Geração automática de *hardware* a partir de programas descritos em linguagem C com *pragmas*

Lucas F. Porto[1], **Ricardo Menotti**[1]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – 13.565-905 – São Carlos – SP – Brazil

{lucas.porto, menotti}@dc.ufscar.br

Abstract. *Compilers for high-level synthesis have been popularized. They allow quickly and easily transform high-level source code to hardware. Current solutions do not generate hardware able to explore techniques to improve pipeline in hardware. This work shows the LALPC compiler that uses techniques to explore parallel processing in FPGAs as from designs described in C language. The techniques allow identify and apply optimizations on loops in source code. LALPC is able to generate high-performance systems while enable design space exploration by the programmer.*

Resumo. *Compiladores de síntese de alto nível vem se popularizando. Esses permitem transformar códigos de alto nível em hardware de maneira simples e rápida. As soluções atuais geram hardware que não exploram as técnicas que permitam melhorar o pipeline em hardware. Este trabalho apresenta o compilador LALPC que utiliza técnicas para explorar paralelismo em FPGAs a partir de projetos descritos em linguagem C. As técnicas permitem identificar e aplicar otimizações para acelerar trechos de códigos baseados em loops. LALPC é capaz de gerar sistemas de alto desempenho, permitindo a exploração de espaço de projeto pelo programador.*

1. Introdução

A computação reconfigurável, mais especificamente o uso dos FPGAs (*Field-Programmable Gate Array*) modernos, tem oferecido um enorme potencial para o desenvolvimento de sistemas altamente paralelos e com baixo consumo de energia. A presença de *hardware* e *software*, caracterizada pelas arquiteturas híbridas, permite acelerar seções críticas em *hardware*, elevando o desempenho, enquanto mantém um alto nível de produtividade no desenvolvimento já que o restante do sistema é projetado em *software*, muitas vezes já disponíveis para a aplicação desejada [Liao et al. 2010].

Nos sistemas embarcados a combinação *software/hardware* é fortemente acoplada, seja usando um processador *softcore* no próprio dispositivo reconfigurável, ou por meio de dispositivos e kits que combinam FPGAs com processador. Neste contexto, podemos mencionar dispositivos como os da família Zynq da Xilinx [Santarini 2011] e o kit DE2i-150 da Altera/Intel [Terasic 2013]. Já na computação de alto desempenho o mais comum é acoplar aos servidores kits de FPGAs via PCIe ou interfaces proprietárias, nos quais seções críticas do código são aceleradas em *hardware*. Dentre os mais recentes podemos citar [Putnam et al. 2014] e muitos outros apresentados em [Vanderbauwhede and Benkrid 2013].

Apesar do bom desempenho em termos de paralelismo e consumo de energia, a computação reconfigurável ainda não é amplamente usada, principalmente pela dificuldade de se projetar o *hardware* e integrá-lo ao *software*, no caso dos sistemas híbridos. Ferramentas de síntese de alto nível, usadas para transformar descrições mais abstratas diretamente em circuitos, são propostas desde a década de 90 na tentativa de facilitar o desenvolvimento. Após mais de 20 anos, finalmente começam a surgir ferramentas mais maduras, que passaram a ser quase uma necessidade, principalmente pelas seguintes razões: i) rápida exploração do espaço de projeto a partir de especificações funcionais; ii) enorme capacidade dos dispositivos atuais, exigindo um maior nível de abstração para projetos grandes; iii) melhora da produtividade por reúso de componentes; iv) verificação em nível de sistema; e v) tendência para o uso extensivo de aceleradores e SoCs heterogêneos [Cong et al. 2011].

Este trabalho descreve a implementação do compilador LALPC, bem como os resultados experimentais obtidos com o seu uso. O restante do artigo está organizado como se segue. Na Seção 2 são apresentados trabalhos relacionados da área. Na Seção 3 é apresentada a estrutura do compilador e suas principais características. Na Seção 4 são expostos os resultados experimentais obtidos, comparando-os com outras ferramentas. Na Seção 5 são apresentadas as conclusões deste trabalho.

2. Trabalhos Relacionados

Nesta seção são abordadas duas ferramentas de síntese de alto nível recentes. A primeira delas é o compilador LegUp [Canis et al. 2011], por se tratar do projeto *open source* de maior relevância científica no momento. A segunda é o compilador LALP, trabalho prévio do grupo de pesquisa, cujas técnicas de geração de *hardware* foram usadas neste trabalho [Menotti et al. 2012].

