismo espacial e temporal. Portanto, aplicações como o FIR e KMEANS, que são grafos com um grande número de operadores e reuso dos dados, terão mais desempenho. Se o grafo tiver poucas operações ou for limitado pelo baixo reuso dos dados, o CGRA terá uma baixa ocupação, resultando em uma taxa menor de Gops/s. Todos os exemplos

da Tabela 1 tem o mesmo tamanho de dados na entrada e portanto, o tempo de execução é bem próximo.

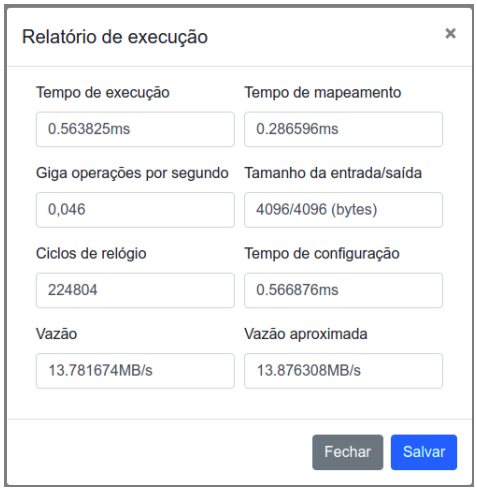


Figura 6. Relatório de Execução da Ferramenta PLAIN para aplicação FIR.

Tabela 1. Tempo de Compilação e Reconfiguração para as 4 aplicações.

Bench	Tempo Compilação+Configuração (µs)	Tempo de Execução CGRA (ms)	CGRA Gop/s	Tempo de Execução Xeon (ms)		Xeon Gop/s
				1 Thread	8 Threads	
fir	778,6	42.9	24	360.8	84.9	11,9
kmeans	793,6	42.3	16,4	285.4	72.5	9,5
paeth	792,2	42.5	7,2	50.0	17.5	17,8
sobel	784,4	42.3	7	277.8	68.2	4,3

Para comparar o tempo de desenvolvimento e projeto com as abordagens tradicionais com síntese de alto nível, as 4 aplicações foram descritas em C++ na ferramenta HLS da Intel e compiladas com a versão 19.4. O Verilog gerado foi sintetizado com Quartus 19.4. A Tabela 2 mostra o tempo de compilação (C++ $\rightarrow$ Verilog) e o tempo de síntese (Verilog $\rightarrow$ FPGA) e os recursos utilizados. Podemos observar uma diferença significativa nos tempos de compilação e síntese em comparação com Ferramenta PLAIN variando de 5 a 6 ordens de grandeza, que são apresentados na coluna *ganho* da Tabela 2. O *ganho* é a razão do tempo gasto para compilar/sintetizar em HLS comparado com o tempo para compilar e reconfigurar no PLAIN. A última linha mostra os recurso gastos pelo CGRA. Apesar de consumir 46% do FPGA, além de ganhar no tempo de mapeamento, o tempo de execução é bem próximo pois o CGRA executa em pipeline com uma frequência de 200 Mhz. O CGRA é compilado uma única vez em tempo de projeto, portanto, esta operação é transparente para o usuário e não afeta o tempo de compilação e execução dos novos algoritmos que podem ser avaliados.

6. Trabalhos Relacionados

A Tabela 3 resume a comparação com outras ferramentas e abordagens para desenvolvimento de aceleradores. A coluna *Comp* exige a ordem de grandeza para compilação,

Tabela 2. Benchmarks no Intel HLS: Tempo de Compilação e Síntese, Recursos gastos no FPGA. Comparação com o CGRA *overlay*.

Bench	Tempo Comp.	Tempo Sintese	Recursos do FPGA				Ganho
			ALMs	BRAM(bits)	DSPs	FMax(MHz)	
fir	22s	14m33s	20.838 (5%)	2.048 (<1%)	20 (1%)	196	9×10^5
kmeans	6m23s	1h21m	139.776 (33%)	1.783.232 (3%)	0 (0%)	192,94	5×10^6
paeth	34s	16m02s	24.236 (6%)	547.968 (<1%)	0 (0%)	232,56	1×10^6
filtro Sobel	40s	30m52s	57.544 (13%)	698.240 (1%)	96 (6%)	237,42	2×10^6
CGRA	-	3h19m23s	195.532 (46%)	5.892.848 (11%)	128 (8%)	200	-

assim como as colunas *Config* para a configuração dinâmica e *Sint* para síntese. A coluna *Nav* se a ferramenta oferece suporte para navegador. A coluna *overlay* se executa como *Overlay* em um FPGA e a coluna *Cloud* se executa na nuvem. A coluna *DFG* mostra se o grafo é explícito ou implícito. Finalmente a coluna *D/E* se a ferramenta oferece recursos gráficos para depuração e relatórios de execução para refinamentos.

