Ministério da
Ciência, Tecnologia e Inovação

sid.inpe.br/mtc-m19/2012/05.10.19.52-TDI

FAST POISSON SOLVER PARA SISTEMA HÍBRIDO RECONFIGURÁVEL

Vitor Conrado Faria Gomes

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelos Drs. Haroldo Fraga de Campos Velho, e Andrea Schwertner Charão, aprovada em 25 de maio de 2012

URL do documento original:
<http://urlib.net/8JMKD3MGP7W/3BS8H3S>

INPE
São José dos Campos
2012

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/6921
Fax: (012) 3208-6919
E-mail: pubtc@sid.inpe.br

Ministério da
Ciência, Tecnologia e Inovação

sid.inpe.br/mtc-m19/2012/05.10.19.52-TDI

FAST POISSON SOLVER PARA SISTEMA HÍBRIDO RECONFIGURÁVEL

Vitor Conrado Faria Gomes

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelos Drs. Haroldo Fraga de Campos Velho, e Andrea Schwertner Charão, aprovada em 25 de maio de 2012

URL do documento original:
<http://urlib.net/8JMKD3MGP7W/3BS8H3S>

INPE
São José dos Campos
2012

Dados Internacionais de Catalogação na Publicação (CIP)

Gomes, Vitor Conrado Faria.

G585f

Fast poisson solver para sistema híbrido reconfigurável / Vitor Conrado Faria Gomes. – São José dos Campos : INPE, 2012.

xxii + 83 p. ; (sid.inpe.br/mtc-m19/2012/05.10.19.52-TDI)

Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2012.

Orientadores : Drs. Haroldo Fraga de Campos Velho, e Adrea Schwertner Charão.

1. fast possion solver. 2. Transformada Rápida de Fourier (FFT). 3. computação híbrida. 4. cray XD1. I.Título.

CDU 004.272.3

Aprovado (a) pela Banca Examinadora em cumprimento ao requisito exigido para obtenção do Título de Mestre em

Computação Aplicada

Dr. Stephan Stephany

Presidente / INPE / SJCampos - SP

Dr. Haroldo Fraga de Campos Velho

Orientador(a) / INPE / São José dos Campos - SP

Dra. Andrea Schwertner Charao

Orientador(a) / UFSM / Santa Maria - RS

Dr. Eduardo Marques

Convidado(a) / USP / São Paulo - SP

Este trabalho foi aprovado por:

( ) maioria simples

(X) unanimidade

Aluno (a): Vitor Conrado Faria Gomes

São José dos Campos, 25 de maio de 2012

"The best race car drivers understand how their cars work. The best architects know how carpenters, bricklayers, and electricians do their jobs. And the best programmers know how the hardware they are programming does computation".

Mark L. Chang
em *"Reconfigurable Computing"*, 2008

A minha amada companheira Luciane

AGRADECIMENTOS

Primeiramente gostaria de agradecer a minha familia, em especial: minha companheira Luciane que me apoiou durante todo o período de realização desse trabalho e me deu força nos momentos difíceis; meus pais que me servem de exemplo e sempre me incentivaram e me apoiaram nos estudos; meu irmão que me deu suporte para a concretização dos meus objetivos.

Aos meus colegas do Curso de Computação Aplicada, companheiros de estudo do laboratório do LAC, em especial, meus amigos Amarísio Araújo, que com sua paciência interminável muito me ensinou e ao Rodolfo Lotte pela motivação diária.

Agradeço aos meus orientadores, Dr. Haroldo F. de Campos Velho e Dra. Andrea S. Charão, que me orientaram, me motivaram e souberam proferir as palavras certas nos momentos de dúvida.

Gostaria de agradecer também às pessoas que anonimamente dedicam parte do seu tempo na criação de software ou obras intelectuais livres, as quais indiretamente deram suporte ao desenvolvimento deste trabalho.

Por fim, agradeço à FAPESP pelo apoio financeiro.

RESUMO

A computação híbrida é uma técnica que vem ganhando destaque no cenário de computação alto desempenho, sendo considerada uma alternativa viável para sistemas que requerem processamento intensivo. Com essa abordagem, é possível utilizar dispositivos arquiteturalmente diferentes que cooperam com processadores de propósito geral na solução de aplicações críticas. No contexto de computação híbrida reconfigurável, FPGAs são configurados como coprocessadores especializados para o processamento de núcleos intensivos. De maneira geral, a solução de problemas científicos necessita de uso intensivo de processamento. Neste trabalho, tem-se por objetivo explorar algoritmos eficientes integrados a arquiteturas de alto desempenho reconfiguráveis, para conhecer o potencial do uso de FPGAs combinados com processadores de propósito geral. Para isso, é proposta uma arquitetura para uma solução da classe de *fast Poisson solver*, que é aplicada em um sistema híbrido reconfigurável. Para essa solução, foi projetado e implementado um coprocessador baseado em FPGA para realizar a computação da transformada de Fourier, que é a região mais intensiva da aplicação. A fim maximizar o uso deste recurso, foi desenvolvido um esquema de endereçamento de memória específico para a computação da FFT. As soluções apresentadas nesse trabalho são avaliadas sob três diferentes abordagens. A primeira avaliação é feita no contexto do desempenho da solução híbrida do *fast Poisson solver* em relação a uma aplicação de referência. A segunda avaliação é realizada do ponto de vista da solução em hardware, onde o número de ciclos é utilizado como métrica. Por fim, foi realizada uma avalição de desempenho da execução da FFT em FPGA frente a uma tradicional biblioteca em software, considerando os tempos medidos e uma estimativa calculada. Os resultados obtidos nesses testes apresentam ganhos de desempenho para as execuções utilizando FPGA como coprocessador, sendo repetido esse comportamento nas estimativas realizadas.

A FAST POISSON SOLVER FOR HYBRID RECONFIGURABLE SYSTEM

ABSTRACT

Hybrid computation technique has gained prominence in the high-performance computing scenario, and is considered a viable option for systems that require intensive processing. With this approach, different devices can cooperate with general purpose processors in the solution of critical applications. In the context of hybrid reconfigurable computing, FPGAs are configured as specialized coprocessors for the processing of core intensive. Generally, the solution of scientific problems need intensive processing. The main goal of this work is explore efficient algorithms integrated with high-performance reconfigurable architectures, to meet the potential of using FPGAs combined with general purpose processors. For this, we propose an architecture for a solution of the class of fast Poisson solver, which is implemented in a reconfigurable hybrid system. For it, we designed and implemented a FPGA-based coprocessor to perform the computation of the Fourier transform, which is the most intensive region of application. In order to maximize the use of this resource, we developed a specific scheme for addressing memory for computing the FFT. The solutions presented in this work are evaluated under three different approaches. The first evaluation is made in the context of the performance of hybrid solution of fast Poisson solver relative to a reference application. The second evaluation is performed in terms of hardware solution, where the number of cycles is used as metric. Finally, we conducted a performance evaluation of the implementation of FFT in FPGA compared to a traditional library software, considering the time measured and a calculated estimate. The results of these tests show performance gains for the executions using FPGA as coprocessor, and repeated this behavior in the estimates.

LISTA DE FIGURAS

Pág.

2.1 Estrutura básica de um FPGA. 7
2.2 Estrutura típica de um CLB. 7
2.3 Fluxo típico de configuração para FPGA. 9
2.4 Arquitetura típica de um sistema híbrido reconfigurável. 9
2.5 Fluxo típico de desenvolvimento. 10
2.6 Fluxo computacional para FFT de 8 pontos 15

3.1 Mapeamento de tarefas e fluxo entre FPGA e CPU para solução de *fast Poisson solver* 20
3.2 Foto de um equipamento Cray XD1. 21
3.3 Arquitetura e Visão lógica de um *blade* do Cray XD1 22
3.4 Interfaces do RapidArray Transporte Core. 24
3.5 Estrutura do *template* de desenvolvimento para aplicações para FPGA do Cray XD1. 25
3.6 Diagrama da arquitetura do coprocessador FFT para FPGA 27
3.7 Agrupamento dos sinais dos barramentos de dados 29
3.8 Diagrama da Unidade de Controle 31
3.9 Diagrama da Unidade de Comunicação 33
3.10 Diagrama da Unidade de Computação 35
3.11 Diagrama da Unidade de Constantes Trigonométricas 37
3.12 Esquema de endereçamento para gravação de dados 42
3.13 Esquema de endereçamento para leitura de dados 43

5.1 Gráfico com os tempos de execução para as versões em software e híbrida da função POIS1D (eixo vertical em escala logarítmica) 57
5.2 Eficiência da transferência de dados entre CPU e FPGA 62
5.3 Gráfico da quantidade de operações em ponto flutuante por segundo computadas pela Unidade de Computação 64
5.4 Gráfico com os tempos das execuções da FFT usando a biblioteca FFTW e o coprocessador FFT (eixo vertical em escala logarítmica) 66
5.5 Gráfico com a quantidade de operações em ponto flutuante por segundo das execuções da FFT usando a biblioteca FFTW e o coprocessador FFT 67

LISTA DE TABELAS

Pág.

3.1 Posição dos dados após distribuição sequencial em 2, 3 ou 4 bancos de memória 39
3.2 Sequência de acesso de FFT para 8 pontos 41
3.3 Destaque para 1°e 2°passos do padrão de acesso para FFT de 8 pontos . 41
3.4 Destaque para 2°e 3°passos do padrão de acesso para FFT de 8 pontos . 41
3.5 Opções de síntese utilizada no Ise Foundation 10.1 45
3.6 Dados da síntese do coprocessador FFT 46

5.1 Tempos médios de execução para as versões em software e híbrida da função POIS1D 56
5.2 Número médio de ciclos e desvio padrão das etapas do coprocessador FFT 59
5.3 Eficiência calculada para o recebimento e envio de dados entre FPGA e CPU 61
5.4 Quantidade de operações em ponto flutuante por segundo computadas pela Unidade de Computação 63
5.5 Tempos médios de execução de FFT usando o coprocessador FFT em FPGA e a biblioteca FFTW em CPU 65
5.6 Quantidade de operações em ponto flutuante por segundo computadas usando o coprocessador FFT em FPGA e a biblioteca FFTW em CPU . 67

LISTA DE ABREVIATURAS E SIGLAS

API	–	Application Program Interface
ASIC	–	Application Specific Integrated Circuit
CLB	–	Configuration Logical Blocks
CPU	–	Central Processing Unit
DSP	–	Digital Signal Processor
FFT	–	Fast Fourier Transform
FIFO	–	First In, First Out
FLOPS	–	Floating point Operations Per Second
FPGA	–	Field Programmable Gate Array
GFLOPS	–	10^9 Floating point Operations Per Second
GPU	–	Graphics Processing Unit
MFLOPS	–	10^6 Floating point Operations Per Second
MEF	–	Máquina de Estados Finitos
IOB	–	Input/Output Block
RAM	–	Random Access Memory
ROM	–	Read Only Memory
RTL	–	Register Transfer Level
SGI	–	Silicon Graphics, Inc.
SRAM	–	Static Random Access Memory
VHDL	–	VHSIC Hardware Description Language
VHSIC	–	Very-High-Speed Integrated Circuit

SUMÁRIO

Pág.

1 INTRODUÇÃO 1

2 CONCEITOS E TECNOLOGIAS 5

2.1 Computação Híbrida 5

2.1.1 Dispositivos Lógicos Programáveis 5

2.1.2 Computação Híbrida Reconfigurável 8

2.2 Fast Poisson Solver 11

2.2.1 Transformada Rápida de Fourier 12

2.3 Trabalhos Relacionados 14

3 PROJETO DE FAST POISSON SOLVER HÍBRIDO 19

3.1 Equipamento alvo: Cray XD1 21

3.1.1 Hierarquia de Memória 22

3.1.2 Transferência de Dados 23

3.1.3 Desenvolvimento 25

3.2 Coprocessador FFT 25

3.2.1 Barramentos de Dados 28

3.2.2 Unidade de Controle 29

3.2.3 Unidade de Comunicação 32

3.2.4 Unidade de Computação 34

3.2.5 Unidade de Contantes Trigonométricas 36

3.2.6 Controlador de Memória 38

3.2.6.1 Endereçamento de Memória para FFT 40

3.3 Implementação 44

3.4 Biblioteca FPSH 47

4 APLICAÇÃO 49

4.1 Modelo DYNAMO 49

4.2 Integração com o *Fast Poisson Solver* Híbrido 53

5 RESULTADOS E DISCUSSÕES 55

5.1 *Speedup* do *Fast Poisson Solver* Híbrido 55

5.2 Métricas do coprocessador FFT 58

5.3 Coprocessador FFT *vs* FFTW 64

6 CONCLUSÕES 69

6.1 Trabalhos Futuros 70

6.2 Outras aplicações de FPGA 71

REFERÊNCIAS BIBLIOGRÁFICAS 73

1 INTRODUÇÃO

O computador eletrônico surgiu como uma máquina automática para realizar computações aritméticas em grande escala, tarefa tediosa e sempre sujeita a erros, quando realizada pelos seres humanos. Uma das primeiras aplicações para o computador eletrônico foi a previsão numérica do tempo. Nos anos 30, John von Neumann viu que o progresso da hidrodinâmica seria bastante acelerado se houvesse um meio de resolver equações matemáticas complexas por métodos numéricos. Reconheceu ainda que a previsão meteorológica era um problema suficientemente grande e cientificamente interessante para o computador automático (LYNCH, 2008).

Desde a época de von Neumann, a computação científica teve um grande progresso, permitindo resolver sistemas complexos e realizar previsões numéricas com grande precisão. O aumento da sofisticação dos algoritmos e a necessidade de executar previsões sobre grades computacionais de modelos numéricos com maior resolução, exigem o incremento, muitas vezes em ordem exponencial, do poder computacional. De maneira geral, a solução de problemas científicos necessita de uso intensivo de processamento.

Visando melhorar o desempenho de aplicações na área de computação científica, diversas estratégias têm sido desenvolvidas, sendo a computação paralela a mais difundida. Nesta abordagem, são utilizados diversos processadores de propósito geral para a computação de uma aplicação em paralelo, sendo possível o uso de sistemas com múltiplos *cores* em um único processador, ou diversos computadores interligados em rede. Independente da arquitetura do sistema, é necessário utilizar técnicas e ferramentas especiais para a paralelização de problemas, nem sempre envolvendo tarefas triviais (SHAN, 2006; WILKINSON; ALLEN, 2004; COMISSÃO REGIONAL DE ALTO DESEMPENHO - RS (CRAD-RS), 2006).

Em resposta à necessidade por maior poder computacional, surge um novo paradigma conhecido como Computação Híbrida, que sugere a utilização de diferentes unidades funcionais para a computação de aplicações. Dispositivos como Unidades de Processamento Gráfico (GPU), Processadores Digitais de Sinais (DSP) e/ou *Field Programmable Gate Arrays* (FPGA) são incorporados em sistemas de alto desempenho para se integrarem a processadores de propósito geral na execução de aplicações críticas. O objetivo é permitir que cada dispositivo execute as tarefas com maior afinidade com sua arquitetura, como, por exemplo, a execução de operações matriciais em GPUs. O uso unidades de processamento gráfico tem ganhado destaque nos grandes centros de computação científica, estando presente em 3 dos 5 supercom-

putadores considerados mais rápidos do mundo pelo Top500.org em novembro de 2011 (Top500.org, 2011).

Recursos como GPUs, DSPs e Circuitos Integrados de Aplicação Específica (ASIC), em geral, têm funções especializadas, as quais garantem otimização na solução das aplicações para os quais foram projetados. Entretanto, reduzem a gama de aplicações que podem executar com ganho significativo de desempenho. Via de regra, o aumento de desempenho devido à especialização do dispositivo de hardware está inversamente relacionado à sua flexibilidade na solução de diferentes aplicações.

FPGAs, por sua vez, tentam combinar o alto desempenho de aplicações específicas e a flexibilidade de processadores de propósito geral. Estes dispositivos lógicos programáveis podem ser configurados e reconfigurados para reproduzir comportamentos lógicos específicos. A flexibilidade vem da possibilidade de descrever o funcionamento do FPGA através de linguagens de alto nível e o desempenho é dado pelo processamento de aplicações diretamente em um hardware especializado (GOKHALE; GRAHAM, 2005). No contexto de Computação Híbrida, o uso de FPGAs na computação de aplicações intensivas tem ganhado popularidade na comunidade de computação de alto desempenho. Nesta emergente técnica, conhecida como Computação Híbrida Reconfigurável, FPGAs são configurados para funcionarem como coprocessadores especializados, permitindo incrementar a capacidade computacional de sistemas, aumentar a densidade computacional e reduzir o consumo de energia (PAPAKONSTANTINOU et al., 2009; BRODTKORB et al., 2010; CARDOSO et al., 2010; LEE et al., 2010).

Devido à sua importância para a computação científica, a Computação Híbrida Reconfigurável vem sendo investigada por diversos trabalhos. Alguns tratam propriamente da técnica e de sua viabilidade (HARTENSTEIN, 2001; ANDREWS et al., 2004; WAIN; AL., 2004; GOKHALE; GRAHAM, 2005; SHAN, 2006; GARCIA et al., 2006; CHAMBERLAIN et al., 2007; HAUCK; DEHON, 2007; KINDRATENKO et al., 2007b; CRAVEN; ATHANAS, 2007; CHAMBERLAIN et al., 2008; CHAVARRÍA-MIRANDA et al., 2008; KOEHLER et al., 2008; EL-GHAZAWI et al., 2008; THOMAS et al., 2009; CARDOSO et al., 2010; BRODTKORB et al., 2010; JONES et al., 2010) , enquanto outros focam no desenvolvimento e avaliação de aplicações (FERNANDO et al., 2005; VITE-FRIAS et al., 2005; ZHUO; PRASANNA, 2005; BONDHUGULA et al., 2006; KINDRATENKO; POINTER, 2006; HE, 2007; SHIGUEMORI, 2007; ZHUO; PRASANNA, 2008; HE; GUO, 2008; GOMES et al., 2009b; PAPAKONSTANTINOU et al., 2009; TSOI; LUK, 2010).

O projeto AXEL (TSOI; LUK, 2010) é uma iniciativa de sucesso no que diz res-

peito ao uso de múltiplos recursos computacionais integrados para o processamento de aplicações críticas. Pesquisadores do Departamento de Computação da *Imperial College London* projetaram e construíram um *cluster* heterogêneo com GPUs e FPGAs colaborando com CPUs para aumentar a capacidade computacional do sistema. O grupo de pesquisa responsável pelo AXEL foi o primeiro a demonstrar o uso de FPGAs, GPUs e CPUs atuando colaborativamente na solução de um problema de N-corpos. Na avaliação dessa solução, o uso de FPGAs foi o que apresentou melhor desempenho computacional e de consumo energético (TSOI; LUK, 2010).

São frequentes os trabalhos que apresentam ganhos significativos de desempenho e que apontam a computação híbrida reconfigurável como alternativa viável para a solução de diferentes aplicações. Apesar disso, ainda são limitados os estudos que envolvem a solução de equações diferenciais parciais neste tipo de arquitetura (BRODTKORB et al., 2010). Essas equações são recorrentes em modelos de previsão numérica e geralmente estão associadas à descrição de fenômenos físicos. O estudo das equações que descrevem a atmosfera é uma área muito importante da dinâmica dos fluidos. A discretização de sistemas de equações diferenciais conduz a modelos numéricos que são usados para prever os estados futuros da atmosfera. Este processo computacional é chamado de Previsão Numérica do Tempo (CAMPOS VELHO; CLAEYSSEN, 1992).

Diversos fenômenos físicos podem ser descritos por equações diferenciais parciais elípticas. Em geral, essas equações descrevem um estado de equilíbrio com aplicações em diversas áreas do conhecimento. Em alguns casos, equações elípticas, como as equações de Poisson, aparecem em passos intermediários de modelos de evolução, como em sistemas operacionais de previsão. Modelos de previsão numérica fazem uso intensivo de processamento (JACOBSON, 2005; HOFFMAN, 2001).

A habilidade de resolver eficientemente esses tipos de equações permite que sistemas complexos possam ser processados em tempo hábil para a elaboração de previsões precisas. Contudo, a melhoria da previsão está vinculada ao uso de malhas computacionais mais refinadas. Muitas vezes, o tempo de computação de códigos deste tipo é proporcional ao cubo do número de pontos de grade computacional. Deste modo, há pesquisa permanente na busca de algoritmos mais eficientes. Uma das técnicas que permitem acelerar esse tipo de processamento vincula-se àquelas que empregam a aplicação da transformada rápida de Fourier (FFT: *Fast Fourier Transform*) na solução de equações de Poisson. A FFT realiza o processamento com custo computacional $O(N \log N)$.

Inserido no contexto de explorar algoritmos mais eficientes integrados a arquiteturas

de alto desempenho reconfiguráveis, este trabalho aplica uma classe de *fast Poisson solver* em um sistema híbrido reconfigurável. Os objetivos estão voltados à exploração das possibilidades de sistemas de alto desempenho emergentes e em conhecer o potencial do uso de CPU combinada com FPGA para o processamento de aplicações críticas. Além disso, a meta é permitir que aplicações tradicionalmente codificadas em software possam aproveitar a solução híbrida sem que haja a necessidade de alterações profundas no código.

O restante dessa dissertação está estruturado da seguinte forma: o segundo capítulo apresenta os conceitos e as tecnologias utilizados no nosso trabalho: computação híbrida e *fast Poisson solver*. No terceiro capítulo é apresentado o projeto para a solução híbrida da equação de Poisson. Além disso, o equipamento híbrido alvo é apresentado e o projeto do coprocessador especializado para FPGA é detalhado. Para finalizar o terceiro capítulo, são informadas questões sobre a implementação e a construção de uma biblioteca em linguagem de alto nível. O quarto capítulo é destinado à validação do nosso projeto em um modelo meteorológico de previsão numérica, sendo, inclusive, apresentadas informações sobre a integração com a solução híbrida desenvolvida nessa dissertação. O capítulo seguinte é destinado para a exibição e discussão dos resultados de testes de desempenho que são comparados a uma aplicação de referência. Por fim, são apresentadas as considerações finais, os trabalhos futuros e brevemente listadas duas outras aplicações híbridas desenvolvidas durante a realização deste mestrado.

2 CONCEITOS E TECNOLOGIAS

2.1 Computação Híbrida

Nos últimos anos uma série de recursos computacionais básicos amadureceram suficientemente para contribuírem substancialmente em aplicações de alto desempenho. Estes recursos incluem processadores de propósito geral multi-core, processadores gráficos (*Graphics Processing Unit* - GPU), dispositivos lógicos programáveis, processadores digitais de sinais (*Digital Signal Processor* - DSP) dentre outros processadores de aplicações específicas (*Application-Specific Integrated Circuit* - ASIC). O aumento de desempenho associado ao uso destes recursos é bastante significativo e em muitos casos, chegando a uma ou mais ordens de grandeza (CHAMBERLAIN et al., 2007; CHAMBERLAIN et al., 2008).

A técnica de combinar várias destas unidades funcionais arquiteturalmente diferentes em um único fluxo de trabalho é conhecida como computação híbrida, e os equipamentos que suportam esta tecnologia são conhecidos como sistemas híbridos. Uma aplicação desta técnica é a utilização de GPUs para o processamento gráfico, deixando a CPU livre para o processamento do restante da aplicação. O ganho de desempenho surge da execução de parte da aplicação por uma unidade especializada. Deste modo, o interesse em combinar diferentes recursos na execuções de tarefas críticas é motivado pelo ganho de desempenho que emerge do emprego de cada unidade de computação na tarefa em que está mais adaptada (SHAN, 2006; CHAMBERLAIN et al., 2007; CHAMBERLAIN et al., 2008).

O uso adequado de cada dispositivo está fortemente relacionado à aplicação alvo. Os processadores gráficos, por exemplo, dependem da utilização de um grande número de núcleos de processamento para lidar com uma alta carga de processamento com alto grau de paralelismo de dados, como ocorre no processamento de imagens.

Dispositivos lógicos programáveis, por sua vez, fornecem melhores formas para explorar padrões específicos de paralelismo de dados e de controle. Além disso, é possível configurar dinamicamente seu funcionamento para diferentes estruturas de hardware, permitindo o desenvolvimento mais flexível de processadores específicos para cada tipo de aplicação. Por estas características, dispositivos lógicos programáveis têm ganhado popularidade na comunidade de alto desempenho nos últimos anos (GOKHALE; GRAHAM, 2005; PAPAKONSTANTINOU et al., 2009; CARDOSO et al., 2010).

2.1.1 Dispositivos Lógicos Programáveis

No âmbito da computação, podemos distinguir duas formas principais de realizar computação: por hardware ou por software. A computação por hardware, realizada principalmente através de Circuitos Integrados para Aplicações Específicas (ASICs), fornece recursos altamente especializados e otimizados para a realização de tarefas críticas. O comportamento lógico destes dispositivos é definido durante seu processo de fabricação e não pode ser alterando durante sua vida útil (HAUCK; DEHON, 2007).

A computação em software, por outro lado, fornece flexibilidade para o desenvolvimento e alteração de aplicações para a computação de uma enorme quantidade de tarefas. Entretanto, esta solução é significativamente pior que aplicações em hardware em relação a desempenho, uso de silício e consumo de energia (GOKHALE; GRAHAM, 2005; HAUCK; DEHON, 2007).

Dispositivos lógicos programáveis surgem como uma solução intermediária entre estas formas de computar. Esses dispositivos, que combinam benefícios de hardware e software, podem ser configurados para comportarem como um circuito lógico em hardware. E, diferentemente de ASICs, podem ser reconfigurados diversas vezes para terem diferentes comportamentos lógicos (GOKHALE; GRAHAM, 2005; HAUCK; DEHON, 2007).