2.1. LegUp

O compilador LegUp é uma ferramenta de código aberto voltada para síntese de alto nível utilizando como entrada a linguagem C. O código é processado, compilado automaticamente e tem como saída sistemas em *hardware* descritos em Verilog. O LegUp utiliza o *front-end* do compilador LLVM para efetuar os passos iniciais de análise e otimização do código para posterior síntese. Nesse compilador é implementado um novo *back-end* que usufrui das técnicas de otimizações disponíveis no LLVM, visando melhorias significativas no *hardware* resultante [Canis et al. 2011].

Com o LegUp é possível sintetizar grande parte do código C para *hardware*. Isso permite a manipulação de várias estruturas, tais como: arranjos multidimensionais, variáveis globais, ponteiros, valores em ponto flutuante. O *hardware* gerado pela ferramenta LegUp tem qualidade comparável com o gerado pelas ferramentas comerciais disponíveis para síntese de alto nível. Além disso, ele permite a integração de vários aceleradores com programas descritos em OpenMP e Pthread [Choi et al. 2013].

2.2. LALP

O compilador LALP surgiu como uma nova opção no processo de síntese de alto nível com foco em otimizar códigos baseados em laços de repetição. O compilador permite o mapeamento dos laços de repetição de maneira que sejam identificadas as dependências

entre os componentes, o que permite o balanceamento das operações. Esse procedimento melhora o desempenho nos laços de repetição, impactando diretamente no *loop pipeline* da aplicação. A região do *kernel* dos laços de repetição no código normalmente pode ser paralelizada uma vez que o código interno de um laço de repetição é executado repetidamente durante a execução da aplicação. LALP processa os laços de repetição para gerar arquiteturas em *hardware* com alta performance em processamento paralelo [Menotti et al. 2012].

A ferramenta permite exploração de espaço de projeto por meio de diretivas de marcação no código. Estas marcações são utilizadas para controlar o *clock* das operações gerando arquiteturas diferentes para o mesmo código. Outra característica importante é que o compilador LALP utiliza uma linguagem específica para descrição do algoritmo. A linguagem utilizada, denominada LALP (*Language for Aggressive Loop Pipelining*), possui um nível de abstração maior em relação à linguagem VHDL (*VHSIC Hardware Description Language*). Contudo, a utilização de uma linguagem especifica limita sua utilização uma vez que o seu aprendizado demanda tempo e disponibilidade do programador. O compilador apresentado neste trabalho não demanda a utilização de uma linguagem específica uma vez que é baseado na linguagem C.

2.3. Outros Trabalhos

Outra abordagem interessante neste contexto é a do ReflectC, uma ferramenta de síntese de alto nível com foco em alta flexibilidade no projeto. O compilador utiliza a ferramenta LARA por meio de programação orientada a aspectos, o que permite o usuário especificar informações adicionais do projeto em um arquivo separado. Essas parametrizações feitas no padrão LARA permitem tanto a aplicação de características específicas quanto a aplicação de otimizações no *hardware* gerado [Cardoso et al. 2013, Coutinho et al. 2013a, Coutinho et al. 2013b]. Diversas ferramentas de síntese de alto nível têm sido propostas, o que demonstra o interesse e pesquisas na área de geração automática de *hardware* para dispositivos reconfiguráveis, entre as quais podemos citar Vivado [Feist 2012], CHiMPS [Putnam et al. 2008], ROCCC [Buyukkurt et al. 2006] e Haydn-C [Coutinho and Luk 2003].

3. LALPC

O compilador apresentado neste trabalho é denominado de LALPC. Esse compilador é baseado no compilador LALP descrito na seção anterior e utiliza os mesmos conceitos de escalonamento e as técnicas de paralelismo para otimização do *hardware*. Além disso, o LALPC utiliza a mesma biblioteca de componentes VHDL utilizada no LALP. No entanto, o LALPC se diferencia do LALP ao utilizar códigos descritos na linguagem C como entrada do compilador, ao invés de utilizar uma linguagem específica. Essa característica é fundamental para sua utilização uma vez que a linguagem C é altamente difundida entre programadores.