Primeiro, a ferramenta ADD [Penha et al. 2019] permite a descrição de grafos de forma explícita, porém gera um código Verilog que deve ser sintetizada, que pode demorar horas, para ser depois implementado no FPGA. O compilador CCF [Dave and Shrivastava 2017] não requer síntese, extraí o grafo de um código C/C++, porém requer minutos para compilar e não foi validado em FPGA. A execução foi validada em um processador ARM utilizando a ferramenta de simulação Gem5. No PLAIN para o mesmo benchmark, o tempo de mapeamento é da ordem de milissegundos.

Já a ferramenta CGRA-ME foi integrada a um gerador de *overlay* por [Chin et al. 2018] para executar sobre um FPGA, mas o trabalho não apresenta medidas de tempo de execução. A compilação demora minutos/horas por usar programação inteira com soluções exatas, além de ser restrita a grafos com no máximo 25 operadores. O CGRA alvo tem apenas 16 elementos de processamento, portanto a 200 MHz só irá ter no máximo 200M*16 = 3,2 Gops/s de desempenho, que é 8x menor que o desempenho do PLAIN que requer apenas milissegundos para mapear/compilar. Por fim, nenhuma destas ferramentas tem suporte para execução na nuvem e nem facilidades de desenvolvimento com implementações em navegadores.

A segunda parte da Tabela 3 mostra soluções de ferramentas comerciais. Primeiro, o uso de linguagem de síntese de alto nível com HLS da Intel, HLS da Xilinx e a ferramenta Legup [Nane et al. 2015]. A principal vantagem é o uso de C/C++ para descrição. Entretanto existe um abismo entre o código e a implementação, que exige conhecimentos de hardware do programador para escrever um bom código. Além disso, todas tem um alto custo para compilação e ajustes finos que podem requerer horas ou dias. A abordagem com OpenCL também compartilha estas vantagens e desvantagens. Outro ponto que o grafo do acelerador é gerado de forma indireta ou implícita. Tanto a Xilinx como a Intel oferecem ferramentas para visualização do grafo gerado, estimativas de uso de recursos do FPGA e da memória com interface gráfica em suas ferramentas de projeto. Porém, caso

haja a necessidade de alteração no grafo, só é possível por meio de mudança no código e, desta maneira indireta, depende do compilador para regerar o grafo e assim prosseguir na ferramenta. Por outro lado, a abordagem PLAIN/READY trabalha explicitamente com os grafos, podendo modificá-lo facilmente executando em um navegador.

A Maxeler [Stanojević et al. 2019] permite a descrição explícita porém a etapa de síntese é necessária e pode demorar minutos/horas. Ela trabalha com o grafo semi-explícito, ou seja, o grafo é descrito junto com o código com uma extensão de JAVA. Para modificações existe a necessidade de recompilar todo o processo. A Maxeler também oferece uma ferramenta para desenvolvimento com interface gráfica e visualização dos grafos gerados.

Tabela 3. Comparação de características entre as ferramentas.

Ferramentas	Comp	Config	Nav	Sint	overlay	D/E	Cloud	DFG
ADD	seg-min	-	-	min/hs	-	sim	-	E
CCF	min-hs	-	-	min/hs	-	-	-	I
CGRAME	min-hs	-	-	hs	sim	-	-	E
HLS	seg	-	-	min/hs	-	sim	sim	I
OpenCL	seg	-	-	hs	-	sim	sim	I
Maxeler	seg	-	-	min/hs	-	sim	sim	E
Ready	ms	ms	-	ms	sim	-	sim	E
Plain	ms	ms	sim	ms	sim	sim	sim	E

Portanto, a ferramenta READY reúne vantagens das abordagens anteriores com mapeamento e configuração em milissegundos, uso transparente do FPGA sem necessidade de síntese e descrição explícita. A contribuição da ferramenta PLAIN foi simplificar ainda mais o processo, encapsular o ambiente para executar em um navegador, permitir a exploração de novos operadores com dois modos de simulação e gerar relatórios de execução real (incluindo tempos de configuração do FPGA, considerando transmissão de dados, falhas de cache, etc.) para refinamento do algoritmo do acelerador.

