



Pós-Graduação em Ciência da Computação

PARTICIONAMENTO E ESCALONAMENTO DE MATRIZES DENSAS PARA PROCESSAMENTO EM HARDWARE RECONFIGURÁVEL

Por

Derci de Oliveira Lima

Dissertação de Mestrado



Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, SETEMBRO / 2009

[PAGINA INTENCIONALMENTE DEIXADA EM BRANCO]



UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DERCI DE OLIVEIRA LIMA

**“Particionamento e Escalonamento de Matrizes Densas para
Processamento em Hardware Reconfigurável”**

*ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM
CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA
UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO
PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA
COMPUTAÇÃO.*

ORIENTADOR: Manoel Eusébio de Lima

RECIFE, SETEMBRO / 2009

Lima, Derci de Oliveira

Particionamento e escalonamento de matrizes densas para processamento em hardware reconfigurável / Derci de Oliveira Lima. - Recife: O Autor, 2009.

xxii, 106 p. : il., fig., tab.

Dissertação (mestrado) – Universidade Federal de Pernambuco. Cln. Ciência da Computação, 2009.

Inclui bibliografia, glossário e apêndices.

1. Engenharia da computação. 2. Co-processadores em FPGA. 3. Computação de alto desempenho. 4. Multiplicação de matrizes. I. Título.

621.39

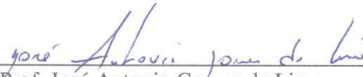
CDD (22. ed.)

MEI2010 – 073

Dissertação de Mestrado apresentada por **Derci de Oliveira Lima** Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **“Particionamento e Escalonamento de Matrizes Densas Para Processamento Em Hardware Reconfigurável”**, orientado pelo Prof. Manoel Eusébio de Lima e aprovada pela Banca Examinadora formada pelos professores:



Prof. Manoel Eusébio de Lima
Centro de Informática / UFPE



Prof. José Antonio Gomes de Lima
Departamento de Informática / UFPB



Prof. Paulo Sérgio Brandão do Nascimento
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 11 de setembro de 2009.



Prof. Nelson Souto Rosa
Vice-Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

[PAGINA INTENCIONALMENTE DEIXADA EM BRANCO]

Dedicatória

Em Memória

*Dedico este trabalho à minha mãe,
Ester de Oliveira Lima,
que sempre lutou para ter seus filhos
criados dentro da honra que deve ter
um ser humano.*

[PAGINA INTENCIONALMENTE DEIXADA EM BRANCO]

Agradecimentos

Agradeço primeiramente a **DEUS**, que, na sua misericórdia infinita, permitiu que eu pudesse chegar até aqui.

Ao meu pai, **José Lima Expedito**, que, com seus sábios conselhos, ajudou-me a perseverar no caminho traçado.

A minha esposa, **Maria do Carmo**, que esteve comigo, por todo este longo caminho, me incentivando e aturando as minhas longas ausências, mesmo estando presente.

Aos meus filhos, **Paulo Henrique** e **Márcia Cristina**, pelo orgulho “*danado*” que eles têm de seu pai.

Ao professor e orientador, **Manoel Eusébio de Lima**, pela oportunidade que me foi dada em participar do seu grupo de pesquisa e desenvolvimento.

Aos **colegas** do grupo de pesquisa **HPCIN**, em especial a **Victor Medeiros** e **Viviane Lucy** pela grande ajuda na reta final deste trabalho.

A todos os **colegas** que durante esta caminhada participaram, em maior ou menor grau, da minha formação acadêmica.

A todas as **pessoas** que fazem o **CIN – Centro de Informática da Universidade Federal de Pernambuco**, pela dedicação, carinho e paciência que demonstraram durante minha passagem pela instituição.

Deixo aqui registrado o meu **MUITO OBRIGADO** e que **DEUS** abençoe a todos.

[PAGINA INTENCIONALMENTE DEIXADA EM BRANCO]

Resumo

A solução de problemas complexos em várias áreas do conhecimento humano, tais como: análise de investimento no setor bancário, análise e visualização de imagens médicas em tempo real, indústria de óleo e gás, etc. que utilizam muitas vezes algoritmos complexos e/ou uma grande massa de dados, têm requerido cada vez mais sistemas computacionais de alto desempenho para seu processamento.

Estes aplicativos, em sua maioria, devido a sua grande massa de dados, grandes laços de processamento em seus procedimentos, podem consumir dias ou até meses de trabalho, em computadores de processamento seqüencial, para apresentar o resultado final. Existem casos em que este tempo excessivo pode inviabilizar um projeto em questão, por perder o *time to market* de um produto.

Diferentes tecnologias e estruturas de dados têm sido sugeridas para lidar com tais problemas, visando uma melhor customização, tentando retirar o melhor da arquitetura e do sistema, seja em termos de software como de hardware. Dentre estas arquiteturas hw/sw, optamos neste trabalho ao estudo de uma solução baseada em FPGAs (Field Programmable Gate Arrays) como um co-processador. O uso deste dispositivo permite uma nova abordagem do problema. Agora, um determinado aplicativo poderia ser particionado em duas partes: a primeira, aquela com características de controle, processo seqüencial, continuaria sendo executado no host com processadores genéricos, enquanto que a parte com os grandes laços de processamento seriam processados, com maior desempenho por explorar o paralelismo, nos co-processadores com FPGAs.

Porém, a movimentação dos dados entre a memória principal do host e a memória externa do FPGA é considerada um grande gargalo para o processamento em hardware. Vários autores em seus trabalhos demonstram o desempenho alcançado com o uso de processamento em hardware, mas consideram que os dados já estão na memória externa do FPGA. Poucos comentam sobre a perda de desempenho quando se considera a movimentação de dados.

Neste trabalho foram estudadas técnicas de particionamento de grandes matrizes densas, reuso de dados e as estratégias que melhor se adéquam para algumas arquiteturas estudadas neste trabalho. As latências desta movimentação de dados entre o host e o co-processador em FPGA foram o foco deste trabalho também. Concluimos com um estudo de caso onde propomos uma estratégia para particionamento e multiplicação de matrizes por

blocos no FPGA virtex 5 (XC5VLX50T -1 FF1136), montado em uma placa (ML 555 Board) da Xilinx.

Palavras-chaves: FPGA, Particionamento, Movimentação de Dados, Latências.

Abstract

The solution of complex problems in various areas of human knowledge, such as analysis of investment in the banking, analysis and visualization of medical images in real time, oil and gas industry, etc. which often use complex algorithms and/or a large body of data, are increasingly required for high performance computing systems for their processing.

These applications, mostly due to its large mass of data, large links in their processing procedures, may take days or even months of work on computers of sequential processing to present the final result. There are cases where the excessive time can prevent a project in question, by losing the time to market of a product.

Different technologies and data structures have been suggested to deal with such problems, aiming a better customization, trying to get the best of architecture and system, be it in terms of software as of hardware. Among these architectures hardware/software, chose this work to study a solution based on *FPGAs* (Field Programmable Gate Arrays) as a co-processor. Using this device allows a new approach to the problem. Now, a specific application could be partitioned into two parts: first, that with the character of control, sequential process, would still be running on a host with generic processors, while the part with the major processing ties would be processed with higher performance by exploiting the parallelism, in co-processors in *FPGAs*.

However, the movement of data between the main memory and the memory of the host's external *FPGA* is considered a major bottleneck for the processing in hardware. Several authors in their work demonstrate the performance achieved with the use of processing in hardware, but they consider that the data is already in the *FPGA* external memory. Few commented on the loss of performance when considering the handling of data.

In this work we studied techniques for partitioning of large dense matrices, reuse of data and the strategies that best fit for some architecture studied in this work. The latencies of the movement of data between the host and co-processor on *FPGA* have also been the focus of this work. We conclude with a case study where we propose a strategy for partitioning and matrix multiplication of blocks in *FPGA Virtex 5* (XC5VLX50T -1 FF1136), mounted on a plate (ML 555 Board) of Xilinx.