O compilador LALPC utiliza a análise do *front-end* do compilador ROSE no processo de síntese de alto nível, efetuando todas as validações necessárias no código de entrada nos padrões ANSI C. O compilador ROSE conta com técnicas de otimização para analisar e modificar laços de repetição, tais como *loop fusion*, *loop fission*, *loop blocking*, *loop interchange* e *loop unrolling* [Quinlan 2000]. A técnica denominada *loop unrolling* é utilizada no compilador LALPC e é descrita em detalhes na Seção 3.4.

O Compilador LALPC como em LALP permite utilizar diretivas de marcação para auxiliar o compilador durante o processo de geração de *hardware*. Essas diretivas de marcação são descritas por meio de *pragmas* inseridos no código que são interpretadas pelo compilador LALPC. A utilização dos *pragmas* permite gerar arquiteturas diferentes com características específicas para o mesmo código fonte de entrada. Com os *pragmas* o programador aplica customizações no *hardware* sem a necessidade de conhecimento específico de implementação. O uso dos *pragmas* no compilador LALPC é descrito em detalhes na Seção 3.3.

A ferramenta LALP necessita que o programador utilize as diretivas de marcação para efetuar o sincronismo das operações. Esse sincronismo garante que o processo de escalonamento seja executado corretamente em alguns exemplos. O compilador LALPC suporta o uso da técnica de sincronização das operações utilizando *pragmas*. Entretanto, a necessidade de definir o sincronismo em exemplos complexos foi resolvido no compilador LALPC modificando o processo de balanceamento em conjunto com a técnica SSA (*Static Single Assignment*) do compilador ROSE. Nos testes realizados neste trabalho não foram necessários os *pragmas* de sincronismo para que o *hardware* gerado calculasse corretamente os resultados, diferentemente do compilador LALP no qual algumas sincronizações manuais eram necessárias.

A utilização das características supracitadas permitem ao programador gerar, de maneira simples, sistemas de *hardware* de alto desempenho baseados em programas descrito em C. O compilador LALPC constitui uma alternativa para a simplificação do processo de geração automática de *hardware* mantendo a exploração de processamento paralelo em arquiteturas reconfiguráveis.

3.1. Estrutura do Compilador

O LALPC instancia partes do compilador ROSE para validar o código de entrada em linguagem C e aplicar algumas técnicas de otimização. Após esta etapa, uma AST (*Abstract Syntax Tree*) com a representação intermediária do código é obtida. Esta representação é utilizada para gerar os componentes de *hardware* necessários e definir as ligações de dados entre eles. A biblioteca de componentes, escrita em VHDL, é a mesma usada pelo compilador LALP [Cardoso 2000].

O compilador ROSE permite efetuar análises e otimizações no código de entrada. Uma dessas análises é a SSA que modifica o nome das variáveis do código conforme necessário, tornando cada atribuição única. Essa otimização facilita o processo para a geração dos componentes de *hardware* de maneira correta. Essa característica supriu a necessidade encontrada no compilador LALP que o usuário deve definir no código novas variáveis toda vez que esta receber uma nova atribuição.

No decorrer do processamento é verificado a existência dos *pragmas* no código. Os *pragmas* como citado anteriormente são responsáveis pelas customizações no *hardware*. Ao identificar os *pragmas* o compilador LALPC pode modificar a representação intermediária diretamente no *front-end* do compilador ROSE e/ou modificar os componentes VHDL. Após criar os sinais entre os componentes, o compilador executa as técnicas de balanceamento semelhantes ao compilador LALP. Por fim, o compilador gera os arquivos VHDL com os componentes e suas respectivas ligações. Na Figura 1 é possível observar a estrutura do compilador LALPC.

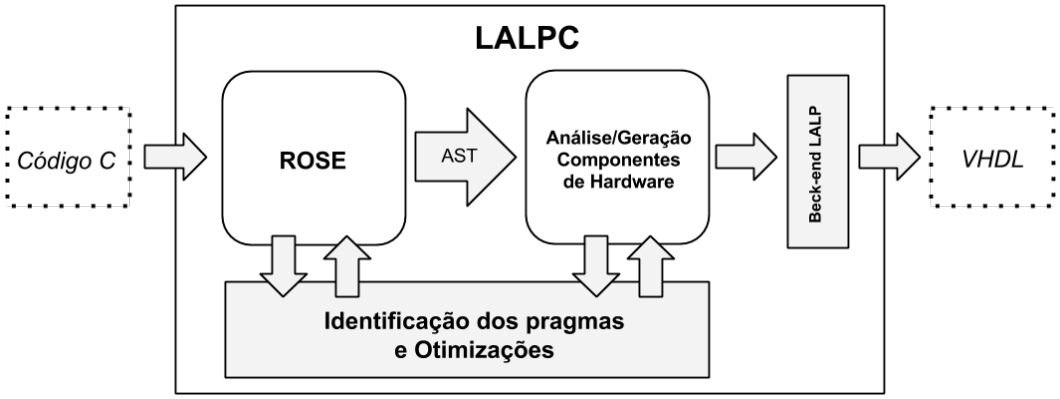

Figura 1. Estrutura do compilador LALPC.