7. Conclusão

O desenvolvimento de aplicações com FPGAs ainda tem muitos desafios. A ferramenta PLAIN apresentada neste trabalho simplifica o ensino e pesquisa na construção de algoritmos modelados com fluxo de dados para execução em aceleradores com FPGA. O uso de um CGRA como *overlay* reduz o tempo de compilação e evita a reprogramação do FPGA. Atualmente, o uso de FPGA em nuvem apresenta um novo desafio de segurança. Neste aspecto, o *overlay* isola o acesso físico ao FPGA. Como trabalhos futuros, um ponto importante é o conjunto de operadores e operadores de domínio específico para obter desempenho com eficiência energética. O FPGA HARP 2 que foi avaliado tem o consumo de 23 Watts. Como novos operadores pode-se obter de 100 a 200 Gops/s, ou seja, uma eficiência de até 10 Gops/W.

Referências

Chin, S. A., Niu, K. P., Walker, M., Yin, S., Mertens, A., Lee, J., and Anderson, J. H. (2018). Architecture exploration of standard-cell and FPGA-Overlay CGRAs using the open-source CGRA-ME framework. In *Int. Symposium on Physical Design*.

Dave, S. and Shrivastava, A. (2017). CCF: A CGRA compilation framework. https://github.com/MPSLab-ASU/ccf. Acessado em: 2020-08-11.

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

Ferreira, R., Vendramini, J., and Nacif, M. (2011). Dynamic reconfigurable multicast interconnections by using radix-4 multistage networks in fpga. In *IEEE International Conference on Industrial Informatics*.

Franz, M., Lopes, C. T., Huck, G., Sumer, O., and Bader, G. D. (2016). Cytoscape. js: a graph theory library for visualisation and analysis. *Bioinformatics*, 32(2).

Intel (2020). Intel Xeon with integrated FPGA systems at PC^2. https://wikis.uni-paderborn.de/pc2doc/HARP2. Acessado em: 2020-08-11.

JSON (2020). Introducing json. https://www.json.org/json-en.html. Acessado em: 2020-07-25.

Krommydas, K., Sasanka, R., and Feng, W.-c. (2016). Bridging the FPGA programmability-portability gap via automatic opencl code generation and tuning. In *Int Conf on Application-specific Systems, Architectures and Processors (ASAP)*.

Luebbers, E., Liu, S., and Chu, M. (2020). Simplify software integration for fpga accelerators with opae.

Mutigwe, C. and Aghdasi, F. (2013). Instruction set usage analysis for application-specific systems design. *Int'l Journal of Information Technology and Computer Science*, 7(2).

Nane, R., Sima, V.-M., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y. T., Hsiao, H., Brown, S., Ferrandi, F., et al. (2015). A survey and evaluation of fpga high-level synthesis tools. *IEEE Trans. on CAD of Integrated Circuits and Systems*.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, 6(2):40–53.

Nowatzki, T., Gangadhar, V., Ardalani, N., and Sankaralingam, K. (2017). Stream-dataflow acceleration. In *Int. Symposium on Computer Architecture (ISCA)*.

Penha, J., Silva, L., Silva, J., Coelho, K., Baranda, H., Nacif, J., and Ferreira, R. (2019). ADD: Accelerator design and deploy-a tool for FPGA high-performance dataflow computing. *Concurrency and Computation: Practice and Experience*, 31(18).

Silva, L. B. D., Ferreira, R., Canesche, M., Menezes, M. M., Vieira, M. D., Penha, J., Jamieson, P., and Nacif, J. A. M. (2019). READY: A fine-grained multithreading overlay framework for modern CPU-FPGA dataflow applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–20.

Stanojević, I., Kovačević, M., and Šenk, V. (2019). Application of maxeler dataflow supercomputing to spherical code design. In *Exploring the DataFlow Supercomputing Paradigm*, pages 133–168. Springer.

Wijtvliet, M., Waeijen, L., and Corporaal, H. (2016). Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In *Int. Conf. on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*.