Keywords: FPGA, Partitioning, Data Handling, latencies.

[PAGINA INTENCIONALMENTE DEIXADA EM BRANCO]

Sumário

DEDICATÓRIA	VII
AGRADECIMENTOS	IX
RESUMO	XI
ABSTRACT	XIII
SUMÁRIO	XV
RELAÇÃO DE FIGURAS.....	XVII
RELAÇÃO DE TABELAS.....	XIX
GLOSSÁRIO	XXI
CAPÍTULO 1 - INTRODUÇÃO	25
1.1 CONTEXTUALIZAÇÃO	25
1.2 DESAFIOS A VENCER	27
1.3 OBJETIVO	27
1.4 ORGANIZAÇÃO DO TRABALHO.....	28
CAPÍTULO 2 – TRABALHOS RELACIONADOS	29
2.1 MULTIPLICAÇÃO DE MATRIZ EM PONTO FLUTUANTE DE 64 BIT EM FPGA	29
2.2 PARTICIONAMENTO DE CÓDIGO PARA COMPUTAÇÃO RECONFIGURÁVEL DE ALTO DESEMPENHO.....	32
2.3 ALGORITMO MODULAR E ESCALÁVEL PARA MULTIPLICAÇÃO DE MATRIZ DE PONTO FLUTUANTE EM SISTEMAS DE COMPUTAÇÃO RECONFIGURÁVEL.....	36
2.4 METODOLOGIA PARA UTILIZAÇÃO EFICIENTE DOS RECURSOS DAS NOVAS GERAÇÕES DE FPGA.	39
2.5 CONCLUSÃO	41
CAPÍTULO 3 – FUNDAMENTAÇÃO TEÓRICA	43
3.1 MULTIPLICAÇÃO DE MATRIZES.....	43
3.2 PARTICIONAMENTO E ESCALONAMENTO DOS DADOS	44
3.2.1 <i>Particionamento em Blocos</i>	45
3.2.2 <i>Granularidade de Blocos</i>	47
3.3 CONCLUSÃO.....	49
CAPÍTULO 4 – PLATAFORMAS HÍBRIDAS RECONFIGURÁVEIS	51
4.1 PLATAFORMA SRC MAPSTATION	52
4.1.1 <i>Usando Dispositivo FPGA para Acelerar Simulação Biomolecular</i>	54
4.2 PLATAFORMA CRAY XD1.....	55
4.2.1 <i>Scalable Hybrid Designs for Linear Algebra on Reconfigurable Computing Systems</i>	58
4.3 CONCLUSÃO.....	59
CAPÍTULO 5 – AMBIENTE DE DESENVOLVIMENTO RASC	61
5.1 VISÃO GERAL DO SISTEMA RASC.....	61
5.2 CORE SERVICES.....	64

5.3	FLUXO DE PROJETO NA PLATAFORMA RASC.....	67
5.4	CAMADA DE ABSTRAÇÃO DO RASC	69
5.5	AMBIENTE DE SIMULAÇÃO VCS/VIRSIM	70
5.6	AValiação da metodologia MITRION-C em computação reconfigurável de alto desempenho.	71
5.7	CONCLUSÃO.....	74
CAPÍTULO 6 - ESTRATÉGIAS PARA PARTICIONAMENTO DE GRANDES MATRIZES		75
6.1	METODOLOGIA DE DESENVOLVIMENTO	75
6.2	COMPUTAÇÃO INTENSIVA	76
6.3	ALGORITMO DE PARTICIONAMENTO DOS DADOS.....	78
6.4	FLUXOGRAMA DO ESCALONAMENTO DOS BLOCOS PARA MULTIPLICAÇÃO EM HARDWARE.....	82
6.5	ARQUITETURA DE MULTIPLICAÇÃO DE MATRIZES PARA A PLATAFORMA RASC	84
6.6	CONCLUSÃO.....	85
CAPÍTULO 7 – ESTUDO DE CASO.....		87
7.1	DESCRIÇÕES DAS PLATAFORMAS UTILIZADAS NAS ESTIMATIVAS.....	87
7.2	ESTIMATIVA DO CUSTO DA MOVIMENTAÇÃO DE DADOS.....	89
7.3	ESTIMATIVA DO CUSTO DE PROCESSAMENTO.....	90
7.4	ANÁLISE DO DESEMPENHO DAS PLATAFORMAS ML 555, RASC E CRAY	91
7.5	CONCLUSÃO.....	95
CAPÍTULO 8 – CONCLUSÃO		97
8.1	RESULTADOS ALCANÇADOS	97
8.2	TRABALHOS FUTUROS	97
REFERÊNCIAS BIBLIOGRÁFICAS		99
APÊNDICE		105
APÊNDICE I – ARQUIVO EXEMPLO COM OS DADOS DO PARTICIONAMENTO DA MATRIZ A		105
APÊNDICE II – ARQUIVO EXEMPLO COM OS DADOS DO PARTICIONAMENTO DA MATRIZ B		105
APÊNDICE III – ARQUIVO EXEMPLO COM OS DADOS DO PARTICIONAMENTO DA MATRIZ C		105

Relação de Figuras

FIGURA 1 - SISTEMA COMPOSTO DE HOST E FPGA	26
FIGURA 2 ALGORITMO SEQUÊNCIAL DE MULTIPLICAÇÃO DE MATRIZES EM BLOCOS	29
FIGURA 3 ALGORITMO PARALELO DE MULTIPLICAÇÃO DE MATRIZES	30
FIGURA 4 ESQUEMA DA LÓGICA COMPUTACIONAL USADA PELO ALGORITMO	31
FIGURA 5 ORGANIZAÇÃO INTERNA DO PE	32
FIGURA 6 KERNEL DA CONVOLUÇÃO SIMPLES	33
FIGURA 7 KERNEL DA CONVOLUÇÃO DUPLA	34
FIGURA 8 ALGORITMO DA CONVOLUÇÃO 2D	34
FIGURA 9 ALGORITMO CONVOLUÇÃO 1D.....	35
FIGURA 10 COMPARAÇÃO ENTRE OS ESQUEMAS DE PARTICIONAMENTO	35
FIGURA 11 ARQUITETURA DO ALGORITMO 1 E 3.....	37
FIGURA 12 ARQUITETURA DO ALGORITMO 2 PARA $R = 2$	38
FIGURA 13 SEQUÊNCIA DA MULTIPLICAÇÃO PARALELA DA MATRIZ	40
FIGURA 14 ARQUITETURA DO ARRAY DE MULTIPLICAÇÃO DAS MATRIZES.....	40
FIGURA 15 ESTRUTURA DO ELEMENTO DE PROCESSAMENTO - PE	41
FIGURA 16 REPRESENTAÇÃO DE UMA MATRIZ GENÉRICA $M \times N$	43
FIGURA 17 ALGORITMO DE MULTIPLICAÇÃO DE MATRIZES	44
FIGURA 18 EXEMPLO DE TRANSPOSIÇÃO DE MATRIZ $N \times N$	44
FIGURA 19 PARTICIONAMENTO EM SUB-MATRIZES N/B	45
FIGURA 20 EXEMPLO DE MATRIZ 6×4 DIVIDIDA PELO FATOR 2.....	46
FIGURA 21 EXEMPLO DE MATRIZ 4×4 DIVIDIDA PELO FATOR 2.....	46
FIGURA 22 EXEMPLO DE MATRIZ 6×4 DIVIDIDA PELO FATOR 2.....	47
FIGURA 23 MATRIZ 15×15 PARTICIONADA EM BLOCOS DE 3×3	47
FIGURA 24 MATRIZ 15×15 PARTICIONADA EM BLOCOS DE 5×5	48
FIGURA 25 ESQUEMA DE UM COMPUTADOR RECONFIGURÁVEL GENÉRICO	51
FIGURA 26 ARQUITETURA DE UM FPGA GENÉRICO	52
FIGURA 27 ARQUITETURA DO HARDWARE MAP DA SRC.....	53
FIGURA 28 ARQUITETURA DO HARDWARE SRC-6 MAP.....	54
FIGURA 29 CONEXÕES DO MÓDULO FPGA NO CRAY XD1	56
FIGURA 30 ARQUITETURA DA PLATAFORMA CRAY XD1	56
FIGURA 31 ARQUITETURA DO PROCESSADOR DCP	57