3.2. Subconjunto Suportado

O compilador LALPC tem como objetivo gerar sistemas de alto desempenho utilizando técnicas de paralelismo em laços de repetição. Além disso, o programador pode utilizar as seguintes estruturas no processo de geração de *hardware*: verificação condicional IF em bloco e ternário, execução de operações aritméticas, acumuladores, vetores, deslocamento de bits, desdobramento de laços de repetição utilizando *loop unrolling*, memórias RAM (*Random Access Memory*) customizadas para acessos simultâneos, diretivas de marcação utilizando *pragmas*, definir largura de bits da aplicação, definir pinos de saída no *hardware*.

Diversas ferramentas de síntese de alto nível, algumas usadas comercialmente, tem limitações oriundas das diferenças entre os paradigmas de *software* e *hardware*. Estas diferenças podem fazer a ferramenta de síntese gerar resultados com semântica diferente da esperada ou até gerar arquiteturas que necessitam de uma grande quantidade de recursos de *hardware*. Modificar a sintaxe ou simplesmente reestruturar o código original costuma ser a solução para estes casos.

O compilador LALPC conta com algumas limitações impedindo à utilização de todos os recursos disponíveis na linguagem C, sendo elas: dados em ponto flutuante, chamadas de funções, *structs*, ponteiros, matrizes, recursão, entre outras. Para estes casos é necessário o programador efetuar modificações no código de entrada adaptando-o para que este seja processado corretamente pelo compilador. Estas limitações de entrada são muito comuns em ferramentas de síntese de alto nível.

3.3. *Pragmas*

O processo de utilização das diretivas de marcação por *pragmas* tem como objetivo permitir ao programador definir algumas características específicas durante o processo de compilação. O uso dos *pragmas* modifica o *hardware* gerado pelo compilador, podendo resultar em arquiteturas diferentes. Para o mesmo código de entrada, podem ser obtidos diferentes desempenhos, bem como uso menor ou maior de recursos do dispositivo reconfigurável, consumo de energia, etc.

O uso dos *pragmas* é justificado não apenas pelas características e limitações existentes na linguagem C quando usada para especificação de *hardware*, mas também para

que o programador possa fazer otimizações específicas para suprir uma necessidade em um projeto.

Um exemplo que representa uma diferença de paradigmas entre a linguagem C e uma arquitetura em *hardware* seria a necessidade de utilizar várias saídas para a mesma função. Neste caso no compilador LALPC, é possível definir com os *pragmas* quais variáveis poderão ter pinos de saída nos seus respectivos registradores.

A seguir são descritos os *pragmas* tratados pelo compilador LALPC atualmente:

- **#pragma alp unroll** - *Pragma* responsável para aplicar no código a otimização de *loop unrolling*, permitindo um ganho em processamento paralelo na aplicação.
- **#pragma alp data_width** - *Pragma* responsável por definir largura de bits utilizados pelos componentes VHDL.
- **#pragma alp bit** - *Pragma* responsável por definir se um registrador será do tipo booleano.
- **#pragma alp multiport** - *Pragma* responsável por verificar se existe múltiplos acessos em uma memória permitindo criar uma memória RAM customizada.
- **#pragma alp out** - *Pragma* responsável por criar pinos de saída nos componentes VHDL, isso facilita a análise dos dados e permite gerar um *hardware* com várias saídas.
- **#pragma alp delay** - *Pragma* responsável controlar os ciclos de *clock* dos componentes, permitindo o sincronismo manual das operações.

3.4. Exploração do Espaço de Projeto

A utilização dos *pragmas* permite ao programador aplicar diferentes técnicas para gerar variações das arquiteturas de *hardware*. Os *pragmas* multiport e unroll são usados para aplicar técnicas de otimizações visando acelerar o processamento da aplicação. Entretanto, seu uso influencia na quantidade de recursos ocupados no dispositivo reconfigurável, como se pode constatar a seguir.