Fazem parte desta categoria dispositivos como *Programmable Array Logic* (PAL), *Generic Array Logic* (GAL), *Complex Programmable Logic Device* (CPLD) e *Field Programmable Gate Array* (FPGA). Destes dispositivos, o FPGA é o dispositivo que possui maior flexibilidade devido à abundância e ao tamanho reduzido de suas unidades básicas, o que permite a configuração de sistemas complexos com um custo muito menor quando comparado a um ASIC (HAUCK; DEHON, 2007).

FPGAs são dispositivos semicondutores formados por blocos lógicos configuráveis (CLBs) incorporados em uma estrutura de roteamento programável, conforme a ilustração da Figura 2.1. Estes dois recursos permitem a alteração do comportamento lógico do FPGA e a criação de inúmeros circuitos lógicos (GOKHALE; GRAHAM, 2005; HAUCK; DEHON, 2007).

Os CLBs (Figura 2.2) contêm elementos para a realização de lógica combinacional simples e flip-flops para a realização de lógica sequencial. Estas unidades lógicas são implementadas utilizando blocos de memória, operações booleanas simples e normalmente possuem de 4 a 6 entradas. A estrutura de roteamento programável

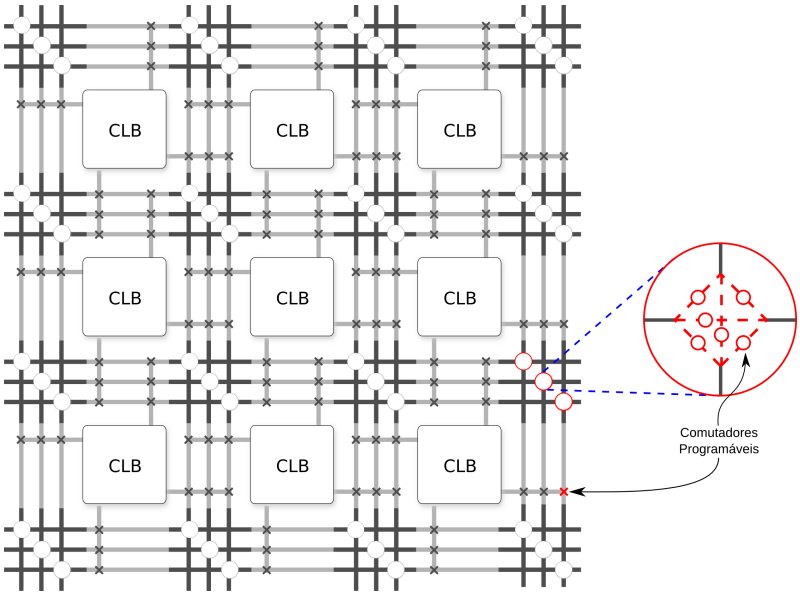


Figura 2.1 - Estrutura básica de um FPGA.
Fonte: Adaptado de Hauck e Dehon (2007)

permite a configuração de ligações arbitrárias, de modo que os elementos lógicos podem ser conectados de forma apropriada (GOKHALE; GRAHAM, 2005; HAUCK; DEHON, 2007).

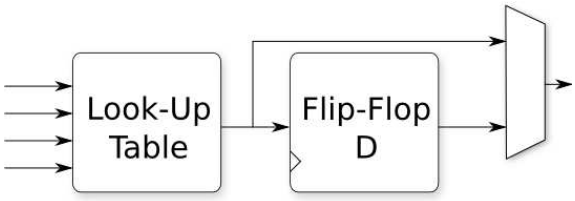


Figura 2.2 - Estrutura típica de um CLB.
Fonte: Adaptado de Hauck e Dehon (2007), Gokhale e Graham (2005)

Para permitir a construção flexível de sistemas complexos, FPGAs atuais contam com milhões de portas lógicas úteis e podem operar em centenas de megahertz. Para aumentar o desempenho e capacidade destes dispositivos, elementos especiais são incorporados, como memórias, multiplicadores, funções lógicas e até microprocessadores. Com estes recursos adicionais, é possível implementar sistemas completos em um único FPGA (HAUCK; DEHON, 2007).

O desenvolvimento de aplicações para FPGAs é tradicionalmente feito através de linguagens de descrição de hardware. Linguagens como VHDL e Verilog permitem a descrição em alto nível do comportamento lógico da aplicação e do fluxo interno de dados (GOKHALE; GRAHAM, 2005). Além dessas, existem outras linguagens, como C2H, System C, System Verilog, etc, que se aproximam mais de linguagens de desenvolvimento de software e, por isso, costumam facilitar o desenvolvimento de aplicações para FPGAs. Algumas dessas linguagens tem propósitos específicos, como a LALP (MENOTTI, 2010), que tem por objetivo permitir a programação de aceleradores eficientes, usando agressivamente *loop pipelining* e sintaxe muito próxima a C ou Java. O autor desse trabalho acredita que em pouco tempo os compiladores de hardware baseados em linguagens de alto nível possam atingir ampla aceitação, mas este ainda é um ponto em aberto e mais pesquisas são necessárias nesta na área (MENOTTI, 2010).

Além de pesquisas de novas linguagens para a descrição de hardware, existem algumas propostas (JASKELAINEN et al., 2010; OWAIDA et al., 2011; Altera Corp., 2011) do uso de OpenCL para a configuração de aplicações para hardware. O OpenCL é um padrão de *framework* de programação paralela para sistemas heterogêneos, sendo atualmente usado mais frequentemente para a programação de GPUs. Esse padrão de desenvolvimento tem uma abordagem mais próxima da natureza paralela de FPGAs, permitindo que os programadores controlem e especifiquem explicitamente o paralelismo das aplicações (CARDOSO et al., 2010; OWAIDA et al., 2011). A Altera, uma das principais fabricantes de FPGA, tem investido em um projeto para permitir a programação de seus FPGAs com OpenCL (Altera Corp., 2011). Apesar dos resultados serem promissores, ainda existe a necessidade de um melhor suporte de ferramentas para sua utilização de forma efetiva.

Além da necessidade de um maior amadurecimento dessas linguagens, o uso de VHDL é vantajoso pela flexibilidade de implementação, por ser uma linguagem padrão (IEEE 1076) e independente da ferramenta de desenvolvimento. Deste modo, uma mesma descrição de projeto pode ser utilizada em diversas arquiteturas de dispositivos (SHIGUEMORI, 2007; HAUCK; DEHON, 2007)

A conversão da descrição de hardware até a configuração do FPGA segue as etapas ilustradas na Figura 2.3. O primeiro passo é a descrição do hardware em VHDL ou Verilog. Este código fonte é sintetizado, transformando a lógica em alto nível em portas lógicas interconectadas. Na sequência, o processo de mapeamento separa as portas lógicas em grupos para serem melhor adaptadas aos recursos lógicos do

FPGA. O quarto passo é o posicionamento, que avalia em qual bloco lógico cada grupo de portas lógicas deve ser configurado. Por fim, no roteamento, são determinadas as interconexões que irão transportar os sinais entre os blocos. Ao final destas etapas é gerado um *bitstream* que contém as informações necessárias para configurar o FPGA com a aplicação projetada (HAUCK; DEHON, 2007).

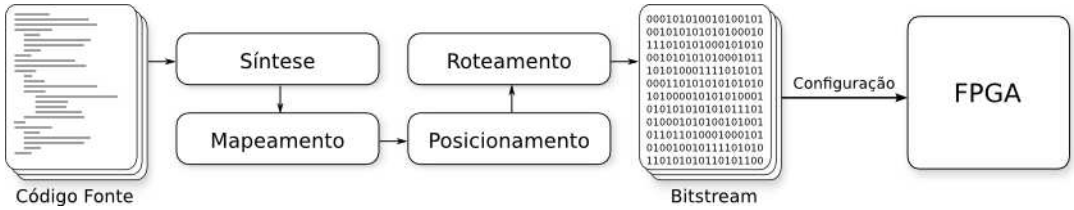


Figura 2.3 - Fluxo típico de configuração para FPGA.
Fonte: Adaptado de Hauck e Dehon (2007)

2.1.2 Computação Híbrida Reconfigurável

Computação Híbrida Reconfigurável é a subárea da Computação Híbrida que utiliza FPGAs combinados com processadores de propósito geral para a computação de aplicações intensivas. Sistemas Híbridos Reconfiguráveis incorporam em suas arquiteturas estes dispositivos na forma de coprocessadores especializados para realizarem tarefas críticas (CARDOSO et al., 2010; CHAMBERLAIN et al., 2008; GOKHALE; GRAHAM, 2005).

Diversos sistemas híbridos reconfiguráveis foram lançados na última década e têm demonstrado capacidade de obter ganho de desempenho em diversas aplicações. Tipicamente estes sistemas apresentam a arquitetura apresentada na Figura 2.4, onde a CPU e o FPGA têm acesso à memória principal, podem comunicar-se através de um canal de comunicação e o FPGA está conectado a bancos de memória local (CARDOSO et al., 2010; BRODTKORB et al., 2010). O modo como é implementada a comunicação entre estes recursos depende do sistema híbrido e comumente são utilizadas soluções proprietárias exclusivas.

A programação para sistemas híbridos reconfiguráveis não é uma tarefa trivial, devido ao duplo paradigma enfrentado para a implementação de aplicações para este tipo de arquitetura. O programador, além de desenvolver o software da aplicação, precisa assumir o papel de projetista de hardware, aprendendo uma linguagem de descrição de hardware e um paradigma totalmente diferente da programação sequen-

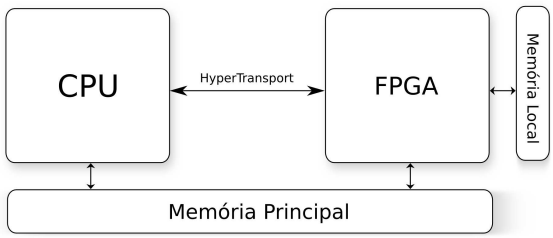


Figura 2.4 - Arquitetura típica de um sistema híbrido reconfigurável.
Fonte: Adaptado de Brodtkorb et al. (2010)

cial clássica. Além disso, precisa conhecer os detalhes técnicos do sistema híbrido alvo para aproveitar todos os recursos disponíveis (KOEHLER et al., 2008; CARDOSO et al., 2010).

O desenvolvimento de aplicações para sistemas híbridos reconfiguráveis é dependente do equipamento alvo. Entretanto, segue um esquema geral para a maioria dos equipamentos (Cray Inc., 2005c; KINDRATENKO et al., 2007a; TSOI; LUK, 2010; Convey Computer Corp., 2011). Na Figura 2.5 é apresentado um fluxograma que mostra a sequência de etapas do desenvolvimento de aplicações híbridas reconfiguráveis. O fluxo esquerdo representa o desenvolvimento da aplicação para o FPGA. O projeto lógico do hardware parte da descrição em linguagem de alto nível e segue para o processo de criação do *bitstream*. Um importante ponto nesta etapa é a utilização da interface de comunicação que é disponibilizada pelo fabricante do sistema. Esta entidade fornece os sinais que devem ser manipulados para realizar a comunicação com os demais recursos do sistema (memória, CPU, outros FPGAs, etc).

Após as etapas de síntese, mapeamento, posicionamento e roteamento da descrição de hardware, o *bitstream* gerado necessita ser preparado para a arquitetura do sistema antes de ser carregado no FPGA. Após estas etapas, o arquivo está pronto para ser carregado no FPGA e iniciar sua execução.

O fluxo do lado direito da Figura 2.5 define o desenvolvimento da aplicação em linguagem de alto nível (normalmente C ou Fortran) que será executada no(s) processador(es) de propósito geral. A comunicação com o FPGA é estabelecida através de uma biblioteca de comunicação que é disponibilizada pelo fabricante do equipamento. A aplicação deve ser programada para carregar o arquivo de configuração no FPGA e para gerenciar, quando preciso, a transferência de dados entre os dispositivos.

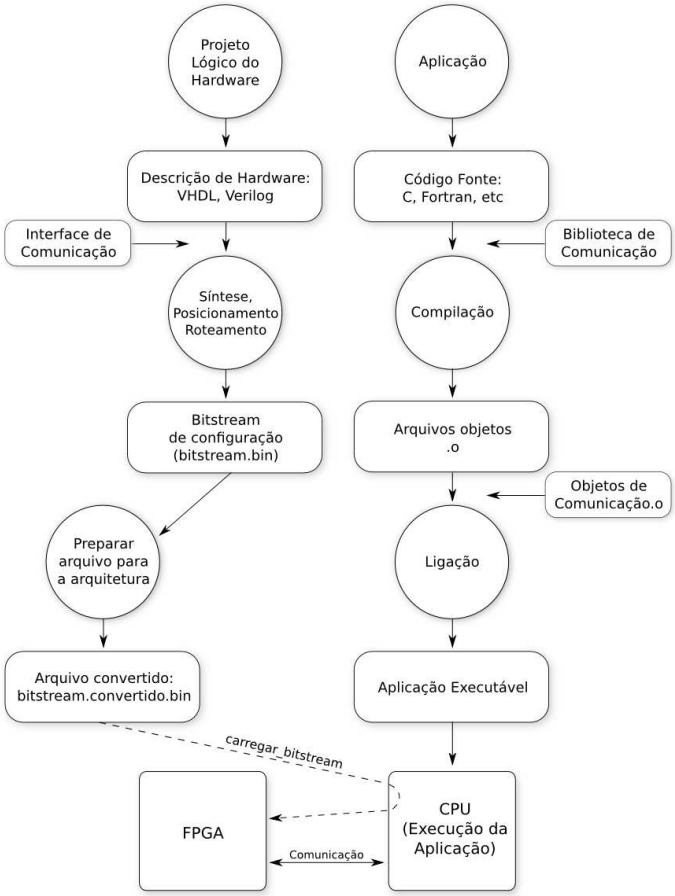


Figura 2.5 - Fluxo típico de desenvolvimento.
Fonte: Adaptado de Cray Inc. (2005c)

2.2 Fast Poisson Solver

Equações de Poisson são equações diferenciais parciais elípticas aplicadas em áreas como difusão de massa, difusão de calor e fluxo de fluido incompressível. A equação de Poisson é a versão não homogênea da equação de Laplace e, em duas dimensões, tem a forma

$$\nabla^2 u(x, y) = f(u, y). \tag{2.1}$$

Em geral, essas equações descrevem um estado de equilíbrio e têm aplicações em engenharia e ciência. A solução de cada ponto desse problema, que envolvem aplicações de valores de contorno em domínios fechados, depende e influencia na solução de todos os outros pontos do domínio, incluindo o contorno (HOFFMAN, 2001).

A solução dessas equações, frequentemente, está associada à região crítica de apli-

cações de áreas que vão da solução de problemas de otimização à modelos de previsão numérica (CAMPOS VELHO; CLAEYSSEN, 1992; YANG et al., 2011). Técnicas que permitem resolver eficientemente essas equações, tornam possíveis que sistemas complexos possam ser processados em tempo hábil para a elaboração de soluções precisas. Contudo, a melhoria da solução está vinculada ao uso de malhas computacionais mais refinadas. Muitas vezes, o tempo de computação de aplicações deste tipo é proporcional ao cubo do número de pontos de grade computacional. Deste modo, há pesquisa permanente na busca de algoritmos mais eficientes.

Algoritmos que reduzem o custo computacional da solução de equações de Poisson são chamados de *fast Poisson solvers*. As técnicas que estão entre as mais rápidas para acelerar esse tipo de processamento vincula-se àquelas que empregam análise de Fourier (GIRAUD et al., 1999). Apesar de existirem outras técnicas com o mesmo propósito, como o uso da transformada seno, se destacam os algoritmos que usam a transformada de Fourier (HOCKNEY, 1965; BUZBEE, 1973; GIRAUD et al., 1999; QIAN; SAPATNEKAR, 2010; YANG et al., 2011).

O uso da transformada de Fourier em *fast Poisson solver* é investigado desde a década de 60, sendo que o trabalho de Hockney (1965) é considerado um dos mais importantes nessa área. Esse trabalho apresenta a solução de equações de Poisson usando análise de Fourier em um domínio retangular, onde as condições de contorno podem ser Dirichlet, Neumann ou periódicas. O princípio básico do método de Fourier apresentado neste trabalho é a expansão da solução em termos de autofunções do operador de Laplace do problema. Este princípio é aplicado em matrizes em blocos que surgem da discretização em diferenças finitas de certas equações diferenciais parciais lineares. Os resultados apresentados indicam um ganho de velocidade na ordem de 10 vezes quando comparado com uma solução interativa.

Uma solução ainda mais eficiente pôde ser obtida após a apresentação por Cooley e Tukey (1965), no mesmo ano do trabalho de Hockney (1965), de um algoritmo rápido para o cálculo da transformada de Fourier. Esse algoritmo, chamado de Transformada Rápida de Fourier (FFT - *Fast Fourier Transform*), reduz a complexidade computacional da transformada de Fourier e é considerado um dos 10 mais importantes algoritmos do século 20 (CIPRA, 2000). O emprego da FFT em *fast Poisson solvers* fornece uma solução que reduz o custo computacional da solução de equações de Poisson de $O(N^2)$ para $O(N \log N)$.

2.2.1 Transformada Rápida de Fourier

A transformada de Fourier é uma transformada linear usada em diversas aplicações científicas. Em sua formulação discreta, esta transformada é usualmente núcleo computacional de aplicações como processamento de sinais, reconhecimento geométrico (KATCHALSKI-KATZIR et al., 1992) e solução de equações parciais. A transformada discreta de Fourier (DFT - *Discrete Fourier Transform*) de uma sequência de N números pode ser computada como

$$F_k = \sum_{j=0}^{N-1} \omega_N^{jk} f_j \qquad\qquad k = 0, 1, ...N-1, \tag{2.2}$$

onde

$$\omega_N = e^{2\pi i/N}, \tag{2.3}$$

são constantes chamadas de raízes complexas da unidade, também conhecidas como fatores de giro (Press et al., 1992; CORMEN et al., 2001).

Usando uma abordagem dividir para conquistar e tirando proveito de propriedades especiais das raízes complexas da unidade, o algoritmo da Transformada Rápida de Fourier (COOLEY; TUKEY, 1965) computa a transformada discreta de Fourier reduzindo sua complexidade de $O(N^2)$ para $O(NlogN)$.

O princípio básico do algoritmo da FFT é dividir sucessivamente F_k em polinômios formados pelos coeficientes de índice par e os coeficientes de índice ímpar para definir dois novos polinômios de limite de grau $N/2$, F_k^e e F_k^o:

$$\begin{aligned} F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \\ &= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \\ &= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + \omega_N^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} \\ &= F_k^e + \omega_N^k F_k^o. \end{aligned} \tag{2.4}$$

Essa técnica divide o cálculo de uma DFT_N de N elementos em dois cálculos de duas $DFT_{N/2}$ de $N/2$ elementos de forma que avaliar F_k em $\omega_N^0, \omega_N^1, \ldots, \omega_N^{N-1}$ se

reduz a avaliar os polinômios de limite de grau $N/2$ nos pontos

$$(\omega_N^0)^2, (\omega_N^1)^2, \ldots, (\omega_N^{N-1})^2. \tag{2.5}$$

Devido às propriedades das raízes complexadas da unidade, onde

$$\begin{aligned}(\omega_N^{k+N/2})^2 &= \omega_N^{2k+N} &(2.6)\\ &= \omega_N^{2k}\omega_N^N &(2.7)\\ &= (\omega_N^k)^2, &(2.8)\end{aligned}$$

a lista 2.5 não consiste em N valores distintos, mas somente nas $N/2$ raízes $(N/2)$-enésimas complexas da unidade, com cada raiz ocorrendo exatamente duas vezes. Assim, os polinômios de limite de grau $N/2$ são avaliados recursivamente nas $N/2$ raízes $(N/2)$-enésimas complexas da unidade. Esses subproblemas apresentam exatamente a mesma forma do problema original, mas têm metade do tamanho. Desta forma, essa divisão pode ser feita recursivamente, permitindo que a redução aplicada a F_k seja também empregada em F_k^e e F_k^o. Esse processo pode ser realizado até não ser mais possível subdividir o polinômio (Press et al., 1992; CORMEN et al., 2001).

Embora existam muitas formas de tratar outros casos, a mais fácil, certamente, é quando N é um inteiro potência de 2. Se o tamanho dos dados não for potência de dois, o conjunto de dados pode ser completado com zeros até a potência de dois mais próxima (Press et al., 1992). Com o algoritmo da FFT estruturado dessa forma, ao final das subdivisões restam dois elementos que devem ser processados. Devido ao formato do símbolo que ilustra essa operação (ver Figura 2.6), ela é conhecida como borboleta e, quando opera sobre dois elementos, é do tipo radix-2. Essa operação consiste em uma multiplicação, uma soma e uma subtração complexas. Se considerarmos f_a^{in} e f_b^{in} cada entrada dessa operação, os resultados f_a^{out} e f_b^{out} da computação são dados por

$$\begin{aligned}f_a^{out} &= f_a^{in} + f_b^{in} * \omega\\ f_b^{out} &= f_a^{in} - f_b^{in} * \omega,\end{aligned} \tag{2.9}$$

onde ω é a mesma constante complexa apresentada em 2.3.

A Figura 2.6 ilusta o fluxo de computação da FFT para 8 pontos. A operação borboleta é representada por cada retângulo desta figura. Nesta ilustração é possível observar que a ordem dos índices do vetor de entrada não é sequencial. Isto acontece devido as sucessivas subdivisões do conjunto de entrada entre os índices pares e

ímpares. Se representarmos esses índices em base binária, podemos considerar que a cada subdivisão, o bit menos significativo (0 ou 1) é usado para testar em qual polinômio será destinado cada coeficiente. Essa sequência resulta em uma ordenação dos dados de entrada em bit reverso. Por exemplo, na posição de índice 1 (001_2) do vetor de entrada, deve estar o elemento que anteriormente ocupava o índice 4 (100_2) e vice-versa. Essas trocas de posições devem ser realizadas para todos os elementos de entrada, sendo que os elementos das extremidades ($00\ldots0_2$ e $11\ldots1_2$) permanecerão, logicamente, em suas posições iniciais.

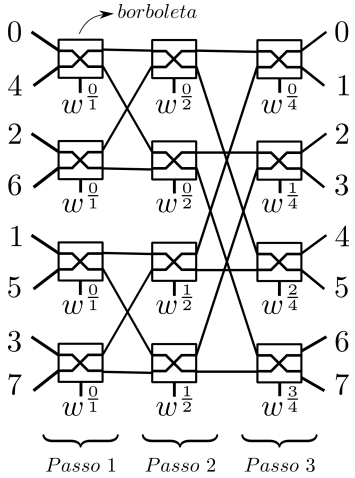


Figura 2.6 - Fluxo computacional para FFT de 8 pontos

Em uma FFT para 8 pontos são necessários 3 ($\log 8$) passos para completar o processamento da transformada. Em cada um desses passos, como ilustrado na Figura 2.6, são realizadas $N/2$ operações borboleta, as quais não possuem dependência de dados entre si. Ao final do último passo, o resultado é fornecido na ordem sequencial dos índices do conjunto de entrada.

2.3 Trabalhos Relacionados

Diversos trabalhos (COLONY; REYNOLDS, 1970; GOLUB et al., 1998; QIAN; SAPATNEKAR, 2010; YANG et al., 2011) apresentam o emprego de *fast Poisson solvers* na solução de problemas específicos em diversas áreas. Alguns desses trabalhos, avaliam o ganho de desempenho de aplicação críticas quando são usados *fast Poisson solvers* em relação a soluções usadas tradicionalmente. Outros trabalhos, propõem novas metodologias para o emprego eficiente de *fast Poisson solvers* em aplicações com características específicas.

Yang et al. (2011) apresentam a aplicação de um *fast Poisson solver* pré-condicionado para o projeto e a otimização de redes de energia não estruturadas com condições de contorno não ideais. Esse trabalho apresenta resultados experimentais que indicam redução de 1/3 no total de iterações em relação a solução considerada clássica, atingindo *speedup* de 20 vezes.

Trabalhos como de Golub et al. (1998) e Qian e Sapatnekar (2010) apresentam novos métodos que permitem a aplicação de *fast Poisson solvers* em aplicações específicas. Qian e Sapatnekar (2010) propõem um método de *fast Poisson solver* que usa o método do gradiente conjugado pré-condicionado para resolver eficientemente domínios 3D não retangulares. Golub et al. (1998) apresentam um *fast Poisson solver* para solucionar equações de Navier-Stokes incompressíveis.

Apesar de tratarem do mesmo tema de pesquisa dessa dissertação, esses trabalhos têm uma abordagem diferente da proposta de nosso trabalho: aplicação de *fast Poisson solver* em arquiteturas de alto desempenho híbridas. Apesar disso, esses trabalhos servem de motivação e justificam a relevância do tema.

Existem trabalhos que tratam *fast Poisson solvers* em arquiteturas de alto desempenho. Esses trabalhos, em sua maioria, estão focados em aplicações paralelas para *clusters* de computadores ou placas gráficas (GPUs). Para nosso conhecimento, ainda não há investigações em projetos que aproveitam o poder de usar sistemas híbridos reconfiguráveis para calcular esse tipo de aplicação.

Shi et al. (2009) usa GPUs para resolver um problema em duas dimensões de rede de energia estruturada. É proposto um novo algoritmo que usa a transformada rápida de Fourier para implementar um *fast Poisson solver*. Os autores avaliam o desempenho do método proposto frente a duas aplicações clássicas. Essas aplicações são testadas em CPU e em GPU. O método proposto atinge aceleração de 10 vezes em relação aos demais, quanto comparadas as versões em CPU. Para os testes com GPU, é obtido aumento de desempenho na ordem de 100 vezes. Para a implementação da versão em GPU, foi utilizada a biblioteca CUDA 2D FFT library, fornecida pela fabricante da GPU.