FIGURA 32 SISTEMA RASC	61
FIGURA 33 MÓDULOS DE HARDWARE DO RC 100.....	62
FIGURA 34 EXEMPLO DE SEGMENTAÇÃO DE MEMÓRIA	62
FIGURA 35 ALGORITMO DO USUÁRIO E MÓDULOS DO CORE SERVICES	63
FIGURA 36 ESQUEMA INTERNO DO CHIP TIO	64
FIGURA 37 DIAGRAMA DE BLOCOS DO RASC CORE SERVICES	65
FIGURA 38 INTERFACE DO ALGORITMO COM O CORE SERVICES	66
FIGURA 39 EXEMPLO DE EXTRACTOR NA DECLARAÇÃO DE ARRAYS DE ENTRADA.....	68
FIGURA 40 FLUXO DE DESENVOLVIMENTO DE PROJETO NO RASC	68
FIGURA 41 CAMADAS DE ABSTRAÇÃO DO RASC	69
FIGURA 42 AMBIENTE DE SIMULAÇÃO DO RASC	70
FIGURA 43 FLUXO DE PROJETO DA MITRION	71
FIGURA 44 FORMAÇÃO DOS BANCOS LÓGICOS DE MEMÓRIA.....	72
FIGURA 45 ESQUEMA DE PARTICIONAMENTO E MULTIPLICAÇÃO	72
FIGURA 46 ESQUEMA OTIMIZADO PARA MAIOR DESEMPENHO	73
FIGURA 47 ESTRUTURA DA METODOLOGIA PCAM - FOSTER, 1995	75
FIGURA 48 STRIP MINING NO ALGORITMO SEQÜENCIAL	79
FIGURA 49 ALGORITMO SEQÜENCIAL DE MULTIPLICAÇÃO EM BLOCOS.....	79
FIGURA 50 ALGORITMO DE PARTICIONAMENTO DA MATRIZ A.....	80
FIGURA 51 ALGORITMO DE PARTICIONAMENTO DA MATRIZ B.....	81
FIGURA 52 ALGORITMO DE PARTICIONAMENTO DA MATRIZ C.....	82
FIGURA 53 FLUXOGRAMA DO PARTICIONAMENTO E MULTIPLICAÇÃO EM HARDWARE	83
FIGURA 54 CONSTRUÇÃO DOS BANCOS DE MEMÓRIA [18]	84
FIGURA 55 MULTIPLICAÇÃO DOS VETORES DAS MATRIZES [18]	84
FIGURA 56 ARQUITETURA DO MULTIPLICADOR [18].....	85
FIGURA 57 DIAGRAMA EM BLOCO DA PLATAFORMA XILINX ML555.....	88
FIGURA 58 DESEMPENHO DO TEMPO DE EXECUÇÃO X FREQUÊNCIA.....	91
FIGURA 59 DESEMPENHO DO TEMPO DE PROCESSAMENTO X QUANTIDADE DE MACs	92
FIGURA 60 DESEMPENHO DO TEMPO DE PROCESSAMENTO X TAMANHO DO BLOCO.....	93
FIGURA 61 DESEMPENHO DO TEMPO DE PROCESSAMENTO X TAMANHO DA MATRIZ	94
FIGURA 62 RELAÇÃO ENTRE OS TEMPOS DE PROCESSAMENTO X TAMANHO DA MATRIZ	94

Relação de Tabelas

TABELA 1 A) TEMPO DE EXECUÇÃO DE SOFTWARE E HARDWARE, B) TEMPOS DE EXECUÇÃO COM VÁRIOS HARDWARE	73
TABELA 2 TEMPOS DE EXECUÇÃO VARIANDO A ENTRADA E A COMPLEXIDADE	77
TABELA 3 - LARGURA DE BANDA DE ACESSO A MEMÓRIA E COMUNICAÇÃO COM O HOST	87
TABELA 4 - ESTIMATIVAS DE CUSTO EM TEMPO DA MOVIMENTAÇÃO DE DADOS NAS PLATAFORMAS COMERCIAIS.....	89
TABELA 5 - ESTIMATIVAS DO CUSTO DE PROCESSAMENTO E DO TEMPO TOTAL CONSIDERANDO A MOVIMENTAÇÃO DE DADOS NAS PLATAFORMAS COMERCIAIS.....	90

[PAGINA INTENCIONALMENTE DEIXADA EM BRANCO]

Glossário

ASIC (*Application System Integrated Circuit*): Circuito integrado de aplicação específica. Apresenta uma funcionalidade específica que não pode ser modificada em campo.

Bitstream: Conjunto de bits que são gerados por uma ferramenta de síntese e que são carregados no FPGA para configuração dos elementos lógicos de acordo com a funcionalidade desejada.

BLAS (*Basic Linear Algebra Subprograms*): Conjunto de rotinas eficientes que provêem blocos básicos para execução de operações entre matrizes e vetores. É dividido em três níveis: BLAS 1 que contém rotinas para operações entre vetores, BLAS 2 com rotinas para operações entre matrizes e vetores e BLAS 3 que contém rotinas para operações entre matrizes.

BRAM Block: Bloco de memória RAM interno ao FPGA e que pode ter seu comportamento, largura e número de palavras configuráveis de acordo com as necessidades do projetista.

DCP (*Direct Connected Processor*): Arquitetura em que vários processadores são integrados em um simples e unificado sistema através de uma interconexão de alta velocidade. Este tipo de configuração elimina disputas por memória compartilhada e gargalos de barramentos.

DMA (*Direct memory access*): Técnica que permite que certos dispositivos de hardware num sistema computacional acessem o sistema de memória para leitura e escrita independentemente da CPU (sem sobrecarregar a CPU).

DRAM (*Dynamic Random Access Memory*): Memória volátil dinâmica baseada em capacitores e utilizada como memória principal dos processadores. Apresenta maior capacidade de armazenamento e menor largura de banda quando comparada a memória SRAM.

DSP Block: Blocos com Lógicas especiais internas ao FPGA, em geral formados basicamente por somadores e multiplicadores, que permitem a criação de circuitos aritméticos de forma eficiente.

Cluster de computadores: Conjunto de computadores com as mesmas características que trabalham juntos na solução de um mesmo problema.

Compute blade: Placa eletrônica que corresponde a uma unidade básica de um sistema computacional. Pode ser composta por processadores, recursos reconfiguráveis e recursos de memória e conexão.

Conexão RapidArray: Rede de interconexão de alta velocidade (2GB/s) proprietária da Cray que permite a interconexão de vários *compute blades* do sistema Cray XD1.

Cray: Companhia de supercomputadores que nos últimos anos têm investido na associação de dispositivos reconfiguráveis com processador de propósito geral para acelerar o desempenho de processamento de seus produtos.

FIFO (First In First Out): Estrutura de dados baseada em fila, em que o primeiro dado a ser armazenado é o primeiro a ser lido.

FPGA (Field Programmable Gate Array): Circuito integrado que pode ser programado em campo. É formado por uma matriz de blocos lógicos configuráveis com interconexões internas e externas a esses blocos, que permitem que o projetista construa sistemas baseados em lógicas combinacionais e sequenciais.

Granularidade fina: Decomposição de uma tarefa num grande número de tarefas pequenas que podem ser executadas em paralelo por várias unidades de execução.

Granularidade grossa: Decomposição de uma tarefa num pequeno número de grandes tarefas.

Hyper Transport: Tecnologia de interconexão entre chips de alta velocidade e alto desempenho, com largura de banda de 3,2 GB/s e utilizada pelos processadores AMD.