3.4.1. #pragma alp multiport

Este *pragma* permite gerar um componente de memória RAM com múltiplas portas, permitindo acessos simultâneos. Isso permite melhorar o desempenho final da aplicação em muitos casos, pois sem o acesso simultâneo é preciso utilizar um multiplexador para controlar o acesso ao conteúdo da memória sequencialmente.

Tabela 1. Recursos em memórias multiport geradas pelo compilador LALPC.

Portas	Elem. Lóg	Comb.	Reg.	Mem. (bits)
1	55	43	46	32768
2	63	53	47	27648
4	69	58	55	24832
8	107	87	87	20096

Na Tabela 1 é possível constatar o impacto em termos de recursos ocupados no dispositivo quando memórias customizadas são usadas. A diferença na arquitetura é significativa quando se compara a memória com apenas uma porta à uma memória que permite

2, 4 ou 8 leituras/escritas simultâneas. Isso ocorre porque durante o processo de síntese a ferramenta Quartus II move os valores da memória RAM, realocando-os para elementos lógicos [Altera 2009]. Essa diferença pode ser constatada observando-se a última coluna da Tabela 1. Quanto mais portas são selecionadas para a memória RAM, menos bits são efetivamente alocados em memória interna do dispositivo e mais vão para elementos lógicos.

3.4.2. #pragma alp unroll

Ao utilizar este *pragma* o compilador analisa as operações dentro do laço tentando identificar dependências entre as mesmas. Tais dependências impedem aplicar a técnica de desenrolar o laço de repetição pois uma operação depende do valor da interação anterior. Por consequência, o processamento deste laço de repetição é obrigatoriamente sequencial.

Não existindo nenhuma dependência entre as operações, o compilador efetua cópias das operações internas do laço de repetição permitindo que estas sejam executadas de maneira paralela. O resultado é um ganho significativo no processamento da aplicação quando comparado com a aplicação sequencial. Por outro lado, o processo de replicar as operações internas aumenta a quantidade de recursos ocupados no dispositivo reconfigurável proporcionalmente à quantidade de operações contidas no laço.

O Código 1 visa exemplificar a utilização deste *pragma*, na Linha 4 temos a informação do *pragma* e o fator que deve ser aplicado para o desdobramento deste laço de repetição. Já no Código 2 temos o resultado do desdobramento por parte do *front-end* do compilador ROSE. Essa passa a ser a nova estrutura na representação intermediária do código que será utilizado na geração dos componentes de *hardware* no compilador LALPC.

Código 1. Repetição inicial

```
1 foo() {
2   int i, x[N], y[N], z[N];
3   #pragma alp multiport
4   #pragma alp unroll 2
5   for (i = 0; i < N; i++)
6     z[i] = x[i] + y[i];
7 }
```

Código 2. Repetição resultante

```
1 foo() {
2   int i, x[N], y[N], z[N];
3   for (i = 0; i < N; i+=2)
4     z[i]   = x[i]   + y[i];
5     z[i+1] = x[i+1] + y[i+1];
6 }
```

No Código 2 é possível constatar que a linha 5 foi gerada a partir da linha 4, porém com acessos diferentes às memórias, o que permite o processamento paralelo de duas iterações a cada ciclo.

Ainda neste exemplo, ao aplicar o *pragma* unroll no código, este gera mais acessos simultâneos para as memórias, sendo representados pelos índices dos vetores "i" e "i+1". Desta maneira é possível combinar o *pragma* multiport para acesso simultâneo à esta memória. No Código 1 na Linha 3 temos o *pragma* que indica ao compilador LALPC customizar as memórias quando encontrar acessos simultâneos para esta, independente se for de leitura ou escrita.

A combinação dos dois *pragmas* supracitados acarreta em ganho significativo de desempenho para a aplicação, principalmente se comparado ao processamento de uma

aplicação sequencial. Outro ponto a ser observado é que a utilização ou não de tais conceitos é uma escolha do programador. A seguir são apresentados os resultados experimentais.