Knittel (2010) descreve um sistema experimental para computação paralela e distribuída para algoritmos numéricos. O autor propõe uma nova arquitetura onde as computações são feitas em GPUs e as comunicações são feitas através de FPGAs e conexões especializadas. Como caso de estudo, é utilizado um problema que envolve a solução de equação de Poisson em 3 dimensões. Esse trabalho foca na arquitetura

de um sistema de alto desempenho e na distribuição de processamento entre os nodos, com ênfase em evitar gargalos de comunicação. A arquitetura é formada por 4 computadores pessoais (PC) com 4 GPUs cada. Cada PC é equipado com 2 placas com FPGAs (*FPGA-board*) que se conectam diretamente com os FPGAs dos outros PCs através de uma rede desenvolvida pelo grupo. É usado o método de gradiente conjugado para resolver a equação de Poisson. Esse trabalho apresenta uma solução específica e a análise é focada na arquitetura do sistema proposto pelos autores.

Um trabalho que possui bastante afinidade com o que propomos em nosso trabalho é a tese de doutorado de He (2007). Esse autor avalia soluções numéricas para equações diferenciais são em computadores que incluem FPGAs em suas arquiteturas. Além de uma nova arquitetura para esses sistemas, esse trabalho apresenta soluções para acelerar a computação de equações diferenciais parciais. Para apresentar a eficiência das soluções propostas, são utilizados o método de diferenças finitas e o método de elementos finitos aplicados na solução de equações baseadas em ondas lineares de sísmica. As soluções em hardware foram simuladas e comparadas com versões puramente em software. As comparações mostraram vantagens para as versões em hardware.

Outro trabalho que usa FPGAs na solução de aplicações críticas é (TSOI; LUK, 2010). Além de FPGAs, GPUs são integradas em um sistema de alto desempenho para aumentar a capacidade computacional do *cluster*. Para avaliar o sistema híbrido proposto, os autores implementaram uma simulação de N-corpos utilizando colaborativamente todos os recursos disponíveis, sendo utilizada representação em ponto flutuante simples IEEE 754 para todos os dispositivos. Uma implementação sequencial foi desenvolvida para servir como referência nos testes. As outras versões incluem uma versão *multi-thread*, uma versão para GPU e uma versão para FPGA. Essa última, possui um *pipeline* de 132 ciclos e pode ser executada em até 400MHz no FPGA. O desempenho medido para a execução de 10 *pipelines* paralelos em um FPGA apresentou aceleração de 17,7 vezes em relação a execução em CPU. Quando FPGA, GPU e CPU colaboram nessa execução, a aceleração chega a 22,7 vezes em relação a um único processador.

Como um algoritmo computacionalmente intensivo, a FFT também é adequada para aceleração em implementações baseadas em FPGA. Arquiteturas de hardware de FFT têm sido extensivamente investigadas em uma série de trabalhos de pesquisa (PALMER, 2005; BANERJEE et al., 2005; HEMMERT; UNDERWOOD, 2005; HE; GUO, 2008; VITE-FRIAS et al., 2005; GONZALEZ-CONCEJERO et al., 2008) e estão dis-

poníveis como módulos em diferentes fornecedores (Dillon Engineering, Inc., ; 4DSP Inc., ; Xilinx Inc., ; Altera Corporation,). Essas arquiteturas diferem em seus objetivos e aplicações alvo. Algumas são focadas em baixo uso de recursos e baixo consumo energético, enquanto outras estão interessadas em alta vazão e alto desempenho. Elas também diferem em termos do algoritmo FFT (radix-2 ou outras bases) e as estratégias de implementação de hardware.

Para alcançar um alto desempenho e vazão, arquiteturas baseadas em FPGA geralmente empregam a técnica de *pipeline* para operar em fluxo de dados contínuos (PALMER, 2005). Essa técnica apresenta grande desempenho, mas é restritiva quanto ao tamanho do vetor a ser processado, devido ao limite de recursos do hardware. Em aplicações híbridas para sistemas reconfiguráveis, os FPGAs geralmente precisam ler um vetor com tamanho fixo para processar a FFT. Neste caso, o fluxo contínuo de dados em tempo real não é um requisito para a FFT em FPGAs.

Enquanto trabalhos de FFT centrados em FPGAs possuem mais de uma década, para nosso conhecimento, não existe uma investigação completa para modelos híbridos, onde o ambiente em que o FPGA é integrado difere de aplicações embarcadas. Apesar de existirem semelhanças, em ambientes híbridos os dados a serem processados geralmente estão em memória do programa e devem ser transferidos para o FPGA. Além da necessidade de arcar com os custos de comunicação, a hierarquia de memória costuma diferir em níveis e quantidade disponível.

3 PROJETO DE FAST POISSON SOLVER HÍBRIDO

Nosso projeto de *Fast Poison Solver* Híbrido tem como foco principal aproveitar o poder computacional combinado de CPU e FPGA. O objetivo é permitir que aplicações que necessitem da solução de equações de Poisson, tradicionalmente codificadas em software, possam aproveitar dessa solução híbrida sem que haja a necessidade de alterações profundas no código.

O projeto visa utilizar CPU e FPGA em um fluxo único de processamento, em que o FPGA fique responsável pelo processamento da região computacionalmente mais intensiva do problema. Além disso, nosso projeto lida com os custos de transferência de dados entre os dispositivos, contabilizando esses custos no tempo total necessário da computação, uma vez que, para a aplicação principal, o recurso utilizado para realizar o processamento deve ser transparente.

De maneira geral, pode-se sistematizar um *fast Poisson solver*, que emprega a transformada de Fourier na solução de equações de Poisson. Após a discretização da equação de Poisson usando algum método numérico (como diferenças finitas, por exemplo), um sistema algébrico linear é obtido:

$$Az = b, \tag{3.1}$$

onde $z = [z_1, z_2, \ldots, z_n]$, $b = [b_1, b_2, \ldots, b_n]$ e a matriz A em 1D é "quasi"-tridiagonal. Para modelos 2D ou 3D pode-se aplicar um algoritmo bloco-FFT, ou se houver condições de contorno periódica em todas as direções pode-se usar FFT 2D (ou 3D), ou uma cascata de FFT. As matrizes de Fourier nestes casos são matrizes bloco. Através da fatorização da matriz de coeficientes, podemos obter uma solução do sistema dada por

$$z = F_n^* \Lambda^{-1} F_n b \tag{3.2}$$

onde F_n^* e F_n são matrizes de Fourier e Λ^{-1} é uma matriz diagonal. As multiplicações pelas matrizes de Fourier são a transformada discreta de Fourier, que podem ser realizadas através de seu algoritmo rápido FFT.

As etapas e o fluxo de processamento dessa solução têm a seguinte forma:

(a) cálculo da FFT sobre a entrada b;

(b) multiplicação do resultado de (a) pela matriz diagonal Λ^{-1}; e

(c) cálculo da FFT inversa do resultado de (b), chegando à solução da equação de Poisson.

Destaca-se que o processamento da operação (b) tem complexidade $O(N)$, pois somente os elementos da diagonal principal da matriz são não nulos. Os outros dois passos têm complexidade computacional $O(N \log N)$ devido ao cálculo da FFT. Deste modo, observa-se que nessa solução da equação de Poisson, a transformada de Fourier é o núcleo crítico do processamento, sendo, então, a operação escolhida para ser processada em FPGA através de um coprocessador especializado. A Figura 3.1 ilustra o mapeamento das tarefas necessárias para processar e o fluxo de informações utilizados no *Fast Poisson Solver* Híbrido proposto neste trabalho.

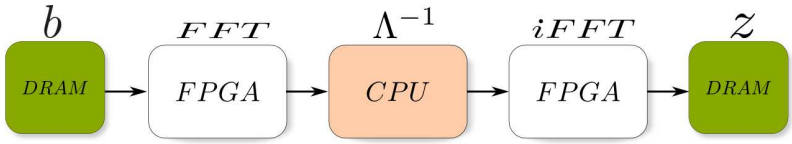


Figura 3.1 - Mapeamento de tarefas e fluxo entre FPGA e CPU para solução de *fast Poisson solver*

O vetor de entrada b, que inicialmente está na memória DRAM, é transferido para o FPGA para que seja computada a FFT. O resultado retorna à CPU, onde será realizada a multiplicação pela matriz diagonal Λ^{-1}. O resultado dessa computação é transferido para o FPGA para que a solução final z possa ser obtida através da computação da FFT inversa. Após o processamento dessa operação no FPGA, o resultado é enviado novamente para a memória RAM conectada à CPU, para que a aplicação principal siga normalmente seu fluxo de computação.

O pseudocódigo apresentado na Listagem 3.1 ilustra um procedimento em software que realiza o processamento da equação de Poisson conforme o fluxo citado anteriormente. Os procedimentos $fpga\_fft$ e $fpga\_ifft$ são os responsáveis por gerenciar as atividades necessárias para que o FPGA realize, respectivamente, a computação das transformadas rápidas direta e inversa de Fourier.

Entre as linhas 3 e 5 na Listagem 3.1, encontra-se a tarefa destinada à CPU. Por se tratar de uma operação simples, somente as tarefas destinadas ao FPGA serão tratadas em detalhes no restante deste trabalho. Como aplicações para sistemas híbrido reconfiguráveis são dependentes da arquitetura dos equipamentos, especialmente quando se deseja maximizar o desempenho, o equipamento alvo do nosso

Listing 3.1 - Pseudocódigo do Fast Poisson Solver Híbrido

```
function fpsh(b) {
    fb = fpga_fft(b)
    for(i=0; i<N; i++) {
        lfb = lambda[i]*fb[i]
    }
    z = fpga_ifft(lfb)
    return z
}
```

projeto de *Fast Poisson Solver* Híbrido é apresentado na subseção seguinte. Na sequência, apresenta-se o projeto de coprocessador FFT para FPGA, assim como os detalhes de sua implementação.

3.1 Equipamento alvo: Cray XD1

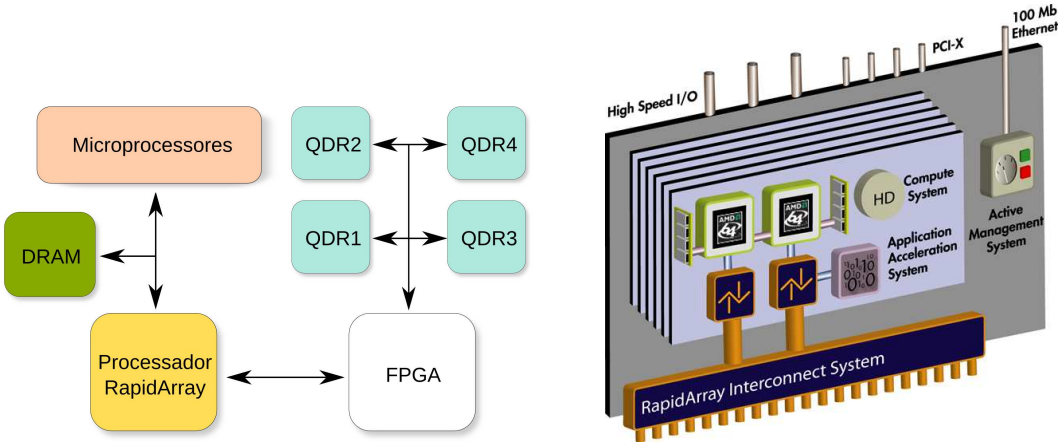
O equipamento alvo de nosso trabalho é o sistema híbrido reconfigurável de alto desempenho Cray XD1. Esse equipamento, disponível no Laboratório Associado de Computação e Matemática Aplicada do Instituto Nacional de Pesquisas Espacias (LAC/INPE), trouxe, no momento de seu lançamento, algumas inovações tecnológicas. Entre elas está a rede de interconexão de alto desempenho RapidArray, otimizações no sistema Linux e a inclusão de FPGAs em seu chassis. Na figura 3.2 é possível ver uma foto de um equipamento Cray XD1.

Figura 3.2 - Foto de um equipamento Cray XD1.
Fonte: Cray Inc. (2005a)

Cada sistema Cray XD1 é composto por seis nós (*blades*), cada um contendo dois processadores de propósito geral AMD Opteron 64 bits e um FPGA Xilinx Virtex II Pro.

A arquitetura de um *blade* do Cray XD1 pode ser vista na Figura 3.3(a), enquanto que a Figura 3.3(b) apresenta a visão lógica de um nó desse sistema. É possível

observar que o dispositivo reconfigurável tem acesso direto a quatro bancos de memória QDR II SRAM, os quais possuem 4MB cada. O processador RapidArray é o elemento responsável por gerenciar a comunicação entre FPGA e CPU. Esse recurso permite que os processadores enviem dados para o FPGA e que o FPGA leia dados da DRAM. No desenvolvimento de aplicações híbridas, existem duas questões-chave que devem ser observadas: o uso eficiente dos diferentes níveis de memória disponíveis no sistema (CARDOSO et al., 2010) e a transferência de dados entre os dispositivos (GOMES et al., 2009a).



(a) Arquitetura (b) Visão Lógica.

Fonte: Adaptado de Cray Inc. (2005a)

Figura 3.3 - Arquitetura e Visão lógica de um *blade* do Cray XD1

3.1.1 Hierarquia de Memória

Na Figura 3.3(a) pode ser visto que o FPGA de um *blade* do XD1 tem acesso a pelo menos dois tipos diferentes de memória. Além dos blocos de QDR e da memória DRAM que aparecem nesse diagrama, o FPGA tem disponíveis blocos de memória em sua arquitetura, chamados de *Block RAMs*.

Cada recurso disponível tem forma de acesso e quantidades distintas. A DRAM é o mais alto nível, com a maior quantidade de memória disponível e com latência de leitura não constante. Esta memória pode ser acessada usando uma interface de comunicação disponibilizada pela Cray, o *RapidArray Transport Core*. Em um nível mais baixo, estão os bancos de memória QDR II SRAM com 4MB cada. A latência de acesso a esses bancos é de 8 ciclos para a leitura, e o acesso é feito utilizando

a interface *QDR II SRAM Core*, também disponibilizada pelo fabricante. Essa interface permite gravações e leituras completamente independentes nesses bancos de memória. Além desses, a família de FPGAs Virtex II Pro possuem bancos de memória internos que podem ser utilizados diretamente pelas aplicações com gravações e leitura em um único ciclo de relógio.

Nessa hierarquia de memória, a DRAM normalmente é utilizada para compartilhar dados da aplicação em software com o FPGA. Esses dados são carregados para a QDR II SRAM para serem acessados durante a execução da aplicação em hardware. A memória interna do FPGA, disponível em menor quantidade, frequentemente é utilizada para registradores e cache de dados. Independente de uma forma geral, cada aplicação necessita de recursos e formas de acesso diferentes que devem ser otimizadas conforme as necessidades, sendo um importante fator para diminuir tempos de execução.

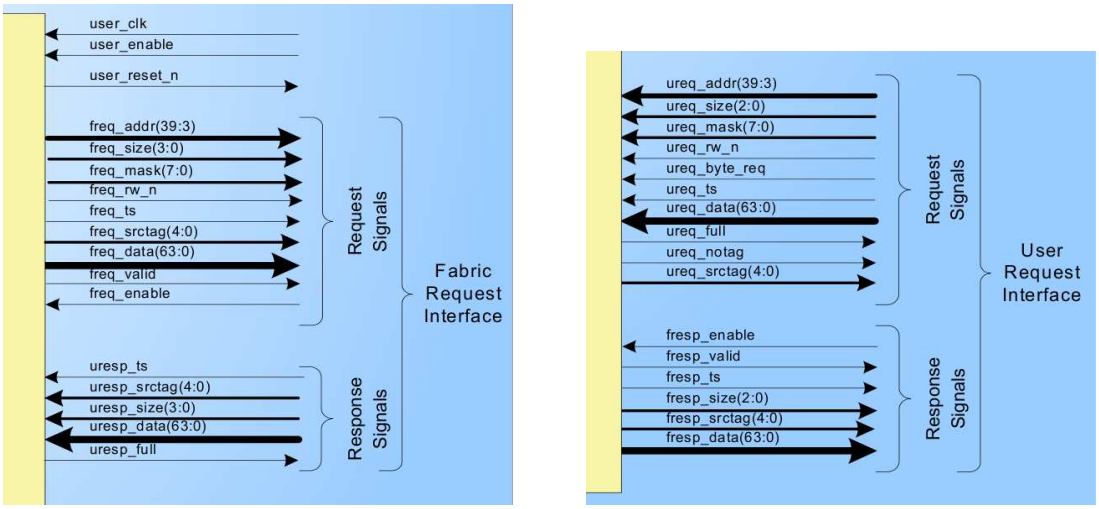
3.1.2 Transferência de Dados

A transferência de dados entre CPU e FPGA é um elemento chave que deve ser levado em consideração durante o desenvolvimento de aplicações nesta tecnologia.

Para a comunicação entre dispositivos em um *blade*, a Cray disponibiliza a API *RapidArray Transport Core* (RTCore), que é um componente que deve ser usado com a descrição do algoritmo em VHDL. Esta entidade é composta por duas interfaces denominadas *Fabric Request* e *User Request* (Cray Inc., 2005d). As Figuras 3.4(a) e 3.4(b) apresentam, respectivamente, os sinais dessas interfaces.

Esses dois conjuntos de sinais permitem duas formas de comunicação, onde ou a CPU ou o FPGA é o responsável pela iniciativa de começar a transferência de dados.

A interface *Fabric Request*, que realiza a comunicação usando uma abordagem *push*, permite que o programa executado nos processadores envie e requisite dados para o FPGA. Esta abordagem mantém a CPU ocupada durante a transferência dos dados. Para permitir esse tipo de comunicação, a Cray disponibiliza a biblioteca em linguagem C *enlib*, a qual abstrai ao programa o FPGA como um arquivo. Deste modo, a transferência de dados entre os processadores e o FPGA é realizada através de leituras e gravações pelo programa neste 'arquivo'. Com a realização de uma leitura ou escrita, o FPGA recebe, através do RTCore, uma requisição que deve ser tratada pela aplicação do FPGA. Em caso de leitura, deverá ser retornado um valor ao *RapidArray Transport Core* para que ocorra o retorno da função chamada



(a) Diagrama da interface Fabric Request (b) Diagrama da interface User Request

Figura 3.4 - Interfaces do RapidArray Transporte Core.
Fonte: Adaptado de Cray Inc. (2005d)

pelo programa em linguagem de alto nível. Somente é permitida a manipulação de um *quadword* (64 bits) por requisição utilizando a interface *Fabric Request* (Cray Inc., 2005d). Na Figura 3.4(a) é possível ver os sinais que são ativados quando uma requisição é feita pelo programa –*Request Signals* – e os sinais que são usados pela aplicação em VHDL para responder essas requisições –*Response Signals*.

A abordagem que utiliza a interface *User Request* é conhecida como *pull* e, diferentemente da *Fabric Request*, mantém os processadores livres durante a transferência de dados entre o programa em C e o FPGA. Para isso, esta interface permite que o FPGA realize leituras e escritas em um espaço de memória DRAM compartilhado pelo programa. O endereço para a região de memória, que é compartilhada utilizando a *einlib*, precisa ser enviado ao FPGA através do bloco *Fabric Request*. Estando disponível o endereço, a aplicação descrita para o FPGA é capaz de fazer até 32 requisições sequenciais ao RTCore. Cada requisição pode solicitar até 8 posições contíguas da memória do programa.

Os retornos das solicitações ao RTCore não são necessariamente na ordem em que foram realizadas. O RTCore garante somente a ordem das 8 posições contíguas de cada requisição (Cray Inc., 2005d). O sistema descrito para o FPGA deve ordenar os dados através do auxílio de *tags* disponibilizadas durante a requisição e o retorno. Outra solução é aguardar o retorno de uma solicitação antes de realizar outra.

Os sinais da interface *User Request* podem ser vistos na Figura 3.4(b). Através dos sinais *Request Signals* o FPGA é capaz de requisitar posições de memória, sendo atendidas através dos sinais *Response Signals*.

3.1.3 Desenvolvimento

A Cray disponibiliza um *template* com a estrutura de um projeto para o desenvolvimento de aplicações híbridas para o Cray XD1. A Figura 3.5 apresenta um esquema que ilustra os componentes desse projeto. O template possui um componente (*prog_clk_gen*) responsável pela geração dos sinais de controle do FPGA (*clock*, *reset_n*, etc), uma instância do componente *rt_core*, o qual permite a comunicação com o programa executado em CPU, e uma instância do *qdr2_core*, que é a interface para acessar os quatro blocos de memória QDR II SRAM. A aplicação a ser desenvolvida é representada pelo componente *user_app*. O componentes estão previamente configurados no *template*, sendo necessário somente a implementação do componente *user_app*.

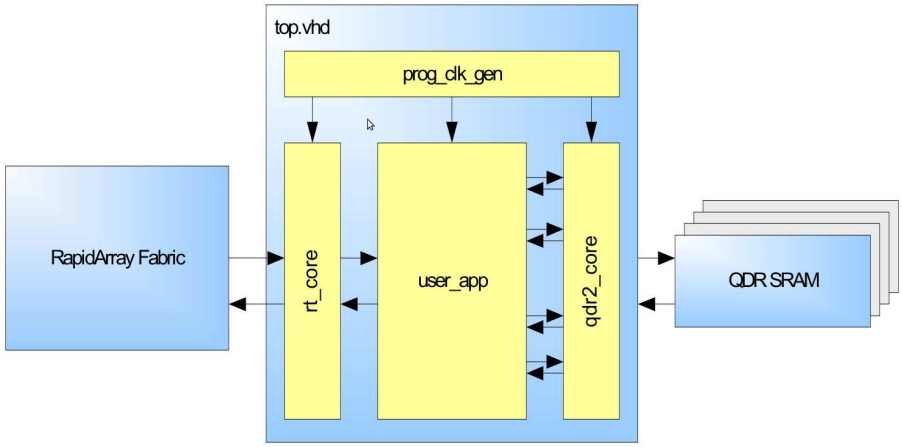


Figura 3.5 - Estrutura do *template* de desenvolvimento para aplicações para FPGA do Cray XD1.
Fonte: Cray Inc. (2005d)

3.2 Coprocessador FFT

Esta seção descreve o projeto do coprocessador responsável pela computação da FFT em FPGA. Esse coprocessador é integrado à solução de *Fast Poisson Solver* Híbrido apresentado no início desse capítulo. Nosso projeto de coprocessador FFT em FPGA visa maximizar as oportunidades de paralelismo, reduzindo os custos de

comunicação envolvidos no transporte de dados entre FPGA e CPU e aproveitar os recursos disponíveis em um *blade* do Cray XD1.

Para que sejam minimizados os problemas quanto a precisão numérica, esse projeto utiliza para o processamento das operações borboleta da FFT a representação numérica e as operações em ponto flutuante definidas no padrão IEEE-754. Esse padrão é o mesmo utilizado em CPUs modernas. Desta forma, as aplicações em software não serão afetadas por potenciais diferenças numéricas causadas pela execução de parte da aplicação em um elemento computacional diferente da CPU. Essa escolha foi feita para garantir os objetivos de transparência do nosso projeto de *Fast Poisson Solver* Híbrido.

Para permitir que a CPU fique livre para a realização de outras computações, o FPGA é o responsável pela transferência dos dados. A CPU acessa o FPGA somente para enviar os endereços das regiões de memória DRAM compartilhadas e para verificar o fim da computação.

Para permitir uma melhor análise das soluções utilizadas nesse projeto, os procedimentos necessários para a computação da FFT em FPGA foram modularizados em unidades que são apresentadas no diagrama da Figura 3.6. Além dessas unidades, esse diagrama apresenta as conexões de sinais entre as unidades e o processador RapidArray. Duas unidades são responsáveis pela comunicação com esse processador. A Unidade de Comunicação usa a interface *User Request* para transferir os dados a serem computados e para enviar os dados processados para a memória do programa. A Unidade de Controle acessa os sinais da interface *Fabric Request* para receber endereços de memória e sinais de controle do software e para informar o estado do processamento.

Como pode ser observado no diagrama da Figura 3.6, a Unidade de Controle interage somente com a Unidade de Comunicação e o Controlador de Memória. As outras duas unidades, Constantes Trigonométricas e de Computação, respondem a estímulos colocados nos barramentos de dados e são responsáveis pelos seus estados de controle. Essa abordagem visa reduzir o uso de recursos do hardware para atividades de controle e evitar que a coordenação das atividades fique centralizada em uma única unidade. Ao dividir os procedimentos de controle entre as unidades, um maior número de operações pode ser realizado em paralelo (inclusive as de controle), maximizando o uso do hardware.