MAC (multiply-accumulate): Módulo que realiza a operação de multiplicação de dois valores de entrada e soma com valor previamente acumulado.

NUMalink: Sistema de interconexão desenvolvido pela SGI para utilização nos seus sistemas computacionais de memória distribuída. O NUMalink 4 possui uma largura de banda de 6,4 GB/s, sendo 3,2 GB/s em cada direção.

Lane: Conexões seriais bidirecionais usadas no PCIe para transferência de dados a 250 MB/s em cada direção.

Loop: Laços, ou seja, execução de uma tarefa repetidas vezes.

QDR (*quad data rate*): Configuração de memória com portas de leitura e escrita separadas, permitindo que leituras e escritas sejam executadas em paralelo e aumentando a largura de banda da memória.

Testbench: Ambiente composto por um conjunto de ferramentas que permitem a verificação funcional de um produto em desenvolvimento.

Time to market: Espaço de tempo em que um produto precisa estar pronto para ser disponibilizado para o mercado.

Pipeline: Técnica que permite acelerar a velocidade de operação do processador, através da utilização paralela dos seus recursos. A busca de instrução (dados) e execução das operações é feitas em paralelo.

RASC: Supercomputador reconfigurável desenvolvido pela SGI, baseado na associação de um servidor Altix, com processador Itanium 2, e uma plataforma reconfigurável, RC100, com dois FPGAs Xilinx Virtex-4.

SGI (*Silicon Graphics Inc*): Empresa especializada em computação de alto desempenho e criadora do RASC. Sistema que integra um servidor baseado em processador de propósito geral e um sistema reconfigurável baseado em FPGA da Xilinx.

SRAM (*Static Random Access Memory*): Memória volátil estática baseada em flip-flop. Apresenta menor capacidade de armazenamento e maior largura de banda quando comparada à memória DRAM.

SSP (*Scalable System Port*): Interface de alta largura de banda (até 3,2 GB/s) e baixa latência utilizada pelo sistema RASC na conexão entre o recurso reconfigurável e o processador de propósito geral.

VHDL (*Very High Speed Integrated Circuit Hardware Description Language*): Linguagem de descrição de hardware utilizada para descrever sistemas digitais. Permite descrições comportamentais, estruturais e temporais da lógica de circuitos.

[PAGINA INTENCIONALMENTE DEIXADA EM BRANCO]

Capítulo 1 - Introdução

1.1 Contextualização

A solução de problemas complexos na área bancária, como análise de investimento, por exemplo, usando uma simulação Monte Carlo, poderá levar horas para ser concluída [1]; na área de saúde como visualização de imagens médicas em tempo real [2]; na indústria de petróleo, como as pesquisa de solo [3] ou na computação científica de uma forma geral, na multiplicação de grandes matrizes, vetores, etc. [4], muito usada nas instituições de ensino e pesquisa, tem exigido cada vez mais o uso de computação com alto desempenho. Estes aplicativos, em sua maioria, devido a sua grande massa de dados, grandes laços de processamento em seus procedimentos, podem consumir dias ou até meses de trabalho, em computadores com processadores de uso geral (processamento sequencial), para apresentar o resultado final. Existem casos em que este tempo excessivo pode inviabilizar a execução do projeto em questão, por perder o *time to market* de um produto [1].

As maiores empresas do ramo de computadores, como IBM [5], HP [6], SGI [7] entre outras, têm se empenhado em resolver este problema, criando máquinas com vários processadores, ou clusters de computadores, com centenas de computadores interligados em rede, para aumentar o desempenho final de processamento e assim conseguir diminuir o tempo de processamento destes aplicativos. Mesmo assim estas empresas esbarram nos limites impostos pela física dos semicondutores. Como o processamento continua a ser sequencial em cada processador, mesmo com aumento da velocidade de trabalho, esta solução apresenta novas dificuldades como, por exemplo, o aumento de consumo de energia que traz junto o problema de aquecimento do dispositivo e baixo desempenho energético.

Várias empresas da área de computação, como as citadas anteriormente, partiram para o desenvolvimento de super computadores, como a IBM que no último semestre de 2008 teve seu super computador, denominado de *Roadrunner*, considerado o super computador mais rápido na seleção do Top 500 [8]. Este computador da IBM juntamente com o super computador da Cray foram os primeiros a ultrapassar a barreira dos *petaflops/s* executando operações de ponto flutuante com o aplicativo *Limpack* [9].

Mas esta alternativa é considerada muito cara para a maioria das organizações que precisam de computação de alto desempenho. Devido a este alto custo, empresas da área de informática e as organizações que precisam acelerar processamento partiram para a construção de cluster de computadores [9] [10] [11]. Com um custo mais baixo, devido ao uso

de computadores comuns, conseguiram aumentar o processamento diminuindo o tempo gasto para execução de seus aplicativos.

Porém a infra-estrutura necessária para acomodar uma grande quantidade de máquinas termina por gerar outras dificuldades. Estes clusters construídos com uma grande quantidade de computadores, de dezenas, centenas, chegando a milhares de máquinas [12], levam ao problema de espaço para instalação destas máquinas, um sistema de ar condicionado apropriado para estas salas, além de um consumo elevado de energia por parte do sistema computacional.

Atualmente muitas destas empresas já partem para soluções especiais, adotando arquiteturas híbridas, formadas por processadores genéricos junto a co-processadores especiais, seja como processadores para vídeo GPU (Graphics Processing Unit) [13] ou customização de parte dos aplicativos em dispositivos lógicos reconfiguráveis, como o *Field Programmable Gate Array (FPGA)* [14]. A Figura 1 mostra este esquema básico de um sistema tendo um *FPGA* como co-processador.

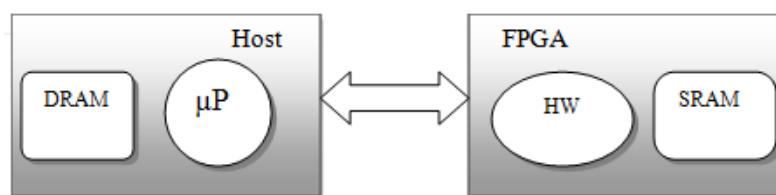


Figura 1 - Sistema composto de Host e FPGA

O uso de *FPGAs* como co-processadores em hardware para acelerar operações, como multiplicação de matrizes, por exemplo, vem sendo explorada, devido a estes dispositivos permitirem acomodar em um único chip vários elementos de processamento customizados para paralelização das tarefas, que fora particionada, em uma operação de multiplicação.

Outra vantagem no uso de co-processadores como este, diz respeito a sua baixa velocidade de operação, na ordem de 200 MHz contra os 2 ou 3 GHz dos processadores genéricos. Devido a esta baixa frequência de operação nos *FPGAs*, e ao menor número, em geral, de ciclos de relógio para processamento, por não haver necessidade de decodificar as instruções, consegue-se também um consumo menor de energia na execução da parte do aplicativo que está sendo executado em hardware.

Os principais fabricantes destes dispositivos, Xilinx [15] e Altera [16], têm oferecido dispositivos com grande capacidade de portas lógicas e blocos especiais, na forma de

hardcores, como módulos DSPs, cpus, etc., além de ambientes de projeto com ferramentas de compilação, síntese e verificação.

1.2 Desafios a Vencer

O uso de dispositivos lógicos reconfiguráveis como co-processador permite uma nova abordagem do problema. Agora, um dado aplicativo poderia ser particionado em duas partes: a primeira, aquela com características de controle, processo sequencial, continuaria sendo executada em *hosts* com processadores genéricos, enquanto que a parte com os grandes laços de processamento seriam processados, com maior desempenho por explorar o paralelismo, em co-processadores baseados em *FPGA* [17].