4. Resultados experimentais

O processo para avaliar o desempenho do *hardware* gerado pelo compilador LALPC é baseado na comparação de desempenho e recursos com o *hardware* gerado pelos compiladores LALP e LegUp. Uma lista com *benchmarks* com características distintas representam os códigos de entrada para estes compiladores, sendo eles, os algoritmos de audio *ADPCM Coder* e *ADPCM Decoder* [Guthaus et al. 2001]; o algoritmo de processamento de imagem *Sobel* [Texas 2003b]; os algoritmos *Dotprod*, *Max* e *Vecsum* utilizados em DSPs [Texas 2003a]; o algoritmo *Accumulator* presente dentro da pasta de testes do compilador LegUp [Canis et al. 2011], e outros algoritmos como o filtro de imagem *Fir* e *Fibonacci* que dispõem de características interessantes para serem exploradas pelo compilador. Para o processo de síntese foi utilizada a ferramenta Quartus II 13.0 (64-bit) e o dispositivo reconfigurável (FPGA) Altera EP2C35F672C6 da Família Cyclone II.

No trabalho proposto em LALP, o autor demonstra o ganho de desempenho da ferramenta comparando-a com outras ferramentas de síntese de alto nível. Nos testes com os mesmos *benchmarks*, o compilador LALPC demonstrou ter resultados bastante semelhantes aos obtidos com o LALP. O compilador LALPC se utiliza de técnicas reimplementadas do compilador LALP, o que justifica os resultados semelhantes [Menotti et al. 2012].

Apesar de praticamente não haver ganho em relação aos resultados obtidos com LALP, cabe ressaltar a facilidade em se descrever os algoritmos em C. Além disso, parâmetros de sincronismo informados manualmente em LALP puderam ser inferidos automaticamente no compilador LALPC. Na Tabela 2 são apresentados os resultados dos testes entre os compiladores com os *benchmarks* citados anteriormente.

Os *pragmas* citados anteriormente resultam na aplicação das otimizações no código de entrada, quando possível. Para demonstrar a eficiência das técnicas implementadas no compilador, foi utilizado o *benchmark Sobel*, comparando-se as arquiteturas geradas com e sem as otimizações. O *benchmark Sobel* foi escolhido pois permite efetuar os testes com os *pragmas* separadamente, pois no algoritmo são realizados 8 acessos de leitura no vetor de entrada e uma única gravação no vetor de saída com o resultado do processamento.

Ao efetuar os 8 acessos na memória RAM é necessário utilizar um multiplexador para gerenciar o controle de acesso na memória, tornando essa etapa de leitura sequencial. Nos testes esse processo demanda 8 ciclos para leitura dos *pixels* da memória por iteração do laço de repetição. Ao utilizar o *pragma* multiport o compilador transforma essa memória RAM com uma única porta de endereçamento e de saída, para uma memória RAM com 8 portas de endereçamento e 8 portas de saída. Ao efetuar essa customização todos os acessos são executados em um único ciclo permitindo um ganho expressivo no tempo de execução da aplicação.

Outra otimização possível de ser aplicada neste exemplo é a de *loop unrolling*, isso é possível pois não existe dependências dentro do *kernel* do laço de repetição. Ao combinar essa otimização com o *pragma* multiport o *hardware* gerado garante um ganho

Tabela 2. Recursos ocupados e tempo de execução nos benchmarks.

Benchmark	Comp	Elementos Lógicos	Comb	Reg	Memória (Bits)	Tempo Exec. (us)
Accumulator	LALP	153	114	142	0	0,07
	LALPC	149	114	143	0	0,07
	LEGUP	1035	927	668	640	0,46
ADPCM Coder	LALP	818	674	37	40960	83,14
	LALPC	986	667	837	41460	117,50
	LEGUP	1072	979	656	19744	197,18
ADPCM Decoder	LALP	508	386	464	41090	14,70
	LALPC	632	417	574	41248	40,56
	LEGUP	1072	979	656	19744	197,26
Dotprod	LALP	96	82	65	131072	26,79
	LALPC	120	104	83	131072	26,47
	LEGUP	802	685	526	131072	130,57
Fibonacci	LALP	373	209	364	0	0,19
	LALPC	146	77	133	0	0,17
	LEGUP	84	84	37	0	0,24
Fir	LALP	74	71	33	0	0,04
	LALPC	100	93	37	0	0,04
	LEGUP	5	1	5	0	0,07
Max	LALP	65	47	62	65356	11,70
	LALPC	92	76	84	65356	13,69
	LegUP	294	273	234	65356	32,91
Sobel	LALP	806	681	504	8269	5,90
	LALPC	1148	753	991	8435	4,84
	LEGUP	1285	1170	820	6400	4,35
Vecsum	LALP	101	53	67	196608	17,52
	LALPC	124	75	84	196608	17,87
	LEGUP	903	802	551	196608	135,90

do tempo de execução quando se comparado no *hardware* gerado apenas com o *pragma* multiport. Essa combinação duplica todas as operações existentes dentro do *kernel* do laço, gerando 16 acessos simultâneos na memória contendo a imagem de entrada e outros 2 acessos de gravação na memória de saída.