Existem diversas maneira de se estruturar as operações da Transformada Rápida de

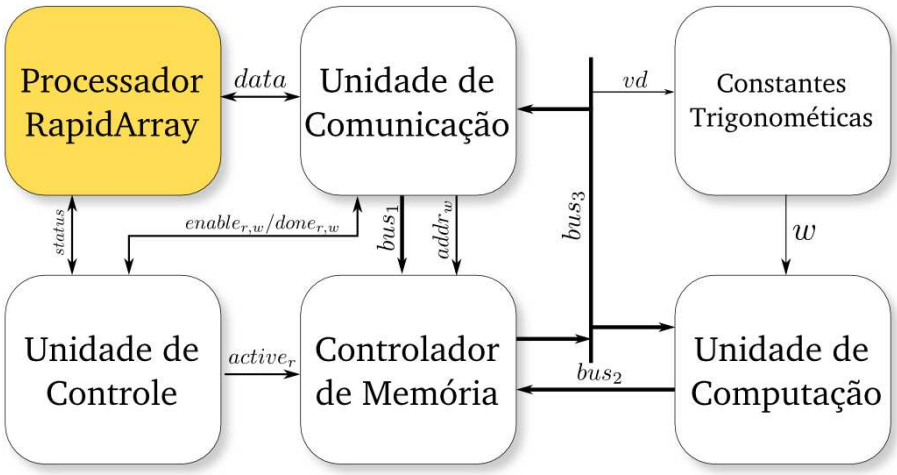


Figura 3.6 - Diagrama da arquitetura do coprocessador FFT para FPGA

Fourier em hardware. Algumas delas são (BANERJEE et al., 2005):

Arquitetura Iterativa: as operações de um passo completo da FFT são implementadas em hardware. Essa estrutura pode ser ilustrada como uma coluna de operações da Figura 2.6. A computação completa da FFT é realizada em $\log N$ etapas, sendo usada iterativamente essa estrutura de operações. Essa solução é possível somente para a operação de FFT em pequenos vetores, devido as limitações do hardware para a implementação de muitas unidades aritméticas e a necessidade elevada de dados disponíveis para manter todas as unidades funcionais ativas;

Arquitetura Pipeline: todas as operações necessárias para a computação de uma FFT são implementadas em hardware. Essa estrutura é utilizada na forma de um *pipeline*, onde o número de estágios é igual ao número de passos da FFT. Essa solução é uma das que apresenta melhor desempenho, mas possui maiores restrições devido ao uso de muitos recursos de hardware. Além disso, é preciso garantir que N posições de memória estejam disponíveis a cada ciclo de relógio para manter todos os estágios do *pipeline* preenchidos. O uso dessa arquitetura torna-se impraticável quando se opta pelo uso de operações complexas em ponto flutuante e pelo suporte a vetores com grande quantidade de pontos; e

Operações Individuais: uma única unidade funcional é implementada em hardware. As operações de borboleta são calculadas individualmente, onde em um fluxo de execução, como o ilustrado na Figura 2.6, as operações ini-

ciam com a operação borboleta do canto superior esquerdo, seguindo para baixo e para a direita até o final da computação da FFT. Essa arquitetura apresenta grande flexibilidade, do ponto de vista do uso de hardware e tamanho do vetor de entrada.

Além dessas, podem ser utilizadas arquiteturas mistas que tentam aproveitar vantagens de duas ou mais técnicas, sem arcar com um custo muito alto. No nosso projeto, utilizamos uma abordagem de Operações Individuais com uma unidade de processamento sendo responsável por calcular todas as operações de borboleta. Para aumentar o volume de dados processados, é utilizada a técnica de *pipeline*, onde um novo conjunto de dados é aceito a cada ciclo de relógio. A escolha dessa arquitetura permite que o projeto possa ser acomodado nos recursos disponíveis no hardware e que seja flexível quanto ao tamanho do vetor de entrada. Os detalhes da unidade responsável por realizar essas operações são apresentados na subseção 3.2.4.

Apesar de grande parte das funções do nosso coprocessador FFT ocorrerem em paralelo, podemos destacar uma sequência de atividades que compõem o processamento da FFT no FPGA:

Carregamento dos dados: os dados, que inicialmente estão na memória DRAM, são trazidos para os bancos de memória QDR do FPGA. A Unidade de Comunicação e o Controlador de Memória são os módulos envolvidos nesse processo;

Processamento: operações de borboleta são realizadas sobre os dados até que estejam completados os passos necessários para o processamento da FFT. As unidades de Controle, Computação, Constantes Trigonométrica e o Controlador de Memória atuam nessa tarefa; e

Envio dos dados: o resultado da computação é enviado para a área de dados da CPU. O Controlador de Memória e a Unidade de Comunicação coordenam esse procedimento.

As subseções a seguir descrevem com detalhes os barramentos, cada uma das unidades e os sinais entre cada entidade.

3.2.1 Barramentos de Dados

Os dados trafegam entre as unidades funcionais do nosso projeto através de três barramentos principais. Dois desses barramentos – bus_1 e bus_2 – são responsáveis,

respectivamente, por levar dados da Unidade de Comunicação e da Unidade de Computação para o Controlador de Memória. O terceiro barramento – bus_3 – é utilizado pelo Controlador de Memória para enviar os dados para a Unidade de Computação ou, ao final do processamento, para a Unidade de Comunicação.

Esses três barramentos compartilham uma mesma estrutura de sinais que é ilustrada pelo diagrama da Figura 3.7. Cada barramento apresenta três conjuntos de sinais, que são chamados de $data_1$, $data_2$ e vd. Os conjuntos $data_1$ e $data_2$ possuem o mesmo formato, o qual representa um número complexo. Cada um desses dados é subdivido em dois valores que representam a parte imaginária (sinal *imag*) e a parte real (sinal *real*) do número complexo. Cada um desses sinais possui 32 bits, que correspondem à representação em ponto flutuante simples no padrão IEEE-754.

O sinal vd (de *valid data*) é um sinal de um único bit que indica às unidades que escutam os barramentos que os dados presentes em $data_1$ e $data_2$ são válidos. Esse sinal atua diretamente no controle das unidades que não possuem conexões com a Unidade de Controle, como na conexão de vd do bus_3 com a unidade de Constantes Trigonométricas, ilustrada na Figura 3.6.

A estrutura de barramento utilizada nesse projeto possui largura de 65 bits e a declaração dos tipos de sinais em VHDL é apresentada no código-fonte da Listagem 3.2. Os demais sinais que aparecem na Figura 3.6 são utilizados para o controle, endereçamento e sincronização das atividades do coprocessador de FFT. Detalhes sobre cada um desses sinais são apresentados nas subseções seguintes.

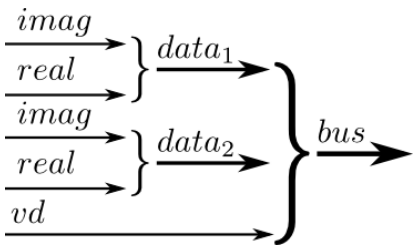


Figura 3.7 - Agrupamento dos sinais dos barramentos de dados

Listing 3.2 - Estrutura dos barramentos de dados

```
type t_complex is                                    1
 record                                              2
  creal : std_logic_vector(31 downto 0);             3
  cimag : std_logic_vector(31 downto 0);             4
 end record t_complex;                               5
                                                     6
type t_bus is                                        7
 record                                              8
  data1 : t_complex;                                 9
  data2 : t_complex;                                 10
  vd    : std_logic;                                 11
 end record;                                         12
```

3.2.2 Unidade de Controle

A Unidade de Controle é a entidade responsável por coordenar o funcionamento das demais unidades funcionais no nosso projeto. Essa unidade atua diretamente com a

Unidade de Comunicação. Através do sinal $enable_r$, essa unidade habilita a leitura dos dados da memória do programa. A Unidade de Comunicação indica o final dessa tarefa através do sinal $done_r$. O sinal $enable_w$ solicita à Unidade de Comunicação o envio dos dados processados para a memória DRAM. A sinalização do término dessa tarefa é feita pelo sinal $done_w$.

A atuação da Unidade de Controle com o Controlador de Memória ocorre através do sinal $active_r$. Esse sinal, de um bit, habilita a leitura de dados que serão colocados no barramento bus_3 pelo Controlador de Memória. O endereço de leitura é gerado pelo próprio Controlador de Memória, pois, como estamos trabalhando com um projeto específico para a execução de FFT, conhecemos antecipadamente a ordem de acesso aos dados.

A Unidade de Constantes Trigonométricas e a Unidade de Computação não recebe sinais da Unidade de Controle, pois são ativadas através dos sinais que são enviados para o barramento bus_3. Essa abordagem, como citada anteriormente, contribui para a redução da quantidade de conexões entre as unidades e do uso de recursos do hardware.

Além de coordenar outras unidades funcionais, a Unidade de Controle gerencia a comunicação do FPGA com a interface *Fabric Request* do *Rapid Array Transport Core*. Essa conexão é ilustrada pelo sinal *status* na Figura 3.6. Esse canal de comunicação é utilizado pela CPU para o envio dos endereços de memória compartilhada com o *blade* e para a leitura de métricas do hardware e do sinal de conclusão da computação.

Os sinais de controle da Unidade de Controle são gerados através de uma máquina de estados finitos (MEF). Essa MEF possui 5 estados e é ilustrada pelo diagrama da Figura 3.8.

Ao ligar o hardware, o estado inicial é *idle*. A MEF permanecerá neste estado até que os endereços de memória compartilhados pelo software sejam enviados para o FPGA. A chegada desses sinais é ilustrada pela condição $addrs$ e o estado seguinte é *downloading*. Essa mudança de estado gera a habilitação do sinal $enable_r$, que habilita a leitura de dados pela Unidade de Comunicação. O envio do sinal $done_r$ pela Unidade de Comunicação fará com que outra mudança de estado aconteça. A MEF segue para o estado *computing*, ativando o sinal $active_r$. Esse sinal, como explicado anteriormente, habilita o Controlador de Memória a realizar sucessivas leituras nos bancos de memória QDR, colocando os resultados no barramento bus_3.

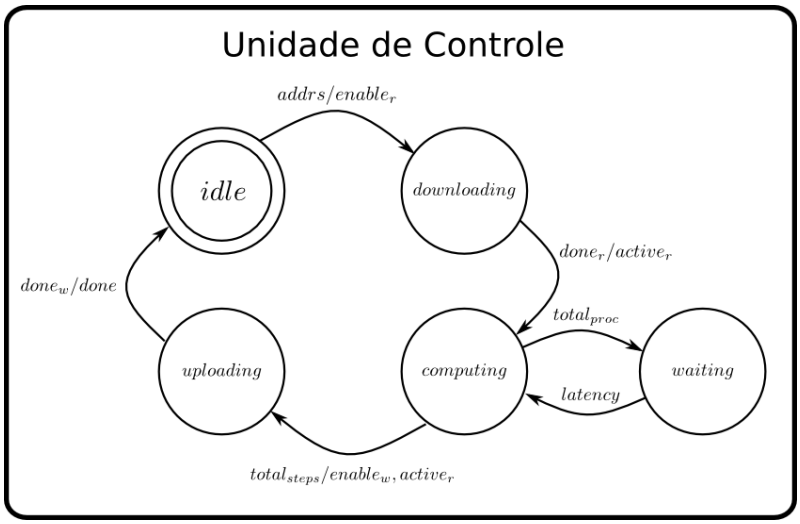


Figura 3.8 - Diagrama da Unidade de Controle

O estado *computing* representa a fase de processamento dos dados, onde são realizadas as operações de borboleta. Nesse estado, o sinal $total_{proc}$ contabiliza o total de dados solicitados ao Controlador de Memória. Quando as solicitações forem suficientes para o processamento de um passo da FFT ($total_{proc} == N/2$), a MEF da Unidade de Controle segue para o estado *waiting*. A MEF aguarda neste estado até que todos os dados tenham sido processados pela Unidade de Computação e enviados de volta para o Controlador de Memória. Esse tempo é fixo, pois a latência de leitura da memória QDR e o total de ciclos necessários para o processamento de uma borboleta são conhecidos.

A MEF da Unidade de Comunicação alterna entre os estados *computing* e *waiting* até que o total de passos da FFT tenha sido atingido ($total_{steps} == \log N$). Quando essa condição ocorre, o sinal $enable_w$ é habilitado e a MEF segue para o estado *uploading*. Assim que os sinais $enable_w$ e $active_r$ são ativados, o Controlador de Memória continua na leitura de dados da QDR e a Unidade de Comunicação inicia o envio dos dados processados de volta para a memória do *blade*. O final desse processo é comunicado pelo sinal $done_w$, que ativa a transição final da MEF. Nessa transição, o sinal *done* é ativado, indicando o final da computação da FFT em FPGA. Na sequência, a máquina de estados finitos volta ao seu estado inicial, esperando novamente por novos dados a serem processados.

3.2.3 Unidade de Comunicação

A Unidade de Comunicação é a interface entre o *RapidArray Transport Core* e o restante da lógica da aplicação. Essa unidade é responsável por transferir o conjunto de dados da memória do *blade* para os blocos de memória do FPGA a cada início de processamento. Ao final da computação, essa unidade gerencia o envio dos dados processados para a memória DRAM dos processadores, enviando um sinal à Unidade de Controle no final dessa transferência.

Além de facilitar a implementação da transferência de dados, a modularização do projeto fornece outras vantagens. No caso da Unidade de Comunicação, é possível a reutilização do restante do projeto em outros ambientes que possuam outro canal de comunicação, sendo necessário, somente, a alteração do módulo de comunicação.

Na Figura 3.6 é ilustrada a conexão da Unidade de Comunicação com o processador *RapidArray* através do sinal *data*. Os sinais utilizados nessa conexão são os apresentados na Figura 3.4(b) da seção 3.1.2.

Devido à necessidade de gerenciar o recebimento e envio de dados e pelo fato desses processos serem independentes e concorrentes, foram construídas 4 máquinas de estados finitos. O diagrama da Figura 3.9 ilustra o funcionamento de cada uma dessas máquinas de estados. Nessa figura, os sinais do lado esquerdo representam as conexões com o RTCore, enquanto que os sinais do lado direito com as demais unidades do coprocessador FFT.

A primeira máquina de estados (MEF_1) é responsável pela solicitação dos dados da memória do *blade*. A partir da habilitação pela Unidade de Controle do sinal $enable_r$, e se o processador RapidArray não estiver ocupado, uma nova requisição (sinal req) é feita a cada ciclo de relógio até que o total de dados a serem processados seja atingido. Cada requisição solicita 8 posições de memória contíguas de 64 bits. Apesar dos dados de uma requisição chegarem em ordem, as requisições podem chegar fora de ordem. Para gerenciar o recebimento dos dados, o RTCore fornece no momento da requisição um marcador (sinal tag_w) que funciona com um identificador da requisição. Para permitir a ordenação dos dados durante o recebimento, esse identificador é utilizado para endereçar um bloco de RAM interno do FPGA que armazena a posição de memória requisitada (sinal $addr$), conforme ilustra a Figura 3.9.

A máquina de estados MEF_2 é responsável pelo recebimento dos dados requisitados

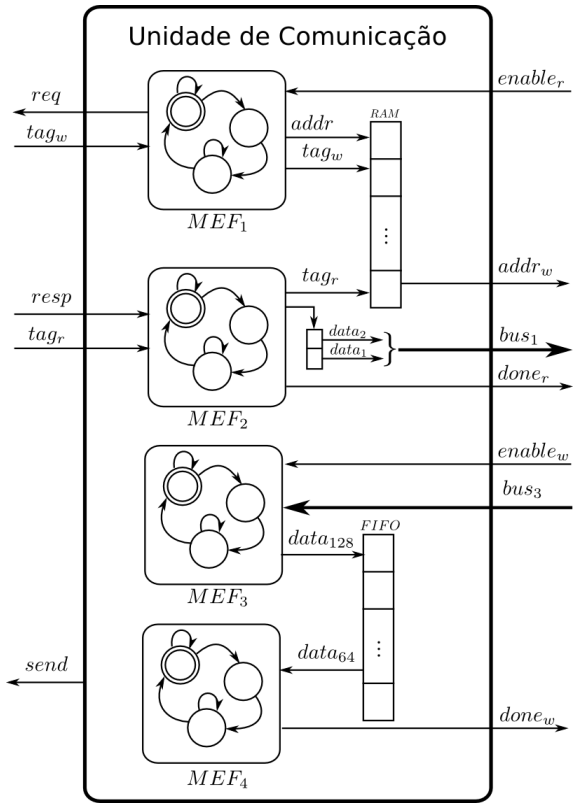


Figura 3.9 - Diagrama da Unidade de Comunicação

ao RapidArray pela MEF_1. Como os dados trafegam pelos barramentos aos pares, e a cada ciclo de relógio somente um número complexo é recebido através do RTCore, é utilizado um *buffer* com duas posições para agrupar os dados antes de serem colocados no barramento de saída bus_1.

Além de um *quadword*, a MEF_2 recebe um identificador (sinal tag_r) para ser usado na ordenação dos dados. Essa informação é usada para endereçar o bloco de memória que contém o endereço que se refere a requisição recebida. Esse endereço é lido e disponibilizado através do sinal $addr_w$ para o Controlador de Memória. Quando a quantidade de dados recebidos é igual ao total esperado, o sinal $done_r$ é habilitado, indicando à Unidade de Controle o final desta tarefa.

Para o envio dos dados dos blocos QDR do FPGA para a memória do *blade* são utilizadas as outras duas máquinas de estados finitos restantes – MEF_3 e MEF_4. A MEF_3, quando habilitada pela Unidade de Controle através do sinal $enable_w$, começa a tratar os dados que são colocados no barramento bus_3 pelo Controlador de Memória. Cada dado que chega é colocado pela MEF_3 em um bloco de memória do tipo Primeiro a Entrar, Primeiro a Sair (FIFO - *First In, First Out*). Como os

dados chegam aos pares e o envio ao processador RapidArray é limitado em 64 bits, o FIFO utilizado para esse propósito tem entrada de 128 bits e uma saída de 64 bits.

A última máquina de estados finitos da Unidade de Comunicação é a MEF_4. Esse sistema fica em laço infinito aguardando a chega de dados no FIFO. Ao identificar que existem dados nesta estrutura, a MEF_4 inicia o envio dos dados para a memória do *blade*. Até oito posições de memória são enviadas sequencialmente a cada requisição de envio. Após isso, é necessário aguardar um ciclo de relógio, conforme especifica o manual do *RapidArray Transporte Core* (Cray Inc., 2005d). Após o envio de todos os dados, a MEF_4 sinaliza o fim da operação à Unidade de Controle através do sinal $done_w$.

3.2.4 Unidade de Computação

A Unidade de Computação é a entidade responsável por realizar o processamento das operações de borboleta em nossa arquitetura. Essa entidade é composta basicamente de unidades aritméticas em ponto flutuante e de alguns registradores para a implementação da lógica de controle. Essa unidade implementa as operações necessárias para o processamento de uma borboleta usando a técnica de *pipeline*. Essa técnica permite que a cada ciclo de relógio um novo conjunto de dados possa ser recebido para processamento. Os resultados serão fornecidos nessa mesma frequência, após o preenchimento de todos os estágios do *pipeline*.

A Figura 3.10 apresenta um diagrama que ilustra o funcionamento interno da Unidade de Computação. Observa-se que o sinal que chega do barramento bus_3 é dividido em $data_2$ e $data_1$. Esse sinais são ainda subdivididos em suas componentes reais e imaginárias para a realização da computação. Usando a notação dos sinais apresentados nesse diagrama, a operação de borboleta é formulada como

$$data_1^{out} = data_1^{in} + data_2^{in} * w; \tag{3.3}$$

$$data_2^{out} = data_1^{in} - data_2^{in} * w; \tag{3.4}$$

onde todos os operandos são complexos, sendo necessários, então, 4 multiplicações reais, 3 somas reais e 3 subtrações reais. Para maximizar o paralelismo das operações no FPGA e permitir a utilização de um fluxo contínuo de dados, foi utilizado um módulo de operação em ponto flutuante para cada uma dessas operações. Esses módulos funcionam internamente em *pipeline*, recebendo dados para processamento a cada ciclo de relógio, sem a necessidade de sinais de controle. A latência para os multiplicadores reais é de 8 ciclos, enquanto que para somadores ou subtratores é

de 13 ciclos.

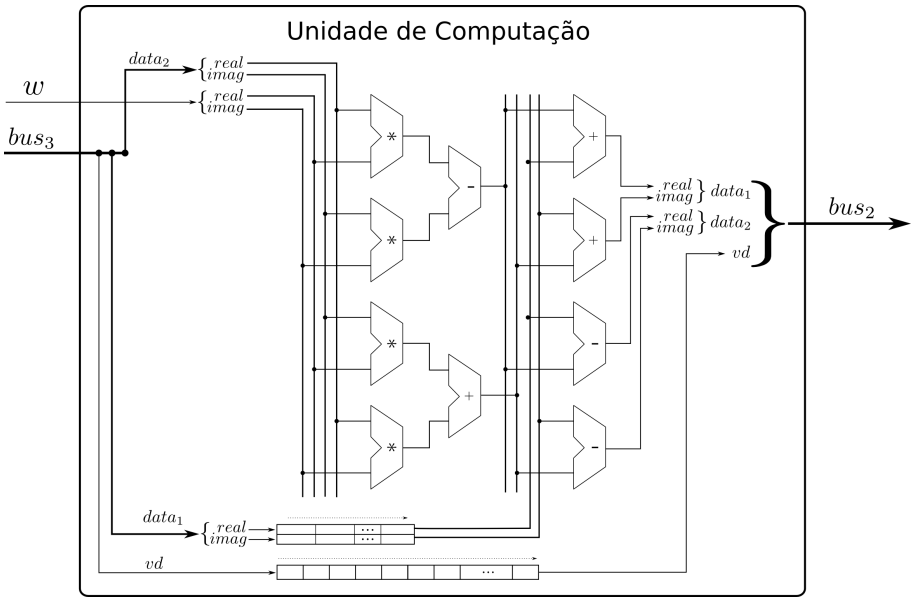


Figura 3.10 - Diagrama da Unidade de Computação

A primeira operação realizada pela Unidade de Computação é a multiplicação complexa entre $dado_2$ e w. As 4 multiplicações reais são realizadas em paralelo, seguidas por uma soma e uma subtração reais também paralelas. Essas operações necessitam de 21 ciclos para completar a multiplicação complexa, 8 da multiplicação e 13 da soma e subtração.

Enquanto essas operações são realizadas, o sinal $data_1$ segue em um registrador de deslocamento com 21 posições. Esse registrador é utilizado para retardar a entrada do $data_1$ no processamento, que só acontecerá após a multiplicação complexa.

A segunda etapa da operação de borboleta, que compreende uma soma e uma subtração complexas, é realizada por 2 somadores e 2 subtratores reais. Essas operações são realizadas em paralelo e, por isso, necessitam de 13 ciclos para serem completadas. As quatro unidades aritméticas alinhadas à direta da Figura 3.10 ilustram essas operações. O resultado dessa computação segue direto para os sinais de saída ligados ao barramento bus_2. O sinal vd, que indica dados válidos no barramento, é gerado a partir do próprio sinal vd de entrada. Esse sinal percorre um registrador de deslocamento que tem o mesmo número de bits que o total de estágios do *pipeline*, 34. Ao final do processamento, o bit que indicou dado válido na início do processamento estará na última posição do registrador e indicará que o resultado colocado

no barramento de saída tornou-se válido.

Essa abordagem para o controle da Unidade de Computação permite que múltiplas operações de borboleta estejam sendo processadas paralelamente, defasadas em um ciclo de relógio. No caso apresentado, até 34 operações podem estar sendo processadas em algum passo dessa computação. A utilização do registrador de deslocamento para carregar o sinal *vd* facilita a tarefa de indicar dados válidos na saída através de uma solução simples e que exige poucos recursos do hardware. Com a abordagem usada nesse projeto, a Unidade de Computação estará, a partir do momento que o dispositivo for ligado, computando operações de borboleta. Os resultados só serão considerados pelas demais unidades quanto o bit *vd* estiver habilitado.

Apesar da Unidade de Computação permitir que dados cheguem a cada ciclo de relógio, é necessário que as demais estruturas do coprocessador FFT forneçam dados nesta frequência para o completo aproveitamento desse recurso. Isto é um desafio, pois cada operação de borboleta requer três operandos de 64 bits (dois pontos complexos e uma constante trigonométrica complexa) e gera dois novos números complexos que devem ser armazenados nesta mesma frequência. Para lidar com estas questões, foram projetados a Unidade de Constantes Trigonométricas e o Controlador de Memória.

3.2.5 Unidade de Contantes Trigonométricas

A Unidade de Constantes Trigonométricas é responsável por fornecer as constantes trigonométricas utilizadas durante o processamento das operações de borboleta. Nosso objetivo é entregar uma constante a cada ciclo de relógio e reduzir ao máximo o uso de memória interna do FPGA. Para isso, realizou-se uma avaliação das constantes usadas no processamento da FFT. Na Figura 2.6 é possível ver um exemplo de processamento de uma FFT com 8 pontos, onde $\omega_N = e^{2\pi i/N}$.

O valor de cada constante trigonométrica usada nas operações de borboleta depende de qual passo (coluna) está sendo executado e de qual operação (linha) é executada da FFT. Observa-se que, apesar de variar os valores para cada operação, existem valores de constantes que se repetem durante o processamento. No primeiro passo é utilizada somente uma constante para todas as operações, enquanto que no segundo são usadas 2 constantes diferentes. No terceiro, 4 constantes são necessárias para a computação de todas as borboletas. Observa-se que a cada passo são necessárias 2^{k-1} constantes, onde k é o número do passo, que varia de $1 \ldots \log N$. Desta forma, o número de constantes necessárias para computar uma FFT de N pontos pode ser

obtido por

$$\sum_{k=1}^{\log N} 2^{k-1} = N - 1. \tag{3.5}$$

Desta forma, e com o intuito de fornecer uma constante trigonométrica por ciclo de relógio à Unidade de Computação, foi projetada uma entidade que conta com um banco de memória interna do FPGA com $N-1$ posições. Uma máquina de estados foi implementada para gerenciar a leitura e entrega dos dados e um FIFO foi conectado à saída para sincronizar a entrega. A Figura 3.11 apresenta um diagrama que ilustra o funcionamento da Unidade de Constantes Trigonométricas.