Este tipo de solução tem sido proposta por vários fabricantes de sistemas de alto desempenho. Porém, o acesso a dados na memória principal do *host* é considerado um grande gargalo para o processamento em hardware, devido a sua limitada largura de banda e a *latência* na requisição dos dados [18], envolvendo rotinas do sistema operacional, *device drivers*, barramento de dados, etc. Também, como estamos lidando com aplicativos de grandes massas de dados, em geral, não é possível fornecer, geralmente, aos módulos co-processadores todos os dados para processamento de forma adequada, devido ao limite de armazenamento de sua memória local. Assim, dependendo do tamanho do problema, como o processamento de grandes matrizes, foco deste trabalho, é preciso quebrar estas estruturas de dados em blocos menores para serem enviados e processados adequadamente no módulo de co-processamento.

Entretanto, esta solução nos leva a outro desafio. Dependendo da quantidade de blocos necessários para concluir a aplicação em hardware, as *latências* destas várias cargas de dados para o hardware, se somadas, podem diminuir o desempenho da aplicação. Ou seja, pode ser que não compense o uso do dispositivo co-processador, devido os tempos de transferência de dados que são introduzidos.

1.3 Objetivo

Esta dissertação tem como objetivo específico o estudo de um algoritmo de particionamento em software de grandes matrizes, para implementação em uma arquitetura computacional de alto desempenho. Este estudo visa também o armazenamento destes dados em hardware, com uma menor quantidade de blocos para concluir a multiplicação e com este armazenamento, facilitar o reuso de dados para melhorar o desempenho.

Neste trabalho estaremos focando a operação do nível 3 do *BLAS*, ou seja, multiplicação de grandes matrizes. A biblioteca *BLAS* (*Basic Linear Algebra Subprograms*) [19] é um conjunto de programas que executam operações em álgebra linear sobre vetores e matrizes. Estes programas são altamente otimizados e divididos em três níveis, a saber: o nível 1 suporta as operações com vetores; o nível 2 suporta as operações de vetores com matrizes e; o nível 3 são implementadas as multiplicações entre matrizes.

Este trabalho visa, portanto o estudo de estratégias para se alcançar uma melhor relação custo benefício no particionamento dos dados e tempo de processamento em uma arquitetura híbrida baseada em CPUs de propósito geral e co-processadores (*FPGA*).

1.4 Organização do Trabalho

Este trabalho está assim organizado: no capítulo 2 estão apresentados os trabalhos relacionados com o tema central desta dissertação, particionamento e multiplicação de grandes matrizes.

O capítulo 3 trata da fundamentação teórica que embasa o trabalho. Nele é explanado o problema do grau da decomposição de dados, o *overhead* que esta decomposição provoca, bem como o escalonamento destes dados, após particionado, para processamento em hardware.

No capítulo 4 são apresentadas algumas plataformas híbridas comerciais que foram estudadas com o intuito de verificar suas viabilidades para implementação do algoritmo de partição.

No capítulo 5 a plataforma *RASC* é apresenta em detalhes, por ter sido o ambiente de desenvolvimento que utilizamos a maior parte do tempo no projeto.

O capítulo 6 apresenta um estudo sobre estratégia de particionamento de grandes matrizes.

No capítulo 7 apresentamos um estudo de caso que realiza uma análise comparativa de desempenho estimado entre 3 plataformas reconfiguráveis comerciais utilizando a estratégia de particionamento proposta. Também é apresentado um comparativo de desempenho com a execução da multiplicação completamente em software.

O capítulo 8 apresenta as conclusões observadas ao longo deste trabalho, bem como aponta sugestões de melhorias e de possíveis novos trabalhos que possam ser desenvolvidos com base nos resultados até aqui alcançados.

As referências usadas neste trabalho estão no capítulo 9.

Também constam neste documento apêndice com informações auxiliares.

Capítulo 2 – Trabalhos Relacionados

Com o crescente e constante aperfeiçoamento dos dispositivos de lógica programável, os *FPGAs*, em quantidades de portas lógicas, blocos de memória *RAM*, módulos *DSPs* entre outras melhorias, vários pesquisadores têm voltado seus trabalhos na investigação do uso destes dispositivos para atuar como co-processador em sistemas híbridos reconfiguráveis, auxiliando um processador de uso geral em tarefas que envolvam um grande custo computacional. Neste capítulo abordaremos alguns destes trabalhos focando o problema de particionamento, reuso e armazenamento de dados na multiplicação de grandes matrizes, tendo como co-adjuvantes os co-processadores em hardware, bem como, problemas relativos à largura de banda disponível em alguns destes sistemas.

2.1 Multiplicação de Matriz em ponto Flutuante de 64 bit em FPGA

Neste trabalho [20] é apresentado um projeto de um multiplicador de matriz em hardware otimizado para implementação em um *FPGA*, com dados descritos em ponto flutuantes com precisão dupla, 64 bits, na norma ANSI/IEEE STD 754.

O algoritmo de multiplicação neste caso é inicialmente implementado em software em um computador de propósito geral, envolvendo o particionamento e a multiplicação de matrizes em blocos, de tamanhos arbitrários. Ao contrário do que se vê em um algoritmo trivial de multiplicação de matrizes, onde o índice k , o índice do produto, encontra-se no loop mais interno dos três loops, neste algoritmo sequencial em bloco o índice k encontra-se no loop externo dos três loops da multiplicação como se pode ver na figura 2. A mudança de posição deste índice provê uma nova lógica para a multiplicação dos elementos das matrizes, possibilitando fazer novos arranjos com seus dados.

```
for (i = 0; i < M/Si; i = i + 1)
  for (j = 0; j < R/Sj; j = j + 1){
    for (Li = 0; Li < Si; Li = Li + 1)
      for (Lj = 0; Lj < Sj; Lj = Lj + 1)
        C[i · Si + Li, j · Sj + Lj] = 0;
        for (k = 0; k < N; k = k + 1){
          for (Li = 0; Li < Si; Li = Li + 1)
            for (Lj = 0; Lj < Sj; Lj = Lj + 1)
              C[i · Si + Li, j · Sj + Lj] =
              = C[i · Si + Li, j · Sj + Lj] +
              + A [i · Si + Li, k] × B[k, j · Sj + Lj];}}
```

Figura 2 Algoritmo sequencial de multiplicação de matrizes em blocos

Este algoritmo faz uso da exploração da localidade de dados e do reuso presente no problema de multiplicação de matrizes para alcançar um bom desempenho. As matrizes são processadas em *streams* e os resultados são gerados em blocos.

Na proposta deste trabalho, o algoritmo seqüencial foi dividido em dois algoritmos paralelos chamados: algoritmo mestre (Master) e algoritmo escravo (Slave) apresentados pela figura 3.

Algoritmo Master:

```
Master(Mpid=0) {
  for (pid=1; pid ≤ P; pid=pid+1) Fork (Slave(pid), pid);
  for (i = 0; i < N/Si; i = i + 1)
    for (j = 0; j < N/Sj; j = j + 1){
      Store (TC[0 : Si - 1, 0 : Sj - 1], 0);
      for (k = 0; k < N; k = k + 1){
        Send (Mpid + 1, FIFOA, A [i : i + Si - 1, k]);
        for (Lj = 0; Lj < Sj; Lj = Lj + 1)
          Send (Mpid+1, FIFOB, B [k, j * Sj + Lj ] );
      }
      Load (C[i* Si : i * Si + Si - 1, j * Sj : j * Sj + Sj - 1], TC[0 : Si - 1, 0 : Sj - 1]); } }
```

Algoritmo Slave:

```
Slave (pid){
  Rcv (pid, FIFOA, A [0 : Si - 1]);
  Send (pid + 1, FIFOA, A [0 : Si - 1]);
  TA[0 : Si/P - 1] = A[(pid-1)*Si:pid*Si - 1];
  for (Lj = 0; Lj < Sj; Lj = Lj + 1){
    Rcv (pid, FIFOB, TB);
    Send (pid + 1, FIFOB, TB);
    for (L = 0; L < Si/P; L = L + 1)
      TC[Lj, L] = TC[Lj, L] + TA[L] * TB; } }
```