Os testes realizados demonstraram o potencial das técnicas de otimização implementadas no compilador LALPC. Os resultados relacionados ao ganho de desempenho são apresentados na Tabela 3. Cabe ressaltar que ao utilizá-las, estas também afetam a quantidade de recursos necessários para a síntese, podendo inviabilizar sua utilização dependendo da limitação de recursos da placa reconfigurável utilizada no projeto. Na Figura 2 é possível visualizar os recursos necessários de *hardware* comparando o teste sem o uso dos *pragmas* com o teste utilizando dois *pragmas* de otimização no *benchmark Sobel*.

Tabela 3. Recursos ocupados e tempo de execução no *benchmark Sobel*.

Comp	Elementos Lógicos	Comb	Reg	Memória (Bits)	Tempo Exec. (us)
LEGUP	1285	1170	820	6400	4,35
LALP	806	681	504	8269	5,90
LALPC	1148	753	991	8435	4,84
LALPC *	913	875	303	4131	0,65
LALPC +	4247	3643	2572	30	0,38

* *pragma* multiport
\+ *pragmas* multiport e unroll com fator 2

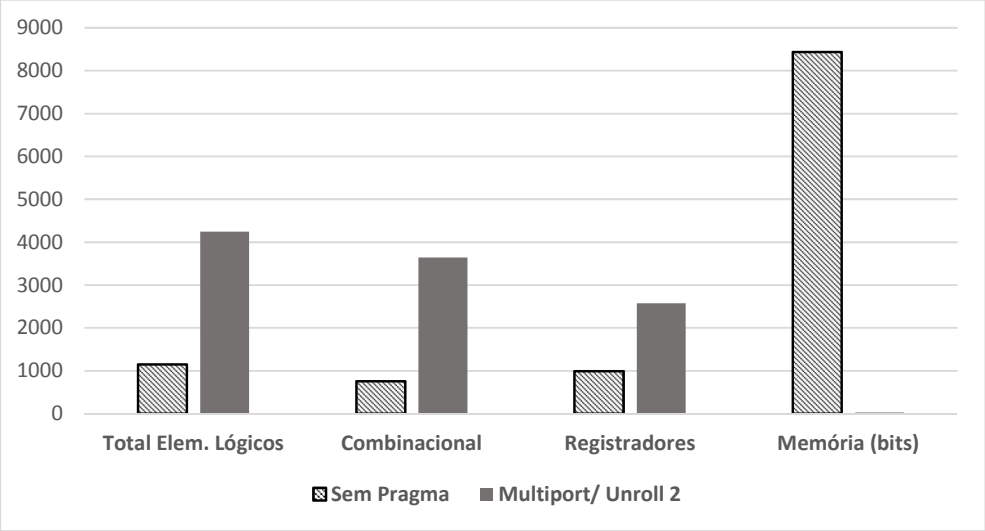

Figura 2. LALPC: Recursos de *hardware* necessários no *benchmark Sobel*

5. Conclusão

A abordagem utilizada neste trabalho para a geração automática de *hardware*, a partir de uma linguagem de alto nível, para computação reconfigurável, demonstrou ser bastante promissora. Apesar das limitações em relação ao subconjunto da linguagem C suportado, o compilador LALPC gerou resultados com uma evolução de desempenho significativa em relação ao compilador LALP. Além disso, o compilador apresentado neste trabalho possui como característica principal a utilização da linguagem C como entrada, ao invés de uma linguagem específica.

O compilador LALP dispõe de diretivas de marcação que permitem ao programador controlar os ciclos de *clock* no código de entrada, possibilitando maior controle sobre o escalonamento das operações em *hardware*. No compilador LALPC, as diretivas de marcação (*pragmas*) permitem o controle dos ciclos de *clock* semelhante ao compilador LALP. O uso de *pragmas* no código fonte possibilita a rápida exploração do espaço de projeto, pois com pequenas modificações na entrada são obtidas arquiteturas diferentes, dentre as quais se pode selecionar a mais adequada em termos de desempenho e recursos ocupados no dispositivo.