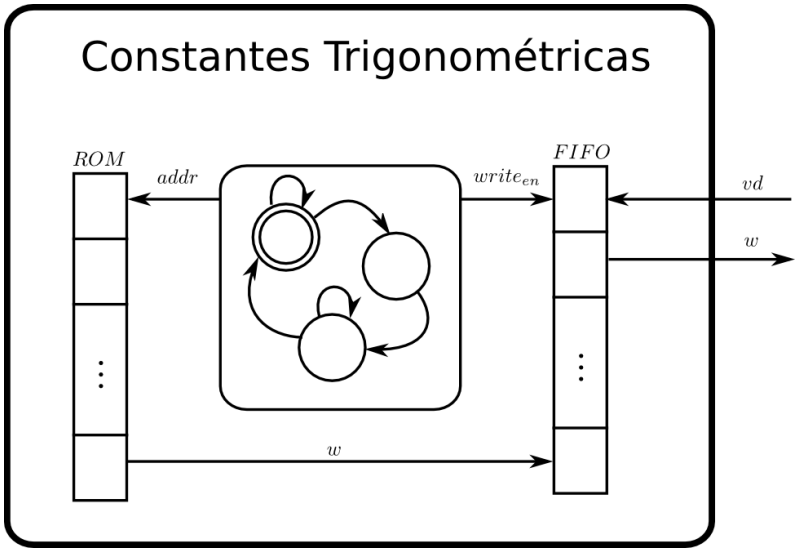


Figura 3.11 - Diagrama da Unidade de Constantes Trigonométricas

A partir do momento que o FPGA é ligado, a MEF dessa unidade inicia a leitura dos dados da ROM do FPGA. Um contador (sinal *addr*) que inicia em zero é utilizado para endereçar a memória com as constantes trigonométricas. O sinal $write_{en}$ é utilizado para habilitar a gravação dos dados no FIFO. Para as primeiras $N/2$ constantes gravadas nessa estrutura, esse contador permanece em seu valor inicial. No início do segundo passo, o contador é incrementado e esse endereço é utilizado até que sejam gravadas $N/4$ constantes trigonométricas no FIFO. Quando esse valor é atingido, incrementa-se novamente o contador e as leituras são feitas até o final desse passo. Esse algoritmo é repetido para todos os passos da FFT, sendo que no passo final da FFT o contador é incrementado a cada nova leitura da ROM.

Por usar a memória interna do FPGA, as leituras são feitas a cada ciclo de relógio até que o FIFO esteja cheio. O sinal $full$ indica esse estado para a MEF, que só volta a ler novos dados da ROM quando que esse sinal é desabilitado.

O FIFO utilizado nessa unidade mantém sempre o próximo valor disponível em sua saída. Esse sinal é conectado à saída ω dessa unidade, que segue diretamente para a Unidade de Computação. O sinal vd, que entra nessa unidade, é conectado ao FIFO e serve para solicitar uma nova constante. Quando um novo dado válido é colocado pelo Controlador de Memória no barramento bus_3, o sinal vd desse barramento ativa diretamente o FIFO e uma nova constante trigonométrica estará disponível no próximo ciclo de relógio.

3.2.6 Controlador de Memória

O Controlador de Memória é a unidade responsável por gerenciar os quatro bancos QDR conectados ao FPGA e fornecer os dados para as unidades de Computação e Comunicação. A interface *QDR II SRAM Core* permite leituras e escritas completamente independentes para cada bloco QDR. Assim, há várias maneiras de organizar as leituras e as escritas na QDR. Essa tarefa exigiu o maior esforço no projeto do coprocessador FFT.

O maior desafio é armazenar os dados distribuindo-os entre os quatro bancos QDR de forma que um par de pontos possa ser lido e gravado a cada ciclo de relógio. Destaca-se a necessidade de ler e escrever em um mesmo ciclo de relógio, pois o total de computações de um passo da FFT pode exceder o número de estágios do *pipeline* da Unidade de Computação. Este fato resulta na chegada, através do barramento bus_2, de dados processados enquanto dados ainda são lidos pelo Controlador de Memória e colocados no barramento bus_3 (ver Figura 3.6).

Cada banco de memória QDR armazena 64 bits por posição, o que nos permite guardar um número complexo (dois números em ponto flutuante de 32 bits) em cada endereço. Assim, para a leitura e escrita de um par de números complexos por ciclo, são necessários, pelo menos, dois bancos de memória QDR.

A primeira abordagem avaliada para atingir nossos objetivos (leitura e escrita de dois pontos por ciclo) foi o espelhamento de dados. Essa técnica permite a leitura paralela de dados através da manutenção de bancos de memória espelhados. Com esse esquema, é possível realizar paralelamente leituras a k posições distintas de memória, se k bancos de memória forem espelhados. Apesar disso, a gravação de

Tabela 3.1 - Posição dos dados após distribuição sequencial em 2, 3 ou 4 bancos de memória

Qtd. de blocos QDR								
2		3			4			
0	1	0	1	2	0	1	2	3
2	3	3	4	5	4	5	6	7
4	5	6	7					
6	7							

dados nesse esquema é sequencial. Para manter a consistência dos bancos, quando, por exemplo, um par de dados precisa ser gravado, são necessários dois ciclos, um para cada nova informação. Desta forma, essa abordagem foi descartada por não satisfazer as necessidades do nosso caso.

Outra alternativa que foi avaliada é a distribuição sequencial dos dados entre os blocos QDR, como a usada em esquemas de disco RAID-0 (TANENBAUM, 2007). Nesta técnica, os dados são distribuídos sequencialmente entre os blocos, sem que haja replicação dos dados. A Tabela 3.1 apresenta o resultado desse tipo de técnica quando aplicada para para distribuir os dados de um vetor com 8 pontos. Nessa simulação é considerado o uso de 2, 3 ou 4 blocos de memória QDR.

No interior de cada célula está presente o índice de cada dado armazenado. No caso da distribuição entre dois blocos de QDR, colunas 1 e 2, os elementos de índice par ficam na primeira QDR e os dados com índice ímpar em um segundo bloco de memória. As distribuições usando 3 ou 4 blocos de QDR são ilustradas nas demais colunas dessa tabela. Comparando essa tabela com o fluxo de computação da Figura 2.6, verificamos que existem conjuntos de dados onde a leitura e escrita podem ser realizadas aos pares para o processamento das operações de borboleta. Entretanto, isso não acontece para todos os casos necessários para o processamento de uma FFT completa. Outros tamanhos de vetores também foram avaliados e, devido às combinações de acessos para o cálculo de todas as borboletas, o total de bancos QDR necessário para satisfazer nossa necessidade seria igual ao número de pontos do vetor de entrada. Desta forma, essa alternativa também foi descartada para o nosso projeto.

Existem trabalhos de pesquisa na área de otimização do desenho de blocos de memória em hardware. Em grande parte, estão focados em algoritmos que definem o total

de bancos, número de portas, tempo de acesso e o agrupamento dos módulos (RAMACHANDRAN et al., 1994; SCHMIT; THOMAS, 1997). Esses trabalhos são destinados às soluções que precisam definir os recursos que serão necessários para a execução de um algoritmo em hardware, diferentemente do nosso caso, onde já conhecemos a *priori* os recursos disponíveis, sem a possibilidade de alterar essas configurações.

Cardoso et al. (2010) afirmam que embora a geração de um bom mapeamento de memória seja fundamental para o desempenho de um sistema, poucos estudos têm sido feitos sobre a atribuição automática de estruturas de dados para sistemas complexos de computação reconfigurável, especialmente pesquisas que consideram otimizações como a alocação de vetores entre bancos de memória.

Neste cenário, e como uma alternativa para distribuir os dados entre os bancos de memória disponíveis para o FPGA, desenvolvemos um esquema de endereçamento para a FFT. Esse esquema leva em conta que leituras e escritas devem ser feitas aos pares de números complexos e evita a replicação de dados. A subseção a seguir descreve o funcionamento desse sistema em mais detalhes.

3.2.6.1 Endereçamento de Memória para FFT

Endereçar eficientemente bancos de memória para a computação da FFT é um desafio. A necessidade de ler e escrever um par de dados complexos por ciclo de relógio requer uma avaliação do padrão de acesso à memória usado na computação da transformada rápida de Fourier. A Tabela 3.2 apresenta a sequência de acesso à memória para a computação da FFT sobre 8 pontos. Essa sequência de acesso pode ser observada também no fluxo de computação da FFT apresentado na Figura 2.6. A leitura dessa tabela deve ser feita aos pares de colunas e de cima para baixo. No primeiro passo, a sequência de acesso é: {0; 4}, {2; 6}, {1; 5} e {3; 7}. No segundo passo, os acessos são: {0; 2}, {1; 3}, {4; 6} e {5; 7}. Os acessos do terceiro passo iniciam em {0; 1} indo até {6; 7}.

Observando essas sequências, verificou-se que há um padrão entre dois pares de acessos de mesmo índice e os acessos do passo seguinte. Para facilitar a identificação desse padrão, as células da Tabela 3.2 foram coloridas e são apresentadas novamente na Tabela 3.3. Nesta segunda tabela, é mais fácil verificar que as posições de memória de dois acessos sequenciais, por exemplo {0; 4} e {2; 6}, aparecem novamente no passo seguinte, agrupadas conforme a ordenação interna dos pares (usando a notação dos barramentos usados nesse projeto, temos $data_1$ com $data_1$ e $data_2$ com $data_2$). As barras coloridas da Tabela 3.3 ajudam a visualizar este caso. No passo 1, as

barras aparecem na vertical e no segundo passo as barras estão na horizontal. A sequência das cores (ou dos pares) que aparece no segundo passo segue de cima para baixo da primeira coluna do passo 1, seguindo da mesma forma na segunda coluna desse passo.

Tabela 3.2 - Sequência de acesso de FFT para 8 pontos

Passo 1		Passo 2		Passo 3	
0	4	0	2	0	1
2	6	1	3	2	3
1	5	4	6	4	5
3	7	5	7	6	7

Tabela 3.3 - Destaque para 1°e 2°passos do padrão de acesso para FFT de 8 pontos

Passo 1		Passo 2		Passo 3	
0	4	0	2	0	1
2	6	1	3	2	3
1	5	4	6	4	5
3	7	5	7	6	7

Se avaliarmos a sequência de acesso do segundo passo em relação ao terceiro, chegaremos ao mesmo padrão de acesso. A Tabela 3.4 nos auxilia nessa tarefa. A única diferença nesse caso trata-se da ordem dos pares. A sequência que aparece no terceiro passo é dada pelo acesso do primeiro bloco da primeira coluna do passo 2, seguido pelo primeiro bloco da segunda coluna desse passo. Na sequência, aparecem as cores dos segundos blocos das colunas 1 e 2 do passo 2.

Tabela 3.4 - Destaque para 2°e 3°passos do padrão de acesso para FFT de 8 pontos

Passo 1		Passo 2		Passo 3	
0	4	0	2	0	1
2	6	1	3	2	3
1	5	4	6	4	5
3	7	5	7	6	7

Ao avaliarmos essa alteração entre os passos em uma FFT com uma quantidade maior de pontos, verificamos que a cada passo o número de grupos sequenciais de cores se reduz pela metade, alternando a coluna na qual o grupo aparece. Esse padrão é o mesmo explorado pela Unidade de Constantes Trigonométricas para reaproveitar os valores de constantes trigonométricas que se repetem. A Figura 2.6 foi utilizada para ilustrar essa situação.

Verificando essas sequências de acesso e conhecendo a técnica empregada na transformada rápida de Fourier, verificamos que o padrão que vemos nesses grupos de cores (pares) esta diretamente relacionado às sucessivas divisões que acontecem no algoritmo entre os índices pares e ímpares. Essas observações permitiram que organizássemos os bancos de memória QDR para que leituras e gravações de pares de dados complexos pudessem ser feitas em um mesmo ciclo de relógio.

A partir disso, desenvolveu-se um esquema de endereçamento que permite que os dados sejam gravados na ordem que serão acessados, como os dados organizados na Tabela 3.2. Esse esquema de endereçamento é ilustrado na Figura 3.12. Nessa figura, o sinal a_w representa o endereço gerado pelo próprio Controlador de Memória. Como os dados estão na ordem que são acessados, esse sinal é um contador que incrementa em 1 cada vez que um novo par de pontos é guardado nos bancos QDR. Nesse esquema de endereçamento para gravação, os blocos são agrupados em dois pares: {QDR1; QDR3} e {QDR2; QDR4}. Ambos os pares de memória compartilham os bits de índice $k-1$ até 1 do sinal a_w (k é a largura do endereçamento). O bit menos significativo desse sinal, $a_w(0)$, é utilizado para habilitar a gravação dos dados. A diferença entre os pares de QDR é que esse sinal é invertido antes de ser conectado ao sinal en_w do par {QDR2, QDR4}.

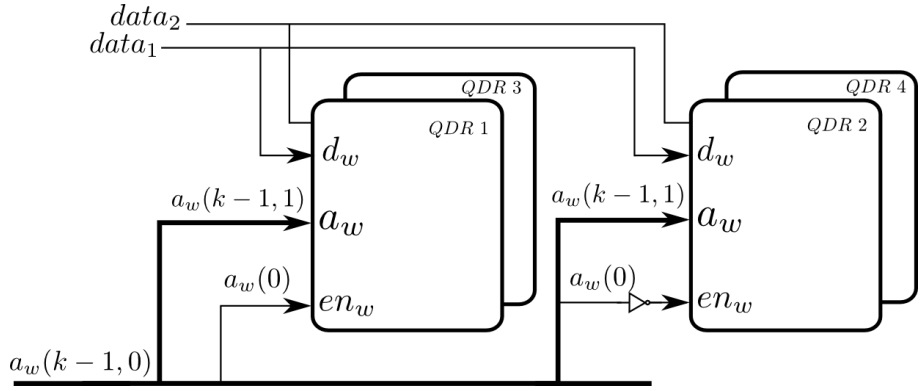


Figura 3.12 - Esquema de endereçamento para gravação de dados

Os barramentos que transportam os dados de entrada que serão guardados nos bancos são cruzados entre os pares de memória. Neste caso, $data_1$ é conectado às entradas (d_w) dos bancos QDR1 e QDR2, enquanto que $data_2$ é conectado às entradas (d_w) dos bancos QDR3 e QDR4.

Para a leitura dos dados distribuídos entre os bancos QDR, foi utilizada uma abor-

dagem parecida à usada no esquema de armazenamento. A Figura 3.13 mostra um diagrama que ilustra esse esquema. Os pares de memória {QDR1; QDR3} e {QDR2; QDR4} compartilham um sinal de endereçamento formado pela concatenação do bit $a_r(k-1)$ aos bits $a_r(k-3,0)$. O bit $a_r(k-2)$ é conectado a um registrador com deslocamento lateral com 8 posições. Esse registrador é utilizado para atrasar o uso de $a_r(k-2)$ no tempo necessário para que a leitura das QDRs seja completada. O bit menos significativo desse registrador é conectado a multiplexadores que selecionam quais dados são entregues. Assim como no esquema de gravação, o sinal a_r, que é utilizado para endereçar os blocos QDR, é gerado por um contador que é incrementado a cada nova leitura realizada. Apesar de ambos os esquemas de endereçamento compartilharem sinais de endereço e barramento de dados, não ocorre replicação de dados com a técnica apresentada.

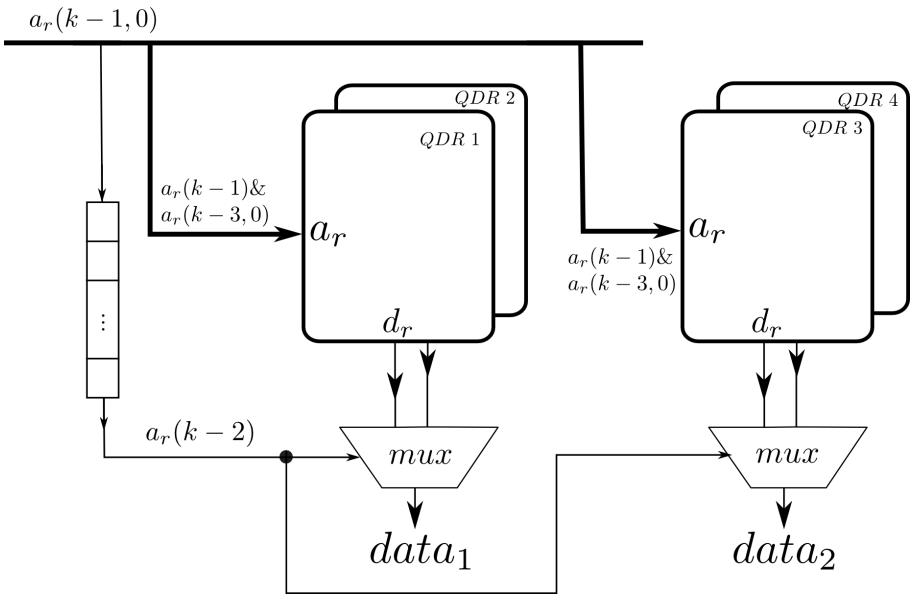


Figura 3.13 - Esquema de endereçamento para leitura de dados

Para gerar o bit *vd* no barramento de saída dessa unidade, bus_3, é utilizada a mesma técnica empregada na Unidade de Computação. Um registrador de deslocamento à esquerda de 8 bits recebe no seu bit mais significativo o sinal $active_r$ que vem da Unidade de Controle. O bit menos significativo desse registrador é conectado ao sinal *vd* do barramento de saída bus_3.

Destaca-se que, para ambos os esquemas de endereçamento, poucos recursos de hardware são utilizados. Os endereços são gerados através de contadores simples, que, provavelmente, seriam utilizados em esquemas diferentes de endereçamento.

Além disso, a manipulação dos bits para a concatenação dos sinais usados para endereçar os blocos QDR tem custo muito baixo em hardware (CARDOSO et al., 2010, p. 14), devido às suas características naturais.

3.3 Implementação

Para validar e testar o desempenho da arquitetura de coprocessador FFT proposta nesse trabalho, foram implementadas em VHDL as unidades descritas anteriormente.

O desenvolvimento e as simulações foram feitos utilizando o ambiente de desenvolvimento Ise Foundation 10.1 da Xilinx. Esta ferramenta dá suporte completo ao desenvolvimento de aplicações para o FPGA disponível no XD1 (Virtex II Pro XC2VP50-7ff1152). Como já mencionado anteriormente, utilizamos a representação em ponto flutuante do padrão IEEE 754 para a representação de números complexos. Para as operações em ponto flutuante, foi utilizada a biblioteca Xilinx Floating-Point Core que implementa o padrão IEEE 754.

As unidades implementadas em VHDL foram integradas ao *template* disponibilizado pela Cray para o desenvolvimento de aplicação para os FPGAs do XD1. Os procedimentos de síntese, mapeamento e roteamento foram todos realizados na mesma ferramenta utilizada para o desenvolvimento, Ise Foundation 10.1.

Devido às necessidades dos esquemas de endereçamento quanto a conhecer previamente o total de bits utilizados para endereçar as posições dos vetores, é necessário a síntese de uma configuração para FPGA para cada tamanho de vetor de entrada. A configuração do tamanho da FFT a ser sintetizada é feita em um único arquivo onde é necessário alterar o valor de duas constantes. O código-fonte da Listagem 3.3 apresenta um trecho do arquivo de configuração. Na linha dois é encontrada a constante c_fft_size, a qual deve ser configurada com o tamanho desejado da FFT. A outra constante a ser alterada é $c\_fft\_log2\_size$, à qual deve ser atribuído o log na base 2 do valor configurado na variável c_fft_size. Essa última constante é utilizada para calcular o tamanho dos sinais de endereçamento utilizados em todo o projeto. O tamanho do vetor de entrada, por sua vez, também é usado no cálculo do tamanho da memória ROM necessária para armazenar as contantes trigonométricas.

As demais constantes que aparecem no código-fonte da Listagem 3.3, guardam a quantidade de ciclos de operações utilizadas na implementação do coprocessador. A utilização de variáveis para armazenar esses valores é uma boa prática recomendada pela Cray (Cray Inc., 2005c) para permitir que poucas alterações precisem ser feitas na

Listing 3.3 - Arquivo de configuração do coprocessador FFT

```
...
constant c_fft_size         : integer := 128;     -- configura o tamanho de amostras da FFT
constant c_fft_size_stdvec  : std_logic_vector(18 downto 0) := conv_std_logic_vector(c_fft_size
    ,19);
constant c_fft_log2_size    : integer := 7;       -- log_2 do total de amostas (2^x)

constant c_mult_delay       : integer := 8;
constant c_add_delay        : integer := 13;

constant c_cx_mult_delay    : integer := (c_mult_delay+c_add_delay-1);
constant c_bfly_delay       : integer := (c_cx_mult_delay+c_add_delay);

constant c_qdr_latence      : std_logic_vector(3 downto 0) := "1001";
...
```

descrição do hardware para a sua execução em sistemas que, por exemplo, possuam tempos diferentes de acesso a memórias QDR.

De posse da implementação em VHDL do coprocessador FFT, foram gerados nove arquivos de configuração (*bitstream*) para o FPGA do Cray XD1. Cada arquivo foi configurado para os seguintes tamanhos de vetores de entrada: 64, 128, 256, 512, 1024, 2048, 4096, 8192 e 16384. A síntese para tamanhos maiores de vetores de entrada não foi possível devido às limitações de quantidade de memória *Block RAM* no FPGA. As propriedades que configuram as estratégias usadas no processo de síntese foram mantidas conforme a configuração padrão do *template* fornecido pela Cray. A Tabela 3.5 apresenta essas propriedades e as configurações usadas.

Tabela 3.5 - Opções de síntese utilizada no Ise Foundation 10.1

Propriedade	Configuração
Optimization Goal	Speed
Optimization Effort	Normal
Global Optimization Goal	AllClockNets
Design Goal	Balacead
Design Stratety	Xilinx Default

Na Tabela 3.6 são apresentados os recursos ocupados e as frequências máximas de operação obtidas para o FPGA Virtex II Pro XC2VP50-7ff1152. São descritos o número de *flip-flops* (FFs), *lookup tables* (LUTs), *Slices* e a frequência máxima de operação da descrição implementada. Observa-se nessa tabela, que, com o aumento da quantidade de pontos, existe um aumento quase linear do aumento de recursos necessários. Esse aumento é bem discreto, sendo que entre a versão com menos

pontos (64) e a versão com mais pontos (16384) as diferenças não passam de 80 FFs, 76 LUTs, 54 *Slices*. Esses valores representam menos de 1% de incremento em relação a versão que utilizou menos recursos. As frequências máximas encontradas variaram entre 203,042 a 212,802 MHz, sendo sempre superiores a frequência máxima permitida para os FPGAs do XD1, 199MHz (Cray Inc., 2005b).

Tabela 3.6 - Dados da síntese do coprocessador FFT

N. Amostras	FFs	LUTs	*Slices*	Freq. Max. (MHz)
64	12816	10240	8530	208,026
128	12826	10241	8535	203,042
256	12836	10259	8537	203,042
512	12846	10268	8547	203,042
1024	12856	10286	8552	203,042
2048	12866	10298	8559	212,802
4096	12876	10294	8569	212,802
8192	12886	10303	8580	212,802
16384	12896	10316	8584	212,802

Além da descrição de hardware, foi desenvolvido, em linguagem C, um conjunto de funções para auxiliar a integração da FFT em FPGA com o software. O protótipo de cada função pode ser visto no código-fonte da Listagem 3.4

A função $fpga\_init$ é responsável por fazer a inicialização do FPGA, carregando o *bitstream* correto dependendo do tamanho da FFT a ser calculada. O argumento n é utilizado para auxiliar nessa tarefa. As funções $fpga\_fft$ e $fpga\_ifft$ são responsáveis, respectivamente, pelo cálculo da FFT direta e inversa. A função $fpga\_fft$ compartilha dois endereços de memória com o FPGA. O primeiro endereço refere-se aos dados de entrada, enquanto que o segundo é o local onde o resultado deve ser colocado pelo FPGA. Ao final do processamento da FFT, essa função retorna o ponteiro de memória com o resultado da computação. Para o cálculo da transformada inversa de Fourier é utilizada a mesma configuração em hardware. Isso é possível, pois calcular a transformada inversa de Fourier é equivalente ao cálculo de:

$$IFFT(x) = \frac{1}{N} conj(FFT(conj(x)) \tag{3.6}$$

onde $conj$ é o conjugado complexo. Desta forma, a função $fpga\_ifft$, realiza o cálculo do conjugado complexo do vetor de entrada (em termos práticos isso con-

Listing 3.4 - Protótipos de funções para cálculo da FFT em FPGA

```
int fpga_init(int n);
complex float *fpga_fft(complex float *v, int n);
complex float *fpga_ifft(complex float *v, int n);
int fpga_close();
```

siste na troca entre os termos reais e imaginários de cada elemento) e chama a função $fpga\_fft$. O conjugado é realizado sobre o retorno da função, dividindo cada elemento por n. As computações extras realizadas pela função $fpga\_ifft$ têm complexidade $O(N)$. Por fim, a função $fpga\_close$ é responsável por fechar o FPGA (a *enlib* trata o FPGA como um arquivo).

Com essas funções implementadas em linguagem C, foi possível implementar as operações necessárias para o cálculo do *Fast Poisson Solver* Híbrido. Essas funções foram agrupadas em uma biblioteca chamada FPSH que utiliza CPU e FPGA na solução de equações de Poisson. Essa biblioteca é apresentada na próxima seção.

3.4 Biblioteca FPSH

A partir da elaboração do coprocessador FFT e do conjunto de funções que auxiliam o uso desse recurso, foi possível, então, desenvolver uma biblioteca responsável por implementar o *Fast Poisson Solver* Híbrido que realiza a execução da arquitetura híbrida apresentada no início dessa seção. Essa biblioteca, chamada de FPSH, possui três funções que são exibidas no código fonte da Listagem 3.5

A função $fpsh\_init$ é responsável por chamar a função responsável pela configuração do FPGA e por calcular o valor da diagonal da matriz Λ^{-1} (ver Equação 3.2) usada na solução do *fast Poisson solver*. Um vetor complexo de N posições é utilizado para armazenar esses valores. A função $fpsh\_close$ faz uma chamada para a função responsável por fechar o FPGA e desaloca a memória usada para armazenar os dados da matriz Λ^{-1}.