Figura 3 Algoritmo paralelo de multiplicação de matrizes

O algoritmo mestre é executado em um único processador, enquanto o algoritmo escravo é executado em vários processadores, implementados em um *FPGA*, chamados de *PEs* (elementos de processamento). O algoritmo mestre tem como principal função o envio de dados das submatrizes de entrada A e B, em blocos de tamanho S_i por S_j , para o array linear de cada *PE*. Em todos os *PEs*, o algoritmo é executado seguindo o seguinte fluxo: o processador mestre envia S_i elementos de uma coluna de A, de forma que cada um dos *PE* recebe S_i/p elementos; O processador mestre envia S_j elementos de uma linha de B para todos os *PEs*. Os elementos de A e B são multiplicados em cada um dos *PE* e somados com os elementos temporários da matriz C resultante. Os resultados são acumulados dentro da

memória local de cada *PE*. Esses passos se repetem N (ordem das matrizes de entrada) vezes até que as memórias locais dos *PEs* contenham $S_i \times S_j$ elementos da matriz C . A Figura 4 mostra um exemplo de execução do algoritmo para $N = 4$ e $S_i = S_j = 2$, com a utilização de dois *PEs*.

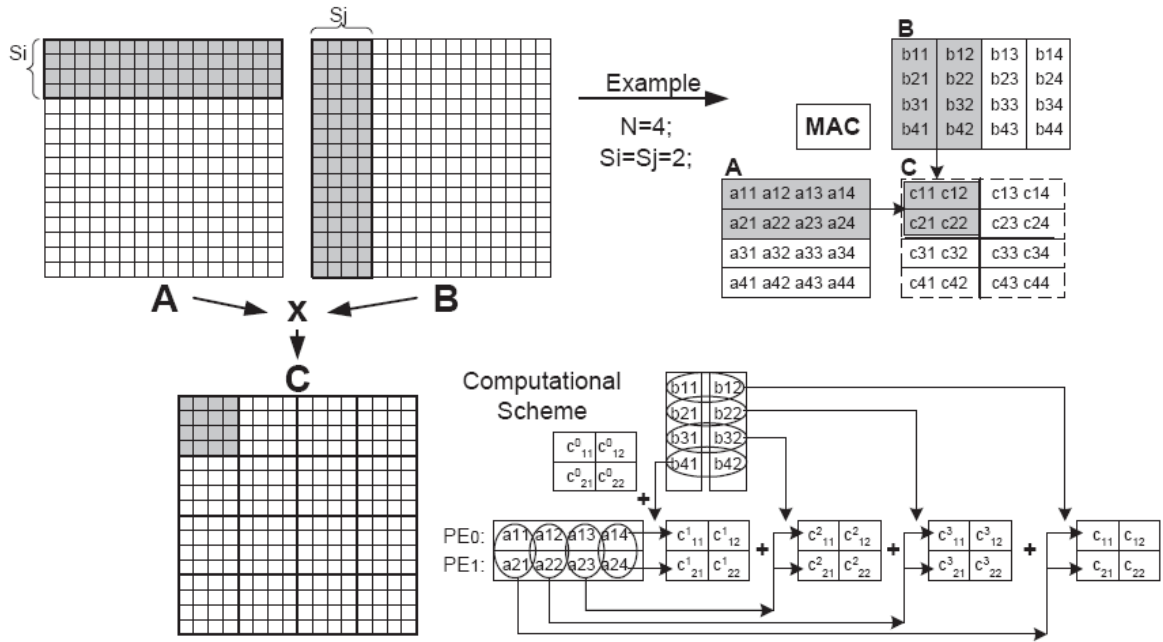


Figura 4 Esquema da lógica computacional usada pelo algoritmo

A arquitetura de [20] constitui-se de um *array linear* de *PE*. O primeiro *PE* se comunica diretamente com o *host* ou *DMA* para receber dados da memória principal e os demais *PE* se comunicam apenas com seus vizinhos da direita e da esquerda. Os *PE* são compostos por dois conjuntos de registradores, duas *FIFO*, um *MAC* e S posições de armazenamento local. Os registradores de dados *TA0* e *TA1* são projetados para armazenar S_i/p elementos da matriz A vindos do *PE* que o precede, e cada um desses elementos é reusado S_j vezes. Os elementos de B são armazenados em um registrador *TB*. Os elementos de A e B são empilhados nas *FIFO* locais (*FIFOA* e *FIFOB*) para que sejam transferidos ao *PE* vizinho. Os resultados das operações dos *MAC* são armazenados nos arquivos de registradores *TC0* e *TC1*. A Figura 5 mostra a organização do *PE*.

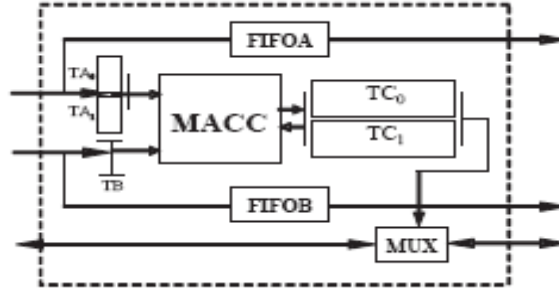


Figura 5 Organização interna do PE

Para implementação em uma arquitetura FPGA, a arquitetura proposta pelo algoritmo necessita de uma capacidade de armazenamento interno de $2 \times Si \times Sj + 2 \times Si$. O termo $2 \times Si$ é usado para armazenar os elementos das colunas de A e $2 \times Si \times Sj$ para armazenar os resultados intermediários do produto $Si \times Sj$ da matriz C .

Neste trabalho os autores mostram que a latência do algoritmo é de
$$\frac{\frac{N^3}{Sj} + \frac{N^3}{Si} + 2 \cdot N^2}{BW \cdot F}$$

ciclos de relógio, onde N é a ordem das matrizes de entrada, BW é a largura de banda no acesso aos dados da memória e F é a frequência de operação do algoritmo no FPGA.

2.2 Particionamento de Código para Computação Reconfigurável de Alto Desempenho.

No trabalho apresentado em [21] o autor explica que a computação reconfigurável baseada na tecnologia de dispositivos programáveis em campo (FPGA) ainda tem potencial para melhorar o desempenho de rendimento além daquele previsto pela Lei de Moore. Os sistemas comerciais de computação reconfiguráveis de alta capacidade, tais como Cray XD1, SGI RASC, e SRC-6 MAP, que são baseados na combinação de processadores convencionais e de FPGA, permitem aos programadores de software explorar o paralelismo funcional de granularidade grossa do processamento paralelo convencional assim como o paralelismo de granularidade fina com a execução direta no hardware em FPGAs. Um dos desafios chaves para a eficácia no uso deste sistema é a necessidade da divisão manual do algoritmo entre os processadores convencionais e o co-processador em FPGA.

Como dividir o código de maneira que o melhor desempenho total da aplicação possa ser conseguido é a pergunta fundamental da pesquisa. Alguns trabalhos têm tratado do particionamento automático do código do aplicativo, mas nenhum dos resultados obtidos foi implantado nos sistemas reconfiguráveis comerciais atuais, tais como o SRC-6 MAP. É o

programador de software que analisa o código e decide o que deve ser movido para o FPGA e o que deve ser deixado no processador convencional.

Algumas métricas comuns, tais como o número de operações e de resultados por unidade de dado, a eficiência do reuso dos dados, dados por latência, podem ser úteis para guiar o processo de particionamento. Contudo, diz o autor, há outras considerações práticas, tais como o número de vezes que a função do FPGA é chamada, o número de vezes que o engenho de acesso direto a memória (DMA) é invocado, e as tarefas da manipulação de dados no processador convencional, que podem ter um efeito adverso no desempenho total do algoritmo.