O trabalho desenvolvido atendeu as expectativas iniciais de se criar uma ferramenta de síntese de alto nível, aproveitando as técnicas existentes no compilador LALP, porém usando como entrada programas descritos em linguagem C. As técnicas de memórias customizadas e *loop unrolling*, implementadas com o *front-end* do compilador ROSE, permitiram gerar arquiteturas com desempenho superior ao obtido com o compilador LALP. Diversas técnicas implementadas no compilador ROSE podem ainda serem aplicadas à geração de *hardware* no contexto deste trabalho. Além das transformações relativas aos *loops*, as técnicas de reúso de dados podem ser exploradas para aumentar o paralelismo sem a necessidade de mais acessos à memória.

Referências

Altera (2009). *Best HDL Design Practices for Timing Closure (OHDL1130)*. Altera Corporation.

Buyukkurt, B., Guo, Z., and Najjar, W. (2006). Impact of loop unrolling on area, throughput and clock frequency in ROCCC: C to VHDL compiler for FPGAs. *Reconfigurable Computing: Architectures and Applications*, pages 401–412.

Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., and Czajkowski, T. (2011). LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM.

Cardoso, J., Diniz, P., de Figueiredo Coutinho, J., and Petrov, Z. (2013). *Compilation and Synthesis for Embedded Reconfigurable Systems: An Aspect-Oriented Approach*. Springer London, Limited.

Cardoso, J. M. P. (2000). *Compilação de Algoritmos em Java para Sistemas Computacionais Reconfiguráveis com Exploração do Paralelismo ao Nível das Operações*. PhD thesis, Universidade Técnica de Lisboa.

Choi, J., Brown, S., and Anderson, J. (2013). From software threads to parallel hardware in high-level synthesis for fpgas. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 270–277. IEEE.

Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., and Zhang, Z. (2011). High-Level Synthesis for FPGAs: From Prototyping to Deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491.

Coutinho, J., Cardoso, J., Carvalho, T., Nobre, R., Bhattacharya, S., Diniz, P., Fitzpatrick, L., and Nane, R. (2013a). Deriving resource efficient designs using the reflect aspect-oriented approach. In Brisk, P., Figueiredo Coutinho, J., and Diniz, P., editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 7806 of *Lecture Notes in Computer Science*, pages 226–228. Springer Berlin Heidelberg.

Coutinho, J. G. F., Cardoso, J. a. M. P., Carvalho, T., Nobre, R., Bhattacharya, S., Diniz, P. C., Fitzpatrick, L., and Nane, R. (2013b). Deriving resource efficient designs using the reflect aspect-oriented approach. In *Proceedings of the 9th international conference on Reconfigurable Computing: architectures, tools, and applications*, ARC'13, pages 226–228, Berlin, Heidelberg. Springer-Verlag.

Coutinho, J. G. F. and Luk, W. (2003). Source-directed transformations for hardware compilation. In *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, pages 278–285. IEEE.

Feist, T. (2012). Vivado design suite. *Xilinx, White Paper Version*, 1.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001). MiBench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA. IEEE Computer Society.

Liao, C., Quinlan, D. J., Willcock, J. J., and Panas, T. (2010). Semantic-aware automatic parallelization of modern applications using high-level abstractions. *International Journal of Parallel Programming*, 38(5-6):361–378.

Menotti, R., Cardoso, J. M. P., Fernandes, M. M., and Marques, E. (2012). LALP: A Language to Program Custom FPGA-based Acceleration Engines. *International Journal of Parallel Programming*, 40(3):262–289.

Putnam, A., Bennett, D., Dellinger, E., Mason, J., Sundararajan, P., and Eggers, S. (2008). Chimps: A c-level compilation flow for hybrid cpu-fpga architectures. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 173–178. IEEE.

Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P. Y., and Burger, D. (2014). A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24.

Quinlan, D. (2000). ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226.

Santarini, M. (2011). Zynq-7000 EPP sets stage for new era of innovations. *Xcell J*, 75(2):8–13.

Terasic (2013). *DE2i-150 FPGA System User Manual*. Terasic Technologies Inc.

Texas (2003a). *TMS320C64x DSP Library: Programmer's Reference*. Texas Instruments Incorporated.

Texas (2003b). *TMS320C64x Image/Video Processing Library: Programmer's Reference*. Texas Instruments Incorporated.

Vanderbauwhede, W. and Benkrid, K. (2013). *High-Performance Computing Using FPGAs*. Springer.