Listing 3.5 - Propótipos das funções da biblioteca FPSH

```
int fpsh_init(int n);
int fpsh_close(int n);
float *fpsh_pois1d(float *v, int n);
```

A função responsável por resolver as equações de Poisson é a $fpsh\_pois1d$. Essa função recebe um ponteiro para um vetor de ponto flutuante simples e um inteiro, que representa o tamanho da entrada a ser calculada. O primeiro parâmetro deve conter o endereço dos dados que serão processados. Essa função retorna um ponteiro contendo o resultado da solução de Poisson processado. O algoritmo implementado pela função $fpsh\_pois1d$ segue exatamente a sequência descrita no pseudocódigo da Listagem 3.1.

4 APLICAÇÃO

Uma aplicação em software foi escolhida para ser integrada a nossa solução de *Fast Poisson Solver* Híbrido. O objetivo é validar e testar o funcionamento da biblioteca FPSH em uma aplicação que necessita da solução de equações de Poisson. A aplicação selecionada para essa finalidade foi o DYNAMO. A seção a seguir apresenta brevemente os detalhes desse sistema.

4.1 Modelo DYNAMO

O DYNAMO é um modelo meteorológico unimensional que simula vários fenômenos de importância na dinâmica atmosférica, sendo deduzido a partir de equações de água rasa, envolvendo a evolução das variáveis primitivas (componentes horizontais do vento e geopotencial) ou a evolução das variáveis de prognóstico (vorticidade, divergência e geopotencial) (LYNCH, 1984).

Entre as principais hipóteses assumidas, destacam-se: a atmosfera é assumida como um fluido incompressível e em equilíbrio hidrostático com a vertical e o campo de velocidades independente da componente z; admite-se atmosfera seca e isentrópica; fronteira livre para a altura geopotencial; a superfície (fronteira no nível $z = 0$) é considerada plana; assume-se condições de contorno periódicas para todas as variáveis dependentes perturbadas.

O modelo de água-rasa num sistema de referência não inercial é descrito pelas equações

$$\frac{du}{dt} - fv + \frac{\partial \Phi}{\partial x} = 0 \tag{4.1}$$

$$\frac{dv}{dt} + fu + \frac{\partial \Phi}{\partial y} = 0 \tag{4.2}$$

$$\frac{d\Phi}{dt} + \Phi\left(u_x + v_y\right) = 0 \tag{4.3}$$

onde $d/dt \equiv \partial/\partial t + \nabla\vec{v}$ é a derivada substantiva; x, y são as coordenadas no sentido oeste e norte; t é o tempo; $\Phi = gh$ o geopotencial, sendo h a altura do fluido acima da superfície plana; u, v são as velocidades zonal e meridional. O parâmetro de Coriolis é assumido como $f = f_0 + \beta y$, com f_0 e β constantes.

Se o vento predominante escoa na direção zonal, puder ser considerado constante e estar geostroficamente balanceado com o geopotencial ($f\bar{u} = -\partial\bar{\Phi}/\partial y$) e os desvios

nas variáveis do modelo são grandezas unidimensionais

$$u = \bar{u} + u'(x,t); \quad v = v'(x,t); \quad \Phi = \bar{\Phi}(y) + \Phi'(x,t)$$

a dependência em y é eliminada subtraindo-se a relação geostrófica da eq. (4.2). As equações da vorticidade ($\zeta = v_x$) e da divergência ($\delta = u_x$) são obtidas aplicando-se os operadores gradiente e divergente nas equações do movimento (componentes da velocidade):

$$\frac{d\zeta}{dt} + \frac{(du\zeta)}{dx} + \delta f + \beta\, v = 0 \tag{4.4}$$

$$\frac{d\delta}{dt} + \frac{(du\delta)}{dx} - \zeta f + \beta\, u' + \Phi_{xx} = 0 \tag{4.5}$$

$$\frac{d\Phi}{dt} + \frac{(du\Phi)}{dx} - f\bar{u}v + \bar{\Phi}\,\delta = 0 \tag{4.6}$$

a equação (4.6) é obtida incorporando-se o balanço geostrófico à equação da continuidade (4.3).

O sistema (4.4)—(4.6) forma o conjunto de equações básicas do modelo, onde as variáveis prognósticas (ζ, δ, Φ) dependem somente de x e t. A dependência na direção y ocorre somente no parâmetro f e no geopotencial médio $\bar{\Phi}$; mas argumentos de escala mostram que a dependência é tão pequena que f e $\bar{\Phi}$ podem ser tomados como constantes.

O campo de vento horizontal $\vec{v}_H = u\,\vec{i} + v\,\vec{j}$ pode ser descrito como a soma de duas compoentes: não divergentes e irrotacionais:

$$\vec{v}_H = \vec{v}_\chi + \vec{v}_\psi = \vec{k} \times \vec{\nabla}\psi + \vec{\nabla}\chi \;;$$

onde ψ é a função corrente e χ o potencial de velocidade. Desta maneira, a vorticidade e a divergência estão associadas a estas funções por equações de Poisson correspondentes:

$$\nabla^2\psi = \zeta \qquad \nabla^2\chi = \delta \;. \tag{4.7}$$

e as componentes horizontais do vento podem ser atualizadas de acordo com o teorema da decomposição citado acima:

$$v = \vec{\nabla}\psi \qquad u = \vec{\nabla}\chi \;.$$

Então, para o presente modelo deve-se resolver, em cada passo de integração, duas equações de Poisson.

O núcleo computacional do algoritmo do modelo Dynamo está relacionado ao cálculo da função corrente Ψ e do potencial de velocidade χ, relacionadas respectivamente à vorticidade ζ e à divergência δ. Estas variáveis são obtidas através da solução de duas equações de Poisson (ver Equações 4.7), empregadas no cálculo das velocidades zonal u e meriodional v (LYNCH, 1984)

$$u = \frac{\partial \chi}{\partial x}; \qquad\qquad v = \frac{\partial \Psi}{\partial x}. \tag{4.8}$$

Aplicando a forma clássica de diferenças finitas centradas para a solução da equação de Poisson com condição de contorno periódica temos (CAMPOS VELHO; CLAEYSSEN, 1992)

$$A_\Delta \Psi = Z \tag{4.9}$$

onde $\Psi = [\psi_1, \psi_2, \ldots, \psi_{N_x}]^T$, $Z = [\zeta_1, \zeta_2, \ldots, \zeta_{N_x}]^T$ e

$$A_\Delta = \begin{bmatrix} -2 & 1 & & & & 1 \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 \\ 1 & & & & 1 & -2 \end{bmatrix}.$$

As aproximações e discretizações de modelos numéricos podem levar a equações singulares onde a matriz de coeficientes não é inversível, como A_Δ do problema apresentado. Para estas situações, o uso da inversa de Moore-Penrose (*M-P inverse*), através da decomposição em valores singulares (SVD) é uma solução apropriada, pois representa a solução por mínimos quadrados do sistema (CAMPOS VELHO; CLAEYSSEN, 1992).

Seja A uma matrix $m \times n$. Então uma matrix $X_{m\times n}$ que satisfaz algumas ou todas as seguintes propriedades (condições de Penrose) é chamada de inversa generalizada

$$(1) AXA = A; \qquad\qquad (2) XAX = X; \tag{4.10}$$

$$(3)(AX)^* = AX; \qquad\qquad (4)(XA)^* = XA. \tag{4.11}$$

onde $*$ representa o conjugado transposto. Uma matriz que satisfaz todas essas

propriedades é chamada de inversa de Moore-Penrose de A, é única (CAMPBELL; MEYER, 2009) e denotada por A^+.

Para propósitos computacionais, a definição acima mencionada não é prática. Contudo, há diversos algoritmos para encontrar a inversa de Moore-Penronse. Umas das maneiras é através da decomposição em valores singulares, grandemente utilizada em equações matriciais retangulares ou quadradas (CAMPOS VELHO; CLAEYSSEN, 1992).

Considere A uma matriz de ordem $m \times n$, com elementos complexos e posto r. Existem matrizes unitárias U, V de ordens m e n, respectivamente, tais que

$$A = UDV^* \qquad\qquad D = \begin{bmatrix} D_1 & 0 \\ 0 & 0 \end{bmatrix} \tag{4.12}$$

onde (*) representa o conjugado transposto, D é uma matriz de ordem $m \times n$ e $D_1 = diag[d_1, d_2, \ldots, d_r]$ é uma matriz diagonal não singular de ordem r.

A partir da decomposição, é possível calcular a inversa de Moore-Penrose (A^+). Desta forma, se $A = U^*DV^*$, onde U e V são matrizes unitárias, então

$$A^+ = VD^+U; \qquad\qquad D^+ = \begin{bmatrix} D_1^{-1} & 0 \\ 0 & 0 \end{bmatrix}. \tag{4.13}$$

Para a equação matricial $Ax = b$, a solução $x = A^+b$ é compreentendida como a solução de mínimos quadrados do sistema (CAMPOS VELHO; CLAEYSSEN, 1992).

Nos casos onde a matriz é circulante, a inversa de Moore-Penrose pode ser calculada de uma forma mais simples. Uma matriz circulante C de ordem n é uma matriz na forma

$$C = circ(c_1, c_2, \ldots, c_n) = \begin{bmatrix} c_1 & c_2 & \ldots & c_n \\ c_n & c_1 & \ldots & c_{n-1} \\ \vdots & \vdots & & \vdots \\ c_2 & c_3 & \ldots & c_1 \end{bmatrix}; \tag{4.14}$$

e pode ser diagonalizada com auxílio de matrizes de Fourier F_n^* e F_n (DAVIDS, 1979)

$$C = F_n^* \Lambda F_n; \tag{4.15}$$

e Λ é a matriz diagonal

$$\Lambda = \sum_{k=0}^{n-1} c_{k+1}(\Omega_n)^n; \qquad \Omega = diag[1, \omega, \omega^2, \ldots, \omega^{n-1}]; \qquad \omega = e^{\frac{2\pi i}{n}}. \tag{4.16}$$

Desta forma, se C é uma matriz circulante, sua matriz inversa de Moore-Penrose é a matriz circulante (DAVIDS, 1979; CAMPOS VELHO; CLAEYSSEN, 1992)

$$C^+ = F_n^* \Lambda^+ F_n; \tag{4.17}$$

onde

$$\Lambda^+ = diag[\lambda_1^+, \lambda_2^+, \ldots, \lambda_n^+]; \qquad \lambda_k^+ = \begin{cases} \frac{1}{\lambda_k} & \text{se } \lambda_k \neq 0 \\ 0 & \text{se } \lambda_k = 0 \end{cases}. \tag{4.18}$$

Com esta formulação, é possível utilizar nossa solução de *Fast Poisson Solver* Híbrido para o cálculo da função corrente Ψ e do potencial de velocidade χ do modelo DYNAMO.

4.2 Integração com o *Fast Poisson Solver* Híbrido

O modelo DYNAMO é implementado em Fortran e a subrotina responsável pela solução das equações de Poisson é chamada POIS1D. Para acoplar nossa biblioteca FPSH escrita em linguagem C neste modelo, foi utilizado o compilador $ifort$ da Intel (Intel Corporation,) com o auxílio do parâmetro -f77rtl, que indica o uso de Fortran 77 ao compilador e de formatação de nomes compatíveis com o compilador *gcc*. Uma versão totalmente em software da biblioteca FPSH foi implementada para servir como referência para análise dos resultados e testes de desempenho. Essa versão usa o mesmo algoritmo da versão híbrida, sendo feitas somente alterações nas funções que calculam a FFT direta e inversa. Na versão de referência, elas são implementadas em C e o executável é gerado pelo compilador *gcc* com a *flag* de otimização $-O3$. Os resultados obtidos com os testes são apresentados e discutidos na próxima seção.

5 RESULTADOS E DISCUSSÕES

Nesta seção são apresentados os resultados dos testes realizados para avaliar o desempenho da arquitetura de *Fast Poisson Solver* Híbrido proposta neste trabalho, a qual é implementada através da biblioteca FPSH. Para isso foram definidos três testes para avaliar nosso projeto sob diferentes óticas. O primeiro teste faz uma avaliação da solução híbrida em relação a uma aplicação de referência implementada em software para esse propósito. O objetivo desse teste é fazer uma comparação numérica dos resultados e obter o *speedup* da versão híbrida em relação a uma versão completamente em software.

Motivada pelos resultados do primeiro teste, uma segunda avaliação foi feita sob o ponto de vista da execução da FFT no hardware. O número de ciclos foi a métrica utilizada neste teste, objetivando identificar o perfil de execução da FFT no FPGA. Uma avaliação da eficiência da transferência de dados entre CPU e FPGA é realizada. Além disso, verifica-se o total de operações de ponto flutuante por segundo obtido em nossa arquitetura de coprocessador FFT.

Por fim, realizou-se uma análise que considera o desempenho da execução da FFT em FPGA em relação a uma execução puramente em software. Esse teste considera o tempo de computação percebido pela aplicação em software que utiliza o coprocessador FFT. Neste teste, compara-se os tempos de execução frente a uma tradicional biblioteca de FFT e calcula-se o número de operações em ponto flutuante por segundo para ambas as aplicações.

Para todos os testes, utilizou-se um *blade* de um sistema Cray XD1 e foram utilizados vetores de entrada com 64, 128, 256, 512, 1024, 2048, 4096, 8192 e 16384 pontos. As execuções em software ocorreram em uma CPU do *blade* (AMD Opteron 64 bits a 2,4GHz), enquanto que o coprocessador FFT foi executado em FPGA (Virtex II Pro XC2VP50 a 190MHz). As próximas seções descrevem os detalhes desses três testes, apresentam os resultados e, por fim, algumas discussões.

5.1 *Speedup* do *Fast Poisson Solver* Híbrido

O primeiro teste do *Fast Poisson Solver* Híbrido consistiu na avaliação numérica dos resultados e na comparação dos tempos de processamento em relação à aplicação de referência. Nesse teste, cada versão da biblioteca FPSH (híbrida e somente em software) foi integrada ao DYNAMO e uma bateria de testes foi executada para os variados tamanhos de vetores. As saídas do modelo em ambas versões foram

comparadas e os valores obtidos foram iguais. Esse resultado já era aguardado, uma vez que as operações realizadas em FPGA utilizam a mesma representação numérica utilizada pela CPU. Além disso, esse resultado nos mostra que a solução da equação de Poisson com o auxílio do coprocessador FFT é transparente para a aplicação, em termos do recurso utilizado para a computação.

Após a validação dos resultados obtidos pela execução do DYNAMO com a biblioteca FPSH, a função POIS1D foi isolada do modelo DYNAMO para a execução de testes de desempenho. Foram usados dados sintéticos como vetor de entrada para a função POIS1D. Esses vetores são gerados aleatoriamente, sendo usado um mesmo conjunto de dados para uma execução totalmente em software e uma execução híbrida. Os tempos médios de 10 execuções de cada versão são apresentados na Tabela 5.1.

A primeira coluna traz a quantidade de pontos do vetor de entrada de cada teste realizado. As duas colunas seguintes apresentam respectivamente o tempo médio de execução da versão de referência e da versão que utiliza o coprocessador FFT combinado com CPU para a execução do *fast Poisson solver*. Na quarta e na quinta coluna são apresentados os desvios-padrão dos tempos das 10 execuções de cada teste. A última coluna apresenta a aceleração obtida com a utilização da arquitetura proposta de *Fast Poisson Solver* Híbrido. O *speedup* é calculado como a razão entre o tempo da implementação de referência e o tempo da versão híbrida.

Tabela 5.1 - Tempos médios de execução para as versões em software e híbrida da função POIS1D

Pontos	Tempo médio (μs)		Desvio Padrão		*Speedup*
	Software	Híbrida	Software	Híbrida	
64	58,00	23,70	1,52	0,64	2,45
128	104,30	32,60	1,60	1,50	3,20
256	183,80	54,90	1,89	0,70	3,34
512	365,80	105,10	1,92	2,77	3,48
1024	728,00	201,30	2,26	4,08	3,62
2048	1482,00	384,80	1,50	3,82	3,85
4096	3059,20	783,60	2,45	6,90	3,90
8192	6391,10	1595,70	3,47	13,83	4,01
16384	15040,10	3241,50	4,18	22,92	4,64

Através desses resultados, observamos que a versão híbrida executou em menor tempo para todos os tamanhos de vetor de entrada testados. O menor ganho foi

observado para a execução com 64 pontos, enquanto que a maior vantagem foi obtida pelo experimento que utilizou vetores com 16384 pontos. Além disso, verifica-se que a aceleração da versão híbrida em relação à versão de referência aumenta quanto maior é a quantidade de pontos dos vetores de entrada. Esse aumento relativo entre as versões também pode ser observado na Figura 5.1, que apresenta na forma de gráfico os tempos coletados para os testes de cada versão. O eixo vertical desse gráfico está em escala logarítmica e observa-se que as linhas que representam os tempos medidos se afastam suavemente com o aumento do número de pontos.

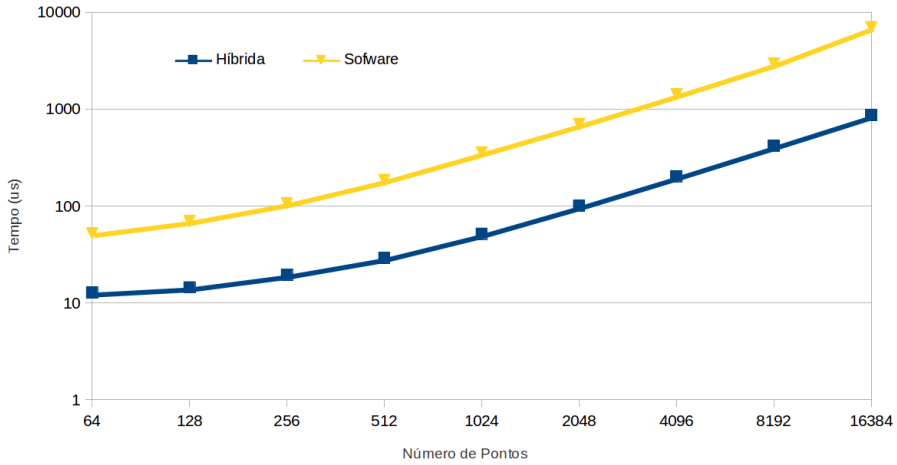


Figura 5.1 - Gráfico com os tempos de execução para as versões em software e híbrida da função POIS1D (eixo vertical em escala logarítmica)

Na maioria dos testes, o desvio padrão é inferior para as execuções da versão somente em software comparados com a versão híbrida. Apesar de não haver um padrão claro, os valores de desvio padrão para ambas as versões tendem a aumentar com o número de pontos, aumentando a distância entre os valores da versão híbrida em relação ao testes que utilizam somente CPU.

Por sofrer uma menor intervenção de fatores como sistema operacional, acesso a disco, cache, etc, esperava-se que a execução híbrida tivesse uma menor variação nos tempos de execução, uma vez que a maior parte da computação é feita no FPGA. A justificativa para esse comportamento pode estar relacionada à transferência de dados entre CPU e FPGA. Para verificar essa hipótese, foram adicionados três registradores à Unidade de Controle do coprocessador FFT para contabilizar o total de ciclos gastos com as etapas de: (a) leitura dos dados; (b) processamento; e (c) envio dos dados. As etapas de síntese, roteamento e mapeamento foram novamente

realizadas para os tamanhos de entrada utilizados nos testes anteriores e realizou-se uma nova bateria de execuções. Os detalhes desse teste e as métricas coletadas são apresentados a seguir.

5.2 Métricas do coprocessador FFT

No sentido de avaliar com detalhes o processamento da transformada de Fourier pelo coprocessador FFT projetado nesse trabalho, três contadores foram adicionados à Unidade de Controle. Esses contadores estão associados aos estados da MEF utilizada nessa unidade e contabilizam o total de ciclos que o hardware gasta para processar cada etapa.

A contagem do número de ciclos necessários para o recebimento de dados inicia a partir do momento em que a Unidade de Controle solicita o recebimento dos dados para a Unidade de Comunicação. Quando essa unidade envia o sinal que indica o término dessa tarefa, a contagem é parada.

O total de ciclos necessários para o processamento da FFT são contabilizados a partir do momento em que a Unidade de Controle ativa as leituras pelo Controlador de Memória e para quando o último dado processado volta ao Controlador de Memória.

Para contar o número de ciclos gastos para enviar os dados do FPGA para o *blade*, é utilizada a mesma abordagem usada para contar a chegada dos dados. Quanto o sinal $enable_w$ é ativado, inicia-se a contagem que só é parada com o recebimento do sinal $done_w$, enviado pela Unidade de Comunicação.

Novamente 10 execuções para cada tamanho de vetor de entrada foram realizadas e, através de leituras da CPU, foram obtidas as quantidades de ciclos gastos em cada etapa da FFT executada em FPGA. A quantidade média de ciclos necessários para calcular cada etapa e o desvio padrão dessas contagens são apresentados na Tabela 5.2. A primeira coluna identifica o tamanho do vetor de entrada, enquanto que as próximas três colunas apresentam as médias de ciclos necessários para o recebimento dos dados (Rec.), processamento (Proc.) e envio dos dados (Env.). A quinta coluna apresenta o somatório desses três valores (Rec.+Proc.+Env.) As últimas três colunas dessa tabela apresentam o desvio padrão das métricas coletadas para recebimento, processamento e envio, respectivamente.

Analisando os dados da Tabela 5.2, verifica-se, como levantado anteriormente, que as variações no tempo de processamento realmente estão associadas às interações entre CPU e FPGA. No caso de recebimento dos dados, para todos os tamanhos

Tabela 5.2 - Número médio de ciclos e desvio padrão das etapas do coprocessador FFT

Pontos	Total de Ciclos				Desvio Padrão		
	Rec.	Proc.	Env.	Soma	Receb.	Proc.	Env.
64	216,5	480	90	786,5	8,6	0	0
128	288,1	784	162	1234,1	6	0	0
256	429,8	1408	306	2143,8	7,4	0	0
512	710,1	2736	594	4040,1	8,2	0	0
1024	1267,7	5600	1383,2	8250,9	5,2	0	6,3
2048	2392,3	11792	3334,5	17518,8	5,7	0	7,8
4096	4618,4	25152	7252,5	37022,9	5,4	0	14,4
8192	9097,9	53872	15039,6	78009,5	6,0	0	5,0
16384	18039	115360	30678,8	164077,8	9,1	0	39,7

de vetores de entrada houve variação da quantidade de ciclos gastos. Para o envio dos dados, essas variações apareceram em testes com mais de 1024 amostras, sendo mais significativo para o teste com 16384 pontos, onde o desvio padrão foi de 39,7.

Para a etapa de processamento da FFT, não houve qualquer variação na quantidade de ciclos gastos. Esse comportamento é esperado, pois todas as atividades dessa etapa ocorrem sob o controle da nossa aplicação e os custos de cada etapa são fixos e conhecidos. Este fato pode ser confirmado pela possibilidade de equacionar a quantidade de ciclos gastos na etapa de processamento da FFT. Essa quantidade, apresentada na terceira coluna da Tabela 5.2, pode ser calculada por

$$\#ciclos = \log_2(N)\left(\frac{N}{2} + 48\right), \tag{5.1}$$

onde N é o total de pontos do vetor de entrada. O primeiro termo desse produto, $\log_2(N)$, refere-se ao total de passos necessários para o cálculo de uma FFT de N pontos. O segundo operando, $N/2 + 48$, é a quantidade de ciclos necessários para processar um passo de uma FFT.

Como nossa arquitetura de processamento é baseada em um *pipeline*, e o total de ciclos necessários para calcular k operações em um *pipeline* de s estágios é igual a $k + s$ ciclos, confirmamos que $N/2$ operações (borboletas) são calculadas por passo da FFT e temos uma latência de **48 ciclos** no nosso *pipeline*. Essa quantidade de ciclos é conhecida e contabilizada da seguinte forma:

1 ciclo: usado pela Unidade de Controle para colocar o sinal $active_r$ em um registrador conectado na saída dessa unidade;

1 ciclo: usado para a formação do endereço de leitura pela concatenação de sinais no Controlador de Memória;

8 ciclos: latência de leitura da memória QDR II SRAM;

1 ciclo: gasto pelo Controlador de Memória para colocar os dados lidos em um registrador de saída conectado ao barramento bus_3;

34 ciclos: utilizados para o cálculo de uma borboleta pela Unidade de Computação;

1 ciclo: gasto pela Unidade de Controle para colocar os dados processados em um registrador de saída conectado ao barramento bus_2;

1 ciclo: usado na gravação dos dados na QDR II SRAM; e

1 ciclo: gasto na troca de estados (*computing* e *waiting*) da MEF da Unidade de Controle.

Outras análises também podem ser feitas a partir dos dados coletados nesse segundo teste. Uma delas é referente à eficiência das etapas que envolvem transferência de dados entre CPU e FPGA.

Sabendo que é possível transferir da CPU para o FPGA até 64 bits por ciclo de relógio, e que um número complexo utilizado no nosso projeto utiliza essa mesma quantidade de bits, teoricamente o FPGA poderia receber um ponto por ciclo, sendo

$$\#Rec_{teorico} = N \tag{5.2}$$

onde N é o número de pontos.

Para o envio dos dados do FPGA para a CPU, por outro lado, é gasto um ciclo de relógio por número complexo e a cada oito ciclos é necessário aguardar um ciclo de relógio para enviar mais dados, conforme especificação do fabricante do XD1 (Cray Inc., 2005d). Desta forma, o FPGA pode teoricamente enviar

$$\#Env_{teorico} = N + (\frac{N}{8} - 1) \tag{5.3}$$

onde N é o total de números a serem enviados, $(\frac{N}{8} - 1)$ é a quantidade de vezes que será necessário aguardar entre o envio de uma sequência de oito pontos.