Neste trabalho foi proposto um estudo de caso de um algoritmo de convolução de imagem. Este algoritmo permite considerar três níveis de particionamento. O nível mais baixo, somente a computação intensiva, executada pela convolução 1D é levada ao FPGA e o processador trata das tarefas de manipulação da memória. No nível intermediário, o algoritmo é dividido ao longo das duas tarefas computacionais principais, sendo que o FPGA executa a convolução das linhas e depois é carregado para executar a convolução das colunas. E no nível mais alto, o algoritmo inteiro é movido ao FPGA. Uma análise é feita sobre o impacto no desempenho total das diferentes maneiras de dividir um aplicativo e a complexidade do código do FPGA.

A idéia básica da convolução da imagem é que uma janela de forma e tamanho finito, $h[k, l]$, é produzida através da varredura da imagem e o valor do pixel da saída é computado como a soma dos pesos dos pixéis da entrada, $a[m, n]$, onde os pesos são os valores do filtro atribuído a cada pixel da janela. O código desta convolução é apresentado na Figura 6.

$$a[m,n] \otimes h[k,l] = \sum_{i=0}^{k-1} \sum_{j=0}^{l-1} a[m+i,n+j]h[i,j]$$

Figura 6 Kernel da Convolução Simples

Esta janela, com seus pesos, é denominada de *kernel* da convolução. A complexidade computacional por pixel para uma janela $K \times L$ é $O(KL)$. Se o kernel da convolução $h[k, l]$ é separável, isto é, se o *kernel* puder ser escrito como $h[k, l] = h_{row}[l] \cdot h_{col}[k]$, então a convolução poderá ser executada como mostrada na Figura 7:

$$a[m,n] \otimes h[k,l] = \sum_{i=0}^{k-1} \left\{ \sum_{j=0}^{l-1} a[m+i, n+j] h_{row}[j] \right\} h_{col}[i]$$

Figura 7 Kernel da Convolução Dupla

Assim, em vez de aplicar um kernel bidimensional da convolução, é possível aplicar dois kernel unidimensional sendo o primeiro na direção de L e o segundo na direção de K. Isto reduz a complexidade computacional por pixel para $O(K+L)$. A Figura 8 apresenta o algoritmo da convolução 2D.

```

2DCONVOLUTION(A, B, M, N, Hr, Hc, L, K)
1  for m ← 0 to M-1
2    for n ← 0 to N-1
3      R1[n] ← A[m, n]
4      R2 ← 1DCONVOLUTION(R1, N, Hr, L)
5      for n ← 0 to N-1
6        B[m, n] ← R2[n]
7    end
8  for n ← 0 to N-1
9    for m ← 0 to M-1
10     C1[m] ← B[m, n]
11     C2 ← 1DCONVOLUTION(C1, M, Hc, K)
12     for m ← 0 to M-1
13       B[m, n] ← C2[m]
14   end
15 return B

```

Figura 8 Algoritmo da Convolução 2D

Neste algoritmo, A significa a imagem da entrada, B significa a imagem da saída, ambos de dimensão $M \times N$. H_r significa a convolução de *kernel* consistindo em L elementos aplicados a cada linha, e H_c significa a convolução de *kernel* consistindo em elementos de K aplicada a cada coluna.

As linhas 1-7 correspondem a convolução da linha, sendo que os pixéis de cada linha são copiados a um array R1 separado (linhas 2 – 3). Uma convolução 1D é aplicada com os coeficientes em R1 (linha 4), e os resultados são copiados para imagem B de destino (linhas 5-6). O processo se repete de maneira similar para cada coluna (linhas 8-14), onde na linha 11 uma convolução 1D também é aplicada. A Figura 9 mostra o algoritmo da convolução 1D.

```

1DCONVOLUTION(I, O, P, H, Q)
1  for  $p \leftarrow 0$  to  $P-1$ 
2     $O[p] \leftarrow 0$ 
3    for  $q \leftarrow 0$  to  $Q-1$ 
4       $O[p] \leftarrow O[p] + I[p+q] \cdot H[q]$ 
5  end
6  return O

```

Figura 9 Algoritmo Convolução 1D

Neste estudo, o autor diz que a complexidade computacional da convolução 2D é considerada $O((K+L) MN)$. Assim, para um tamanho fixo de janela, o tempo total de execução da convolução é em função do tamanho da imagem.

Figura 10 mostra um quadro comparativo de quatro implementações, sendo a primeira considerando somente o uso da CPU e mais três estratégias de particionamento do código da convolução 2D e seus tempos de execução num ambiente SRC MAP. Fica claro que a primeira estratégia é penalizada devido ao *overhead* das chamadas de função do MAP. Mesmo sendo esta estratégia de divisão intuitiva e simples de executar, ela aumenta o tempo de execução total porque a função MAP é chamada frequentemente.

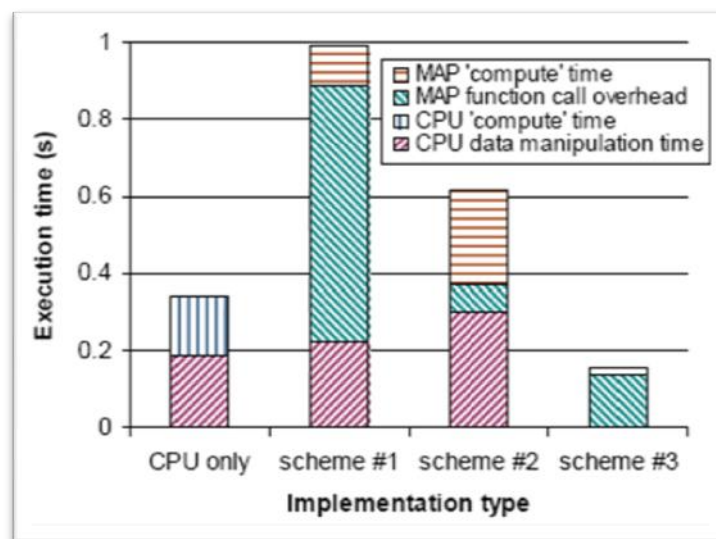


Figura 10 Comparação entre os esquemas de particionamento

A segunda estratégia que divide o código da convolução é penalizada devido à necessidade de executar manipulações de dados na memória do processador e também devido à necessidade de fazer varias chamadas ao engenho de transferências de dados (DMA).

A terceira estratégia de particionamento elimina a necessidade de manipulações de dados na memória no processador. Também elimina a necessidade de uso frequente do

engenharia de transferência de dados de e para a memória porque a imagem inteira é transferida somente uma vez. A consequência direta destas ações é que sendo o tempo de execução do código da convolução no MAP muito curto, o tempo de execução total fica dominado pelo *overhead* das chamadas de função do MAP.

2.3 Algoritmo Modular e Escalável para Multiplicação de Matriz de Ponto Flutuante em Sistemas de Computação Reconfigurável

Na proposta em [22] são apresentados três algoritmos para execução de multiplicação de matrizes em *FPGAs*. Os algoritmos são baseados em elementos de processamento *PEs*, organizados em uma arquitetura de *array* linear com lógicas simples de controle. Os autores apresentam uma análise detalhada na implementação desse problema para processamento em hardware, levando em consideração o paralelismo da aplicação e as restrições de recursos, tais como, número de *slices* configuráveis, tamanho da memória interna do *FPGA*, e largura de banda com a memória externa. Todos esses parâmetros são importantes para determinar a *latência* total da operação.

Os algoritmos levam em conta certos compromissos com relação ao tamanho da memória no projeto e os requisitos de largura de banda. Se o tamanho da memória disponível no *FPGA* para a execução do algoritmo diminui, a largura de banda com a memória externa deve ser aumentada, para evitar que a *latência* da operação seja aumentada. Por outro lado se a largura de banda da arquitetura é aumentada, para reduzir o tempo de busca dos dados, uma maior quantidade de *PEs* é necessária para reduzir, também, o tempo de computação e assim diminuir a *latência* total do projeto. Quando mais *PEs* são implementados para diminuir o tempo de computação, aumenta-se a necessidade por quantidade de dados para alimentar os *PEs*, ou seja, uma maior largura de banda com a memória.