Para obter o percentual médio da eficiência alcançada em relação ao limite teórico, podemos calcular a razão entre o número teórico de ciclos pelo total de ciclos gastos

em cada uma das etapas de transferência de dados. O valor máximo possível para essa conta é 1, o que representaria que, a cada ciclo, um novo ponto foi transferido. Um valor de 0,5, por exemplo, representa que em média dois ciclos foram gastos para enviar cada número complexo. A Tabela 5.3 apresenta os percentuais médios calculados.

Tabela 5.3 - Eficiência calculada para o recebimento e envio de dados entre FPGA e CPU

Pontos	Eficiência		
	Rec.	Env.	Média
64	0,30	0,79	0,54
128	0,44	0,88	0,66
256	0,60	0,94	0,77
512	0,72	0,97	0,84
1024	0,81	0,83	0,82
2048	0,86	0,69	0,77
4096	0,89	0,64	0,76
8192	0,90	0,61	0,76
16384	0,91	0,60	0,75

A primeira coluna dessa tabela contém os tamanhos dos vetores de entrada, enquanto que a segunda e a terceira colunas apresentam respectivamente o percentual médio de transferência alcançado em relação ao limite teórico para o recebimento de dados (Rec.) e envio de dados (Env.). Na última coluna é apresentada a média desses valores.

Na transferência de dados da CPU para o FPGA, a eficiência cresceu com o aumento do número de pontos dos vetores de entrada. Esse crescimento foi se atenuando quanto maior a quantidade de pontos transportados. Esse comportamento pode ser mais facilmente identificado através da Figura 5.2. A curva azul com losangos apresenta a eficiência calculada do recebimento dos dados. A menor eficiência para esse caso foi de 30%, enquanto que no melhor caso, o recebimento ocorreu a 91% do limite teórico. Esse valor pode ser considerado muito satisfatório, uma vez que em termos práticos dificilmente se chega aos limites teóricos.

Os resultados do cálculo da eficiência para o envio dos dados apresentam um comportamento diferente do recebimento. A eficiência inicia em um patamar superior (79%), crescendo até atingir 97% com 512 pontos transferidos. Após esse tamanho, existe uma queda acentuada com uma aparente estabilização próxima aos 60%. A

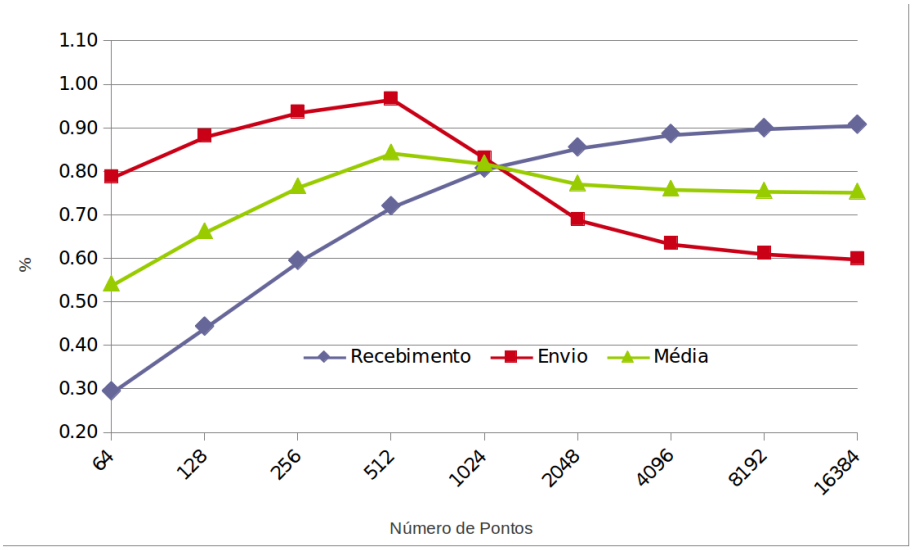


Figura 5.2 - Eficiência da transferência de dados entre CPU e FPGA

curva vermelha com quadrados do gráfico da Figura 5.2 ilustra esse comportamento, que pode ser justificado pelo uso de *cache* pelo *RTCore*. Essa interface utiliza um buffer de 512×64 bits (Cray Inc., 2005d, p. 12), o qual é suficiente para guardar até 512 pontos, indo ao encontro com a queda de desempenho a partir dessa quantidade de números complexos transferidos.

A eficiência média é de 54% para a menor quantidade de pontos e de 75% para a maior quantidade. O pico acontece também nos 512 pontos, atingindo 85%. A curva média apresenta uma certa tendência em estabilizar por volta dos 75% com o aumento do número de pontos.

Outra análise que pode ser feita com os dados desse segundo teste é quanto à quantidade de operações em ponto flutuante por segundo (FLOPS - *Floating point Operations Per Second)*) realizadas pela Unidade de Processamento do coprocessador FFT. Esse valor pode ser calculado através da divisão do número de operações em ponto flutuante realizadas para o processamento de uma FFT pelo tempo necessário para a etapa de processamento. Como a quantidade de ciclos pôde ser equacionada, o tempo para o processamento de uma FFT de N pontos é dado por

$$T_N = \#ciclos \frac{1}{freq}$$

$$= log_2 \left(\frac{N}{2} + 48 \right) \frac{1}{freq}, \tag{5.4}$$

onde T_N é o tempo de processamento, $\#ciclos$ é o número de ciclos e $freq$ é a frequência de execução do hardware em Hertz (Hz). Uma operação borboleta realiza 10 operações em ponto flutuante (ver Figura 3.10) e na computação de uma FFT são necessárias $(N/2)\log_2(N)$ borboletas. Dessa forma, o número de operações em ponto flutuante por segundo é

$$FLOPS_N = \frac{10\left(\frac{N}{2}\log_2(N)\right)}{T_N} \tag{5.5}$$

$$= \frac{10\left(\frac{N}{2}\log_2(N)\right)}{log_2\left(\frac{N}{2}+48\right)\frac{1}{freq}}$$

$$= \frac{10N}{N+96}freq. \tag{5.6}$$

O máximo de FLOPS que nossa arquitetura de coprocessador FFT realizaria se os recursos do FPGA fossem ilimitados e um vetor muito grande fosse processado seria de $10freq$, que é obtido por

$$\lim_{N\to\infty}\frac{10N}{N+96}freq = 10freq. \tag{5.7}$$

Na frequência utilizada pelo FPGA nos testes realizados, esse valor seria de 1,9 GFLOPS. Esse valor logicamente não é possível de ser obtido, mas a execução com 16384 pontos chegou a um valor correspondente a 99% desse valor teórico. A Tabela 5.5 apresenta o número de FLOPS calculado para cada tamanho de entrada usados nos testes.

Tabela 5.4 - Quantidade de operações em ponto flutuante por segundo computadas pela Unidade de Computação

Pontos	MFLOPS
64	760,00
128	1085,71
256	1381,82
512	1600,00
1024	1737,14
2048	1814,93
4096	1856,49
8192	1877,99
16384	1888,93

A primeira coluna apresenta o número de pontos, enquanto que a segunda coluna dessa tabela apresenta o valor calculado em milhões de operações de ponto flutuante por segundo (MFLOPS). A Figura 5.3 apresenta um gráfico com uma curva formada com esses dados calculados. Esses valores são promissores quando comparados aos resultados de uma biblioteca amplamente utilizada para o cálculo da transformada de Fourier em software (FFTW) (FRIGO; JOHNSON, 2012). A fim de avaliar a execução do coprocessador FFT frente a uma execução em software, um terceiro e último teste foi estabelecido. A próxima seção apresenta os detalhes desse teste.

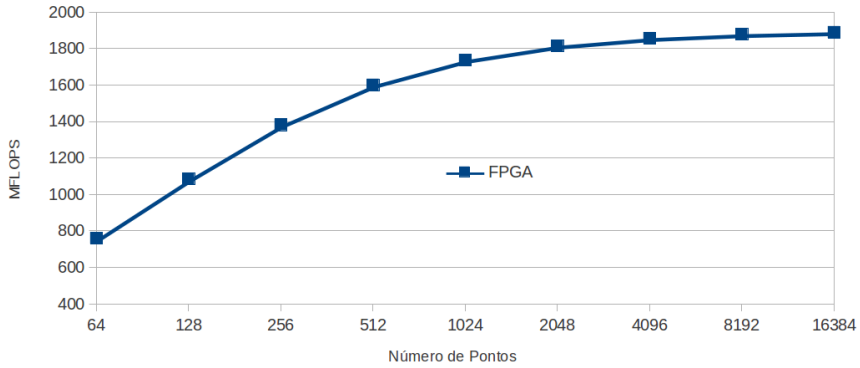


Figura 5.3 - Gráfico da quantidade de operações em ponto flutuante por segundo computadas pela Unidade de Computação

5.3 Coprocessador FFT *vs* FFTW

A fim de fazer uma avaliação completa dos projetos apresentados nessa dissertação, um último teste foi definido. O objetivo é conhecer o desempenho da execução da transformada de Fourier em FPGA observado por uma aplicação em software que utiliza esse recurso.

A biblioteca FFTW (FRIGO; JOHNSON, 2005) (*Fastest Fourier Transform in the West*) foi escolhida como referência por ser considerada uma das implementações mais rápidas disponíveis, superando inclusive bibliotecas otimizadas manualmente (GU; LI, 2010). Essa biblioteca utiliza diversas técnicas de otimização automáticas, adaptando-se à CPU que executará o código durante o processo de sua compilação (FRIGO; JOHNSON, 2005; GU; LI, 2010).

Esse teste foi organizado da seguinte forma: uma aplicação gera aleatoriamente vetores complexos com 64, 128, 256, 512, 1024, 2048, 4096, 8192 e 16384 pontos. Cada vetor gerado é utilizado para a computação da FFT usando FPGA (função

$fpga\_fft$) e usando a biblioteca FFTW (função $fftwf\_execute$). Para permitir uma comparação com dados estimados do coprocessador FFT, além dos testes citados, foram executados testes com a biblioteca FFTW para vetores com 32768, 65536, 131072 e 262144 pontos. Os tempos de 10 execuções de cada um dos testes foram coletados e calculou-se o tempo médio para cada teste. A Tabela 5.5 apresenta os tempos obtidos e o *speedup* calculado.

Tabela 5.5 - Tempos médios de execução de FFT usando o coprocessador FFT em FPGA e a biblioteca FFTW em CPU

Pontos	Tempo médio (μ s)			*Speedup*	
	FFTW	FPGA	FPGA Est.	FPGA	FPGA Est.
64	2,10	12,80	−−	0,16	−−
128	3,80	14,50	−−	0,26	−−
256	6,90	19,60	−−	0,35	−−
512	13,40	29,30	−−	0,46	−−
1024	31,20	51,90	−−	0,60	−−
2048	71,10	100,70	−−	0,71	−−
4096	144,30	203,60	−−	0,71	−−
8192	285,10	419,30	−−	0,68	−−
16384	917,50	872,10	−−	1,05	−−
32768	2055,00	−−	1868,79	−−	1,10
65536	4576,90	−−	3986,74	−−	1,15
131072	13919,60	−−	8471,83	−−	1,64
262144	48283,90	−−	17940,34	−−	2,69

A primeira coluna da Tabela 5.5 apresenta o número de pontos do vetor, enquanto que a segunda e a terceira colunas apresentam o tempo da execução usando a biblioteca FFTW e o tempo usando o coprocessador FFT em FPGA, respectivamente. A quarta coluna apresenta dados estimados para execução em FPGA. Mais detalhes serão apresentados posteriormente. O *speedup* é calculado através da razão do tempo da execução com a FFTW e a execução em hardware.

Observando os tempos medidos neste teste, verifica-se que a biblioteca FFTW é mais rápida para o cálculo de vetores até 8192 pontos. Para vetores com 16384 elementos, o coprocessador FFT foi mais rápido para o processamento da transformada de Fourier. O gráfico apresentado na Figura 5.4 ilustra esse resultado. A linha azul com quadrados representa os tempos da biblioteca FFTW, enquanto que a curva laranja com losangos apresenta os dados da execução usando o FPGA. A escala vertical desse gráfico é logarítmica.

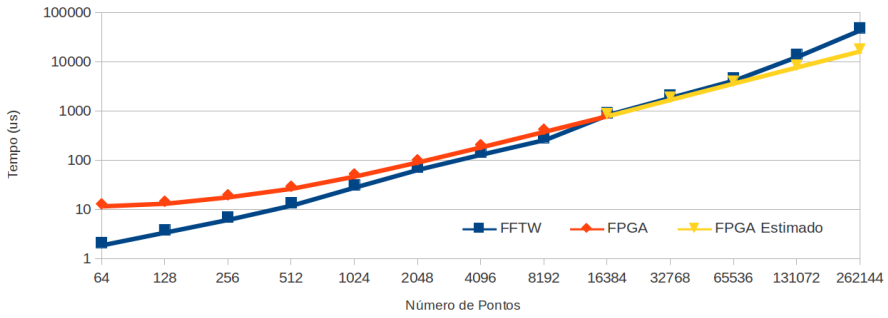


Figura 5.4 - Gráfico com os tempos das execuções da FFT usando a biblioteca FFTW e o coprocessador FFT (eixo vertical em escala logarítmica)

Observa-se que as distâncias entre as curvas vão se aproximando com o aumento do número de pontos, ocorrendo o cruzamento das curvas na proximidade do valor 16384, como sugere os dados da Tabela 5.5. Potencialmente, a arquitetura do coprocessador FFT proposta neste trabalho poderia continuar superando a FFTW para valores maiores que 16384 pontos. Entretanto, devido às limitações quanto ao número recursos no FPGA, não foi possível executar uma FFT maior no dispositivo disponível no Cray XD1. Entretanto, existem disponíveis no mercado famílias de FPGAs mais modernas, como a Virtex 7 da Xilinx, que chegam a ter dezenas de vezes mais recursos que o FPGA utilizado em nosso teste (Virtex 2). Além disso, trabalhos recentes, como o de Tsoi e Luk (2010), apresentam projetos de equipamentos híbridos que utilizam FPGAs mais modernos executando a 400MHz. Segundo os autores, o FPGA Virtex-5 LX330T, utilizado no sistema híbrido apresentado por eles, é capaz de acomodar 300 multiplicadores ou 400 somadores em ponto flutuante simples, volume muito superior aos 10 módulos usados no nosso projeto de coprocessador FFT.

Estes fatos nos motivam em uma tentativa de realizar uma aproximação de desempenho, mesmo que subestimada em relação aos dispositivos modernos, para estimar os tempos de execuções do coprocessador FFT, caso fosse possível o cálculo de vetores com 32768, 65536, 131072 e 262144 pontos. As considerações que são feitas para esta estimação são: (a) o FPGA possui recursos suficientes para acomodar operações com vetores de até 262144 pontos; e (b) o número de operações em ponto flutuante por segundo não piora com o aumento do número de pontos. As demais variáveis, como eficiência de transferência de dados, frequência máxima de execução e número de bancos de memória disponíveis são mantidas.

Para auxiliar o desenvolvimento da estimativa, foi calculado o número de MFLOPS

das execuções utilizando os tempos medidos. A Tabela 5.6 apresenta esses valores, sendo que na primeira coluna estão o número de pontos e na segunda e terceira colunas o número de MFLOPS para os dois tipos de testes executados. O gráfico da Figura 5.5 ilustra esses valores, sendo que a linha azul com quadrados representa os tempos coletados com a biblioteca FFTW e a curva em laranja com losangos os tempos da execução em FPGA.

Tabela 5.6 - Quantidade de operações em ponto flutuante por segundo computadas usando o coprocessador FFT em FPGA e a biblioteca FFTW em CPU

Pontos	MFLOPS		
	FFTW	FPGA	FPGA Est.
64	914,29	150,00	−−
128	1178,95	308,97	−−
256	1484,06	522,45	−−
512	1719,40	786,35	−−
1024	1641,03	986,51	−−
2048	1584,25	1118,57	−−
4096	1703,12	1207,07	−−
8192	1867,70	1269,93	−−
16384	1250,01	1315,08	−−
32768	1195,91	−−	1315,08
65536	1145,51	−−	1315,08
131072	800,39	−−	1315,08
262144	488,63	−−	1315,08

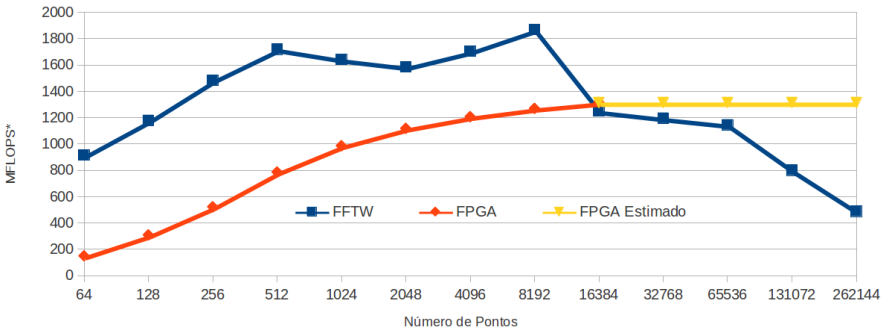


Figura 5.5 - Gráfico com a quantidade de operações em ponto flutuante por segundo das execuções da FFT usando a biblioteca FFTW e o coprocessador FFT

Uma possível observação a ser feita é quanto às oscilações do número de MFLOPS

da execução usando a FFTW. Esse comportamento é recorrente, sendo encontrado em resultados de *benchmarks*[1] executados pelos mantenedores da biblioteca. A curva do número de MFLOPS das execuções que usam FPGA, por outro lado, possui um crescimento suave, no mesmo formato que sugere a curva dos valores calculados de FLOPS (ver Equação 5.6 e Figura 5.3). Esse fato reforça nossa hipótese de não piora no número de MFLOPS considerada nessa estimativa.

Desta forma, foi considerado que o número de MFLOPS para os valores estimados de 32768, 65536, 131072 e 262144 pontos é de 1315,08 MFLOPS, o mesmo valor obtido para vetores com 16384 pontos. Esse valor é apresentado na última coluna da Tabela 5.6 e no gráfico da curva em amarelo da Figura 5.5.

Isolando a variável tempo (T_N) da equação utilizada para o cálculo do número de FLOPS (Equação 5.5), podemos, então, fazer a estimativa usando a equação

$$T_N = \frac{10\left(\frac{N}{2}\log_2(N)\right)}{FLOPS_N}. \tag{5.8}$$

Os tempos estimados foram calculados e estão presentes na quarta coluna da Tabela 5.5. A curva amarela que aparece na Figura 5.4 ilustra esses resultados.

Os valores estimados para a execução da FFT em FPGA superam o desempenho da execução da biblioteca FFTW. O *speedup* calculado, apresentado na última coluna da Tabela 5.4, indica uma potencial de aceleração de até 2,69 vezes.

Reforça-se que os valores apresentados, medidos ou estimados, são para a execução em um FPGA executando a 190MHz frente a uma CPU a 2,4GHz. A diferença de frequência desses recursos é superior a 12 vezes. Os valores obtidos são promissores, uma vez que esses tempos poderiam ser facilmente reduzidos somente com a utilização de FPGAs mais modernos já disponíveis no mercado.

[1] Os resultados desses *benchmarks* podem ser acessados em `http://www.fftw.org/speed/index.html`

6 CONCLUSÕES

Neste trabalho foi apresentada uma arquitetura híbrida para calcular uma solução da classe de *fast Poisson solver* em um sistema híbrido reconfigurável. Em nosso projeto, CPU e FPGA colaboram para a computação desse problema, lidando com os custos de transferência de dados. O núcleo computacional desse *solver* foi migrado para ser processado em um coprocessador especializado baseado em FPGA. Para esse coprocessador, uma arquitetura que implementa o processamento da transformada rápida de Fourier foi proposta. Este projeto usa a técnica de *pipeline* para a computação das operações borboleta e operações em ponto flutuante para ser melhor integrado às aplicações em software. Um esquema de endereçamento para 4 bancos de memória foi projetado para permitir a leitura e gravação concorrente de dois números complexos por ciclo de relógio. A implementação em VHDL da FFT gerou um hardware que consegue operar ao menos a 203MHz, superando a frequência máxima permitida para o FPGA instalado no equipamento utilizado para os testes.

Uma biblioteca em software foi desenvolvida para facilitar a utilização do *Fast Poisson Solver* Híbrido e um conjunto de experimentos foi realizado para avaliar o desempenho da nossa proposta. Na primeira análise foram coletados os tempos de execução e comparados em relação à uma versão implementada para servir de referência. Os resultados apresentam *speedups* médios de 2,45x a 4,64x em favor da versão híbrida.

Um segundo experimento foi realizado para avaliar com detalhes o processamento da transformada de Fourier no coprocessador projetado. Os ciclos de três etapas da computação da FFT foram contados e análises quanto a eficiência na transferência de dados e operações em ponto flutuante por segundo foram realizadas. A análise da eficiência do recebimento dos dados aponta valores acima de 70% para vetores acima de 256 pontos, chegando a 91% quando transferidos 16384 números complexos. O envio dos dados possui um pico da eficiência a partir de 512 pontos, com 97% de eficiência. Após esse valor, a taxa de envio decresce atingindo 60% com 16384 pontos. Esse comportamento está associado ao tamanho do *buffer* utilizado pelo RTCore do Cray XD1.

Uma análise quanto a FLOPS foi feita a partir do equacionamento do número de ciclos necessários para o processamento das operações de borboleta no FPGA. Nossas contas apontaram para um limite teórico máximo de 1,9 GFLOPS para uma execução a 190 MHz. Valores calculados com base nos tamanhos de vetor de entrada, apontam que o coprocessador FFT atingiu 1,89 GFLOPS na execução de uma FFT

em 16384 pontos.

Quanto comparado a execução da biblioteca FFTW, o coprocessador FFT foi mais rápido para execuções com 16384 pontos, sendo inferior nas demais. Valores estimados indicam que existe potencial para que a execução em FPGA seja superior a execução dessa biblioteca em software para valores superiores a 16384.

Quanto ao projeto do *Fast Poisson Solver* Híbrido, as principais contribuições desse trabalho são:

a) Arquitetura híbrida para a solução de uma classe de *fast Poisson solver* em sistema híbrido reconfigurável;

b) Arquitetura de FFT para FPGA integrado em sistema híbrido reconfigurável, onde as transferências de dados são gerenciadas pelo FPGA; e

c) Esquema de endereçamento eficiente para leitura e gravação de dois números complexos por ciclo de relógio em quatro bancos de memória.

Com as atividades realizadas nesse trabalho, pôde-se concluir que implementações em sistemas híbridos reconfiguráveis necessitam de uma abordagem diferenciada em relação aos sistemas de computação tradicionais. Devido a maior flexibilidade na construção de soluções, um entendimento mais profundo do algoritmo a ser implementado é necessário, para que oportunidades específicas possam ser exploradas no sentido de aumentar o desempenho.

6.1 Trabalhos Futuros

Os resultados obtidos nesse trabalho de pesquisa nos motivam a estabelecer uma continuidade. Planejamos avaliar formas de reduzir o uso de memória interna do FPGA pelas unidades de Comunicação e Constantes Trigonométricas para permitir a síntese de hardware para processar FFTs com maiores tamanhos de entrada.

Outros trabalhos estão voltados para a avaliação do consumo de energia necessário para o processamento da FFT em FPGA em relação a outros tipos de recursos, como GPU e CPU. Esse tipo de análise faz parte da tendencia atual de *Green Computing* (LI; ZHOU, 2011), sendo que FPGAs já vem se destacando nessa área para diversas aplicações (THOMAS et al., 2009; BRODTKORB et al., 2010; LEE et al., 2010).

Além disso, temos como objetivo futuro a execução da arquitetura proposta nessa dissertação em sistemas híbridos reconfiguráveis mais modernos para ser possível a comparação com outros recursos computacionais da mesma geração tecnológica. Resultados recentes (TSOI; LUK, 2010) mostram que algumas aplicações em FPGA apresentam melhor desempenho que execuções em GPU ou CPU.

6.2 Outras aplicações de FPGA

Existem diversas aplicações potenciais para serem avaliadas em sistemas híbridos reconfiguráveis, como reconhecimento molecular (KATCHALSKI-KATZIR et al., 1992), aplicações financeiras (MACKIN; WOODS, 2006), apenas para citar algumas. Apesar disso, nessa seção serão brevemente apresentadas duas outras aplicações que foram desenvolvidas durante o período de mestrado e que também estão voltadas para a exploração da arquitetura de alto desempenho híbrida e para conhecer o potencial desse paradigma.

A primeira aplicação foi desenvolvida no contexto de criptografia, através da implementação de funções *Hash* para FPGA (GOMES et al., 2010). Em geral, aplicações de criptografia tem alta afinidade com dispositivos reconfiguráveis, por serem principalmente baseadas em manipulação de bits. Apesar disso, aplicações cotidianas dessa área estão fortemente vinculadas a software e, por isso, são boas candidatas aos sistemas híbridos reconfiguráveis. Neste contexto, foi projetada e implementada um arquitetura híbrida para o processamento de *Hashing SHA-2* com alto *throughput* para o sistema CRAY XD1. A arquitetura é baseada em um sistema de *pipeline* para reduzir a profundidade do circuito e aumentar a frequência de operação. Na solução desenvolvida, múltiplos fluxos de dados podem ser processados paralelamente. Os resultados experimentais apontam aumento da vazão em 1,76 vezes em relação a arquitetura canônica conhecida para essa função. Essa solução, por exemplo, poderia ser utilizada para auxiliar a CPU na criptografia de grandes fluxos de dados ou em aplicações de quebra de senha por força bruta.

A segunda aplicação desenvolvida durante o período de mestrado foi uma Rede Perceptron de Múltiplas Camadas para Sistema Híbrido Reconfigurável (GOMES et al., 2011). Essa solução usa uma abordagem de fluxo contínuo de dados, onde dados chegam ao FPGA através da interface *Fabric Request* e são enviados de volta para a CPU pela interface *User Request*. O projeto desenvolvido é flexível quanto ao número de neurônios das camadas de saída e de entrada, sendo configurados através de um único arquivo, assim como acontece no projeto de *Fast Poisson Solver* Híbrido. O

total de ciclos necessários para processar uma camada da rede neural é dada por

$$Ciclos = (E + 5) + (N + 8) \tag{6.1}$$

onde E é o total de entradas e N é o número de neurônios da camada. As constantes que aparecem nessa equação são referentes ao número de estágios das etapas de processamento. Após o preenchimento do *pipeline* um novo dado sairá por ciclo de relógio desse neuroprocessador para FPGA.

Diversas aplicações podem ser beneficiadas pelo uso de redes neurais artificiais em FPGA em sistemas híbridos. Apenas para citar os trabalhos mais recentes desenvolvidos no Laboratório Associado de Computação e Matemática Aplicada do INPE, temos: Recuperação de perfis de temperatura e umidade da atmosfera a partir de dados de satélite (SHIGUEMORI, 2007); Assimilação de dados com redes neurais artificiais (CINTRA, 2010); Recuperação do perfil de concentração de gases traço atmosféricos a partir de dados de satélites (CARVALHO, 2011); e Redes Neurais Artificiais em imagens para estimação da posição de um VANT (GOLTZ, 2011).

REFERÊNCIAS BIBLIOGRÁFICAS

4DSP Inc. EEE-754 floating point FFT/IFFT IP core. Disponível em <http://www.4dsp.com/fft.htm>. Acesso em: 10 abr. 2012. 17

Altera Corp. Implementing FPGA Desing with the OpenCL Standard. Disponível em <http://www.altera.com/b/opencl.html>. Acesso em: 25 abr. 2012. nov. 2011. 8

Altera Corporation. FFT megacore function. Disponível em <http://www.altera.com/products/ip/dsp/transforms/m-ham-fft.html>. Acesso em: 12 mai. 2012. 17

ANDREWS, D.; NIEHAUS, D.; JIDIN, R.; FINLEY, M.; PECK, W.; FRISBIE, M.; ORTIZ, J.; KOMP, E.; ASHENDEN, P. Programming models for hybrid FPGA-CPU computational components: a missing link. **IEEE Micro**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 24, n. 4, p. 42–53, 2004. ISSN 0272-1732. 2

BANERJEE, S.; KESH, A.; S.THAKKAR, C.; GUPTA, R.; SETH, S. S.; ANISH, T. Implementation of fast fourier transform (FFT) on FPGA using verilog HDL. Disponível em <www.stanford.edu/ sidseth/files/fft_report.pdf>. Acesso em: 06 jan. 2012. 2005. 17, 26

BONDHUGULA, U.; DEVULAPALLI, A.; DINAN, J.; FERNANDO, J.; WYCKOFF, P.; STAHLBERG, E.; SADAYAPPAN, P. Hardware/software integration for FPGA-based all-pairs shortest-paths. In: 14TH ANNUAL IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, 2006, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2006. p. 152–164. ISBN 0-7695-2661-6. 2

BRODTKORB, A. R.; DYKEN, C.; HAGEN, T. R.; HJELMERVIK, J. M.; STORAASLI, O. O. State-of-the-art in heterogeneous computing. **Sci. Program.**, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 18, p. 1–33, January 2010. ISSN 1058-9244. 2, 3, 9, 70

BUZBEE, B. L. A fast poisson solver amenable to parallel computation. **Computers, IEEE Transactions on**, C-22, n. 8, p. 793 –796, aug. 1973. ISSN 0018-9340. 12

CAMPBELL, S.; MEYER, C. **Generalized Inverses of Linear Transformations**. [S.l.]: Society for Industrial and Applied Mathematics, 2009. (Classics in Applied Mathematics). ISBN 9780898716719. 51

CAMPOS VELHO, H. F. de; CLAEYSSEN, J. C. R. Singular value decomposition in the numerical integration of an atmospheric model. In: XIII CONGRESSO ÍBERO-LATINO-AMERICANO DE MéTODOS COMPUTACIONAIS PARA A ENGENHARIA, 1992, Porto Alegre, RS, Brasil. **Anais...** Porto Alegre, RS, Brasil: CILAMCE 1992, 1992. p. 344–353. 3, 11, 51, 52, 53

CARDOSO, J. a. M. P.; DINIZ, P. C.; WEINHARDT, M. Compiling for reconfigurable computing: a survey. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 42, p. 13:1–13:65, June 2010. ISSN 0360-0300. 2, 5, 8, 9, 22, 40, 44

CARVALHO, A. R. **Uso de redes neurais otimizadas para recuperação do perfil de concentração de gases traço atmosféricos a partir de dados de satélites**. Disponível em <http://urlib.net/8JMKD3MGP7W/39R454E>. Acesso em: 10 fev. 2012. Tese (Doutorado) — Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2011. 72

CHAMBERLAIN, R. D.; FRANKLIN, M. A.; TYSON, E. J.; BUHLER, J.; GAYEN, S.; CROWLEY, P.; BUCKLEY, J. H. Application development on hybrid systems. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2007, Reno, Nevada. **Proceedings...** New York, NY, USA: ACM, 2007. (SC '07), p. 50:1–50:10. ISBN 978-1-59593-764-3. 2, 5

CHAMBERLAIN, R. D.; LANCASTER, J. M.; CYTRON, R. K. Visions for application development on hybrid computing systems. **Parallel Computing**, v. 34, n. 4-5, p. 201 – 216, 2008. ISSN 0167-8191. Reconfigurable Systems Summer Institute 2007. 2, 5, 9

CHAVARRÍA-MIRANDA, D.; NIEPLOCHA, J.; GORTON, I. Hardware-accelerated components for hybrid computing systems. In: 2008 COMPFRAME/HPC-GECO WORKSHOP ON COMPONENT BASED HIGH PERFORMANCE, 2008, Karlsruhe, Germany. **Proceedings...** New York, NY, USA: ACM, 2008. p. 1–8. ISBN 978-1-60558-311-2. 2

CINTRA, R. S. C. **Assimilação de dados com redes neurais artificiais em modelo de circulação geral da atmosfera**. Disponível em <http://urlib.net/8JMKD3MGP7W/389UKPS>. Acesso em: 12 dez. 2011. Tese (Doutorado) — Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2010. 72

CIPRA, B. A. The best of the 20th century: editors name top 10 algorithms. **SIAM News**, Society ofr Industrial and Applied Mathematics, v. 33, n. 4, 2000. 12

COLONY, R.; REYNOLDS, R. R. An application of hockney's method for solving poisson's equation. In: SPRING JOINT COMPUTER CONFERENCE, 1970, Atlantic City, New Jersey. **Proceedings...** New York, NY, USA: ACM, 1970. (AFIPS '70 (Spring)), p. 409–416. Disponível em: <http://doi.acm.org/10.1145/1476936.1476999>. 14

COMISSÃO REGIONAL DE ALTO DESEMPENHO - RS (CRAD-RS). **Caderno dos cursos permanentes.** Porto Alegre, 2006. 1

Convey Computer Corp. **Personality development kit (PDK) for Convey hybrid-core computers**. Texas, USA: Convey Computer Corporation, 2011. Disponível em <http://www.conveycomputer.com/Resources/PersonalityDevelopmentKit.pdf>. Acesso em: 24 jan. 2011. 10

COOLEY, J. W.; TUKEY, J. W. An algorithm for the machine calculation of complex fourier series. **Mathematics of Computation**, v. 19, n. 90, p. 297–301, 1965. 12, 13

CORMEN, T. H.; STEIN, C.; RIVEST, R. L.; LEISERSON, C. E. **Introduction to Algorithms**. 2nd. ed. [S.l.]: McGraw-Hill Higher Education, 2001. ISBN 0070131511. 12, 13

CRAVEN, S.; ATHANAS, P. Examining the viability of FPGA supercomputing. **EURASIP J. Embedded Syst.**, Hindawi Publishing Corp., New York, NY, United States, v. 2007, n. 1, p. 13–13, 2007. ISSN 1687-3955. 2

Cray Inc. **CRAY XD1 datasheet**. Mendota Heights, MN, USA: Cray Inc., 2005. 21, 22

_____. **Cray XD1 FPGA development**. Mendota, MN, USA: Cray Inc., 2005. 46

_____. **Cray XD1 FPGA programming**. Mendota, MN, USA: Cray Inc., 2005. 10, 44

_____. **Design of cray XD1 rapidArray transport core**. Mendota, MN, USA: Cray Inc., 2005. 3–7 p. 23, 24, 25, 34, 60, 62

DAVIDS, P. J. **Circulant matrices**. New York, NY, USA: John Wiley & Sons, 1979. 52, 53

Dillon Engineering, Inc. Fast Fourier Transform (FFT) IP Cores for FPGA and ASIC. Disponível em <http://www.dilloneng.com/fft_ip>. Acesso em: 10 abr. 2012. 17

EL-GHAZAWI, T.; EL-ARABY, E.; HUANG, M.; GAJ, K.; KINDRATENKO, V.; BUELL, D. The promise of high-performance reconfigurable computing. **Computer**, IEEE Computer Society, Los Alamitos, CA, USA, v. 41, p. 69–76, 2008. ISSN 0018-9162. 2

FERNANDO, J.; DALESSANDRO, D.; DEVULAPALLI, A.; WOHLEVER, K. Accelerated FPGA Based Encryption. In: CUG, 2005, New Mexico, USA. **Proceedings...** New Mexico, USA: The Cray User Group, 2005. 2

FRIGO, M.; JOHNSON, S. G. The design and implementation of FFTW3. **Proceedings of the IEEE**, v. 93, n. 2, p. 216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation". 64

_____. FFTw Benchmark Results. Disponível em <http://www.fftw.org/speed/>. Acesso em: 18 abr. 2012. 2012. 64

GARCIA, P.; COMPTON, K.; SCHULTE, M.; BLEM, E.; FU, W. An overview of reconfigurable hardware in embedded systems. **EURASIP J. Embedded Syst.**, Hindawi Publishing Corp., New York, NY, United States, v. 2006, p. 13–13, January 2006. ISSN 1687-3955. 2

GIRAUD, L.; GUIVARCH, R.; STEIN, J. A parallel distributed fast 3d poisson solver for méso-nh. In: 5TH INTERNATIONAL EURO-PAR CONFERENCE ON PARALLEL PROCESSING, 1999, London, UK, UK. **Proceedings...** London, UK, UK: Springer-Verlag, 1999. (Euro-Par '99), p. 1431–1434. ISBN 3-540-66443-2. Disponível em: <http://dl.acm.org/citation.cfm?id=646664.700908>. 11, 12

GOKHALE, M.; GRAHAM, P. S. **Reconfigurable computing: accelerating computation with field-programmable gate arrays**. Dordrecht, The Netherlands: Springer, 2005. ISBN 0387261052. 2, 5, 6, 7, 9

GOLTZ, G. A. M. **Redes Neurais Artificiais em imagens para estimação da posição de um VANT**. Disponível em <http://urlib.net/8JMKD3MGP7W/39EBJ7S>. Acesso em: 10 fev. 2012.

Dissertação (Mestrado) — Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2011. 72

GOLUB, G. H.; HUANG, L. C.; SIMON, H.; TANG, W. pai. A fast poisson solver for the finite difference solution of the incompressible navier-stokes equations. **SIAM J. SCI. COMPUT**, v. 19, n. 5, p. 1606–1624, 1998. 14, 15

GOMES, V. C. F.; CHARãO, A. S.; CAMPOS VELHO, H. F. Avaliação de abordagens de comunicação com FPGA no supercomputador cray XD1. In: IX ESCOLA REGIONAL DE ALTO DESEMPENHO, 2009, Caxias do Sul, RS, Brasil. **Anais...** Caxias do Sul, RS, Brasil: SBC, 2009. 22

_____. Uma FFT híbrida e paralela para o supercomputador cray XD1. In: X SIMPóSIO EM SISTEMAS COMPUTACIONAIS, 2009, São Paulo, SP, Brasil. **Anais...** [S.l.], 2009. 2

GOMES, V. C. F.; CHARãO, A. S.; CAMPOS VELHO, H. F.; CALEGARO, B. Uma arquitetura de hashing SHA-2 com alto throughput para sistema híbrido reconfigurável. In: X ESCOLA REGIONAL DE ALTO DESEMPENHO, 2010, Passo Fundo, RS, Brasil. **Anais...** Passo Fundo, RS, Brasil: SBC, 2010. 71

GOMES, V. C. F.; SHIGUEMORI, E. H.; CHARÃO, A.; CAMPOS VELHO, H. F. Rede perceptron de múltiplas camadas para sistema híbrido reconfigurável. In: CASTRO, A. P. A. d.; DRUMMOND, I. N.; SANDRI, S. A. (Ed.). **Anais...** Instituto Nacional de Pesquisas Espaciais (INPE), 2011. Disponível em: <http://urlib.net/sid.inpe.br/mtc-m18/2011/10.18.00.23>. 71

GONZALEZ-CONCEJERO, C.; RODELLAR, V.; ALVAREZ-MARQUINA, A.; ICAYA, E. M. d.; GOMEZ-VILDA, P. An fft/ifft design versus altera and xilinx cores. In: 2008 INTERNATIONAL CONFERENCE ON RECONFIGURABLE COMPUTING AND FPGAS, 2008. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2008. p. 337–342. ISBN 978-0-7695-3474-9. 17

GU, L.; LI, X. Dft performance prediction in fftw. In: 22ND INTERNATIONAL CONFERENCE ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING, 2010, Newark, DE. **Proceedings...** Berlin, Heidelberg: Springer-Verlag, 2010. (LCPC'09), p. 140–156. ISBN 3-642-13373-8, 978-3-642-13373-2. Disponível em: <http://dx.doi.org/10.1007/978-3-642-13374-9_10>. 64

HARTENSTEIN, R. A decade of reconfigurable computing: a visionary retrospective. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN

EUROPE. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2001. (DATE '01), p. 642–649. ISBN 0-7695-0993-2. 2

HAUCK, S.; DEHON, A. **Reconfigurable computing: the theory and practice of FPGA-based computation (systems on silicon)**. Burlington, MA, USA: Morgan Kaufmann, 2007. Hardcover. ISBN 0123705223. 2, 6, 7, 8, 9

HE, C. **Numerical Solutions of Differential equations on FPGA-Enhanced Computers**. Disponível em <http://hdl.handle.net/1969.1/ETD-TAMU-1248>. Acesso em: 06 jan. 2012. Tese (Doutorado) — Texas A&M University, 2007. 2, 16

HE, H.; GUO, H. The realization of FFT algorithm based on FPGA co-processor. **Intelligent Information Technology Applications, 2007 Workshop on**, IEEE Computer Society, Los Alamitos, CA, USA, v. 3, p. 239–243, 2008. 2, 17

HEMMERT, K. S.; UNDERWOOD, K. D. An analysis of the double-precision floating-point fft on fpgas. **Field-Programmable Custom Computing Machines, Annual IEEE Symposium on**, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 171–180, 2005. 17

HOCKNEY, R. W. A fast direct solution of poisson's equation using fourier analysis. **J. ACM**, ACM, New York, NY, USA, v. 12, n. 1, p. 95–113, jan. 1965. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/321250.321259>. 12

HOFFMAN, J. D. **Numerical methods for engineers and scientists**. 2. ed. New York, NY, USA: Marcel Dekker, Inc, 2001. 3, 11

Intel Corporation. Intel Fortran Compiler for Linux. Disponível em <http://software.intel.com/en-us/articles/intel-fortran-compiler-for-linux-9x-manuals/>. Acesso em: 12 mai. 2012.
53

JACOBSON, M. Z. Weather, Climate, and Air Pollution. In: _____. **Fundamentals of atmospheric modeling**. Cambridge, UK: Cambridge University Press, 2005. cap. 1.4, p. 6–8. ISBN 0521548659. 3

JASKELAINEN, P. O.; LAMA, C. S. de L.; HUERTA, P.; TAKALA, J. H. OpenCL-based design methodology for application-specific processors. In: INTERNATIONAL CONFERENCE ON EMBEDDED COMPUTER SYSTEMS (SAMOS), 2010. **Proceedings...** 2010. p. 223–230. Disponível em: <http://dx.doi.org/10.1109/ICSAMOS.2010.5642061>. 8

JONES, D.; POWELL, A.; BOUGANIS, C.-S.; CHEUNG, P. GPU Versus FPGA for High Productivity Computing. In: INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 2010. **Proceedings...** [S.l.], 2010. p. 119 –124. ISSN 1946-1488. 2

KATCHALSKI-KATZIR, E.; SHARIV, I.; EISENSTEIN, M.; FRIESEM, A. A.; AFLALO, C.; VAKSER, I. A. Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques. **Proceedings of the National Academy of Sciences of the United States of America**, Department of Membrane Research and Biophysics, Weizmann Institute of Science, Rehovot, Israel., v. 89, n. 6, p. 2195–2199, mar. 1992. ISSN 0027-8424. Disponível em: <http://dx.doi.org/10.1073/pnas.89.6.2195>. 12, 71

KINDRATENKO, V.; POINTER, D. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In: 14TH ANNUAL IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, 2006, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2006. p. 13–22. ISBN 0-7695-2661-6. 2

KINDRATENKO, V. V.; BRUNNER, R. J.; MYERS, A. D. Mitrion-C application development on SGI altix 350/RC100. In: 15TH ANNUAL IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, 2007, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2007. p. 239–250. ISBN 0-7695-2940-2. 10

KINDRATENKO, V. V.; STEFFEN, C. P.; BRUNNER, R. J. Accelerating scientific applications with reconfigurable computing: getting started. **Computing in Science and Engg.**, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 9, n. 5, p. 70–77, 2007. ISSN 1521-9615. 2

KNITTEL, G. A CG-based Poisson solver on a GPU-cluster. In: INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING (HIPC), 2010. **Proceedings...** [S.l.], 2010. p. 1 –10. 16

KOEHLER, S.; CURRERI, J.; GEORGE, A. D. Performance analysis challenges and framework for high-performance reconfigurable computing. **Parallel Comput.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 34, n. 4-5, p. 217–230, 2008. ISSN 0167-8191. 2, 9

LEE, J.; SUN, J.; PETERSON, G. D.; HARRISON, R.; HINDE, R. Power-aware performance of mixed precision linear solvers for fpgas and gpgpus. In: SYMPOSIUM ON APPLICATION ACCELERATORS FOR HIGH PERFORMANCE COMPUTING, 2010. **Proceedings...** [S.l.], 2010. 2, 70

LI, Q.; ZHOU, M. The survey and future evolution of green computing. In: IEEE/ACM INTERNATIONAL CONFERENCE ON GREEN COMPUTING AND COMMUNICATIONS, 2011, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2011. (GREENCOM '11), p. 230–233. ISBN 978-0-7695-4466-3. Disponível em: <http://dx.doi.org/10.1109/GreenCom.2011.47>. 70

LYNCH, P. **DYNAMO—A one dimensional primitive equation model**. Dublin, Irlanda: Irish Meteorolgical Service, 1984. 49, 51

_____. The origins of computer weather prediction and climate modeling. **J. Comput. Phys.**, Academic Press Professional, Inc., San Diego, CA, USA, v. 227, n. 7, p. 3431–3444, 2008. ISSN 0021-9991. 1

MACKIN, B.; WOODS, N. FPGA Acceleration in HPC: A Case Study in Financial Analytics. Disponível em <http://oldwww.xtremedatainc.com/pdf/FPGA_Acceleration_in_HPC.pdf>. Acesso em: 12 mar. 2012. 2006. 71

MENOTTI, R. **LALP : uma linguagem para exploração do paralelismo de loops em computação reconfigurável**. Disponível em <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-17082010-151100/>. Acesso em: 14 abr. 2012. Tese (Doutorado) — Universidade de São Paulo, São Carlos, SP, 2010. 7, 8

OWAIDA, M.; BELLAS, N.; ANTONOPOULOS, C. D.; DALOUKAS, K.; ANTONIADIS, C. Massively parallel programming models used as hardware description languages: the opencl case. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 2011, San Jose, California, USA. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2011. (ICCAD '11), p. 326–333. ISBN 978-1-4577-1398-9. Disponível em: <http://dl.acm.org/citation.cfm?id=2132325.2132409>. 8

PALMER, J. M. **The Hybrid Architecture Parallel Fast Fourier Transform (HAPFFT)**. Dissertação (Mestrado) — Brigham Young University, 2005. 17

PAPAKONSTANTINOU, A.; GURURAJ, K.; STRATTON, J. A.; CHEN, D.; CONG, J.; HWU, W.-M. W. High-performance CUDA kernel execution on FPGAs. In: 23RD INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 2009, Yorktown Heights, NY, USA. **Proceedings...** New York, NY, USA: ACM, 2009. (ICS '09), p. 515–516. ISBN 978-1-60558-498-0. 2, 5

Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; Flannery, B. P. **Numerical recipes in C. The art of scientific computing**. Second. [S.l.]: Cambridge: University Press, 1992. 12, 13, 14

QIAN, H.; SAPATNEKAR, S. S. Fast poisson solvers for thermal analysis. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 2010, San Jose, California. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2010. (ICCAD '10), p. 698–702. ISBN 978-1-4244-8192-7. Disponível em: <http://dl.acm.org/citation.cfm?id=2133429.2133577>. 12, 14, 15

RAMACHANDRAN, L.; GAJSKI, D.; CHAIYAKUL, V. An algorithm for array variable clustering. In: EUROASIC, THE EUROPEAN EVENT IN ASIC DESIGN, 1994. **Proceedings...** [S.l.], 1994. p. 262 –266. 40

SCHMIT, H.; THOMAS, D. Synthesis of application-specific memory designs. **Very Large Scale Integration (VLSI) Systems, IEEE Transactions on**, v. 5, n. 1, p. 101 –111, march 1997. ISSN 1063-8210. 40

SHAN, A. Heterogeneous processing: a strategy for augmenting Moore's law. Disponível em <http://www.linuxjournal.com/article/8368>. Acesso em: 06 jan. 2011. 2006. 1, 2, 5

SHI, J.; CAI, Y.; HOU, W.; MA, L.; TAN, S. X.-D.; HO, P.-H.; WANG, X. Gpu friendly fast poisson solver for structured power grid network analysis. In: **Proceedings of the 46th Annual Design Automation Conference**. New York, NY, USA: ACM, 2009. (DAC '09), p. 178–183. ISBN 978-1-60558-497-3. Disponível em: <http://doi.acm.org/10.1145/1629911.1629961>. 16

SHIGUEMORI Élcio H. **Recuperação de perfis de temperatura e umidade da atmosfera a partir de dados de satélite - abordagens por redes neurais artificiais e implementação em hardware**. Disponível em <http://urlib.net/8JMKD3MGP7W/389UKPS>. Acesso em: 23 jan. 2011. Tese (Doutorado) — Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2007. 2, 8, 72

TANENBAUM, A. S. **Modern Operating Systems**. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633. 39

THOMAS, D. B.; HOWES, L.; LUK, W. A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. In: ACM/SIGDA INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, 2009, Monterey, California, USA. **Proceedings...** New York, NY, USA: ACM, 2009. (FPGA '09), p. 63–72. ISBN 978-1-60558-410-2. Disponível em: <http://doi.acm.org/10.1145/1508128.1508139>. 2, 70

Top500.org. **TOP500 list - november 2011**. Novembro 2011. Disponível em <http://www.top500.org/list/2011/11/100>. Acesso em: 06 mar. 2012. 2

TSOI, K. H.; LUK, W. Axel: a heterogeneous cluster with fpgas and gpus. In: 18th annual ACM/SIGDA international symposium on Field programmable gate arrays, 2010, Monterey, California, USA. **Proceedings...** New York, NY, USA: ACM, 2010. (FPGA '10), p. 115–124. ISBN 978-1-60558-911-4. Disponível em: <http://doi.acm.org/10.1145/1723112.1723134>. 2, 10, 16, 66, 71

VITE-FRIAS, J. A.; ROMERO-TRONCOSO, R. d. J.; ORDAZ-MORENO, A. VHDL core for 1024-point radix-4 FFT computation. In: 2005 INTERNATIONAL CONFERENCE ON RECONFIGURABLE COMPUTING AND FPGAS (RECONFIG'05) ON RECONFIGURABLE COMPUTING AND FPGAS, 2005, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2005. p. 24. ISBN 0-7695-2456-7. 2, 17

WAIN, R.; AL. et. **An overview of FPGAs and FPGA programming; initial experiences at Daresbury**. Daresbury, Cheshire, UK: Council for the Central Laboratory of the Research Councils, 2004. 2–4 p. 2

WILKINSON, B.; ALLEN, M. **Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2004. ISBN 0131405632. 1

Xilinx Inc. Fast fourier transform. Disponível em <http://www.xilinx.com/products/intellectual-property/FFT.htm>. Acesso em: 10 abr. 2012. 17

YANG, J.; CAI, Y.; ZHOU, Q.; SHI, J. Fast poisson solver preconditioned method for robust power grid analysis. In: INTERNATIONAL CONFERENCE ON

COMPUTER-AIDED DESIGN, 2011, San Jose, California. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2011. (ICCAD '11), p. 531–536. ISBN 978-1-4577-1398-9. Disponível em: <http://dl.acm.org/citation.cfm?id=2132325.2132449>. 11, 12, 14, 15

ZHUO, L.; PRASANNA, V. K. High performance linear algebra operations on reconfigurable systems. In: 2005 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2005, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2005. p. 2. ISBN 1-59593-061-2. 2

_____. Scalable hybrid designs for linear algebra on reconfigurable computing systems. **IEEE Trans. Comput.**, IEEE Computer Society, Washington, DC, USA, v. 57, n. 12, p. 1661–1675, 2008. ISSN 0018-9340. 2