Por fim, um terceiro compromisso ocorre entre o número de *PEs* do projeto e o tamanho da memória local em cada *PE*. Com um maior número de *PEs* configurados em um dispositivo, a quantidade de armazenamento local de cada um dos *PEs* diminui. Por outro lado, se as restrições de memória do dispositivo *FPGA* requerem que o projetista diminua o tamanho da memória local, vários dispositivos podem ser necessários para prover mais *PEs*.

Com base nestas observações três algoritmos para multiplicação de matrizes quadradas de ordem n são propostos. Os dois primeiros algoritmos necessitam de uma capacidade de armazenamento interno da ordem de n^2 , e alcançam uma *latência* ótima da ordem de n^2 com

máximo reuso de dados. No terceiro algoritmo, se M representa a capacidade de armazenamento interno, é admitido que M seja menor que n^2 . Com p PEs , a latência do terceiro algoritmo é da ordem de $\frac{n^3}{p}$.

O primeiro algoritmo consiste de um *array* linear de $\frac{n^2}{s}$ ($1 \leq s \leq n$) PEs . Cada um dos PE contém um multiplicador e um somador ponto-flutuante, além de dois registradores e uma memória de tamanho s . Cada um dos PE possui uma porta de entrada para A, uma porta de entrada para B e uma porta de saída para C. O l -ésimo PE recebe dados do $l-1$ -ésimo PE a sua esquerda e passa-os para o $l+1$ -ésimo PE a sua direita. Os elementos finais de C são transferidos da direita para a esquerda. O primeiro PE , o PE_0 , é conectado à memória externa. A largura de banda desse algoritmo é três palavras por ciclo, sendo duas palavras de leitura e uma de escrita. A Figura 11 ilustra a arquitetura do primeiro algoritmo que é idêntica ao terceiro.

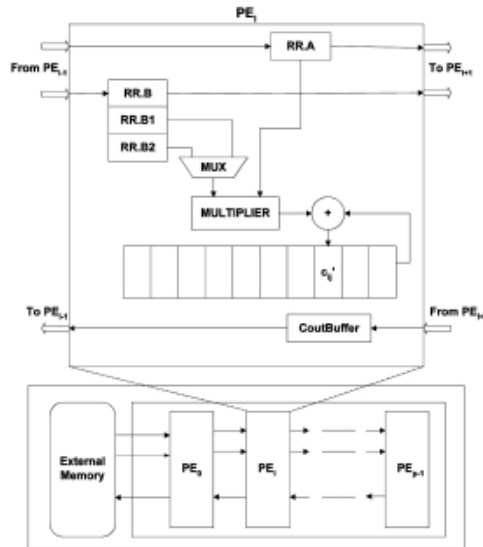


Figura 11 Arquitetura do algoritmo 1 e 3

Os autores mostram que o primeiro algoritmo apresenta uma latência da ordem de n^2 , o qual é considerado uma ótima *latência*, e necessita de uma capacidade de armazenamento M da ordem de n^2 palavra para garantir o máximo reusa dos dados.

O segundo algoritmo apresenta uma largura de banda de $3 \times r$ palavras por segundo ($1 \leq r \leq n$) e uma latência aproximada de $\frac{1}{r}$ da latência do algoritmo 1. Neste segundo

algoritmo há um array linear de $\frac{n^2}{r^2 * s}$ PEs . Cada um dos PE contém r^2 registradores, r^2 multiplicadores e somadores em ponto flutuante e r^2 blocos de armazenamento de s palavras ($1 \leq s \leq \frac{n}{r}$). Cada um dos PE possui r^2 portas de entrada, sendo r para leitura dos elementos da matriz A e r para a leitura dos elementos da matriz B, concorrentemente e r portas de saída. Neste algoritmo, o cálculo dos elementos da matriz de saída é executado em paralelo, pelos r^2 MAC de cada PE . A arquitetura do segundo algoritmo é mostrada na Figura 12.

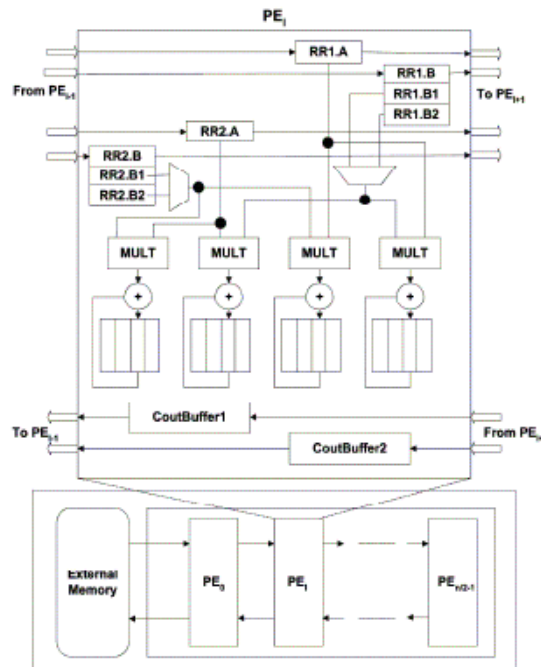


Figura 12 Arquitetura do algoritmo 2 para $r = 2$

Os autores provam que o segundo algoritmo apresenta uma latência da ordem de $\frac{n^2}{r}$ ciclos de relógio e necessita de uma capacidade de armazenamento, M , de n^2 palavras.

A arquitetura do terceiro algoritmo é similar a do primeiro algoritmo. Existem p PEs conectados em um array linear e o primeiro PE , o PE_0 , é conectado à memória externa. Cada um dos PE contém um multiplicador e um somador ponto flutuante. Entretanto, diferente do que ocorre no primeiro algoritmo, cada um dos PE no terceiro algoritmo possui uma memória

local de tamanho $\frac{M}{P}$. O algoritmo executa multiplicações de matrizes em blocos de tamanho \sqrt{M} . Quando $\sqrt{M} = n$, o primeiro e terceiro algoritmo se igualam. Quando $n > \sqrt{M}$ as matrizes de entrada são particionadas em $\left(\frac{n}{\sqrt{M}}\right)^2$ submatrizes de ordem \sqrt{M} .

No terceiro algoritmo é mostrado que a latência desse algoritmo é da ordem de $\frac{n^3}{p}$ o que representa a necessidade de uma largura de banda da ordem de $\frac{P}{\sqrt{M}}$.

2.4 Metodologia para Utilização Eficiente dos Recursos das Novas Gerações de FPGA.

No trabalho em [23] é apresentada uma metodologia para utilização eficiente dos recursos das novas gerações de FPGAs em aplicações de multiplicação de matrizes com elementos inteiro ou em ponto flutuante. O objetivo do trabalho é apresentar uma arquitetura de multiplicação de matrizes escalável e eficiente, de forma a reduzir o tempo de computação do algoritmo.

Na arquitetura proposta neste trabalho não há comunicação entre os elementos de processamento, o que ocasiona uma melhor utilização dos recursos, tendo em vista a diminuição de lógicas para comunicação e roteamento entre esses elementos. Essa melhor utilização dos recursos possibilita um aumento no grau de paralelização da arquitetura, tendo em vista a possibilidade de instanciar um maior número de elementos de processamento. Além disso, cada elemento de processamento opera independentemente, sendo a frequência de operação do elemento independente do tamanho da matriz.

A arquitetura de multiplicação de matrizes proposta baseia-se no ótimo reuso de dados dos elementos das matrizes de entrada. Este reuso é observado quando os dados das matrizes A e B são lidos apenas uma vez. Com a leitura simultânea de uma coluna da matriz A e uma linha da matriz B é possível a execução de todas as operações baseadas nesses valores, acarretando desta maneira, um ótimo reuso dos dados. Dados lidos nessa sequência permitem o cálculo de termos parciais de todos os elementos da matriz de saída C. Um exemplo do processo é mostrado na Figura 13.

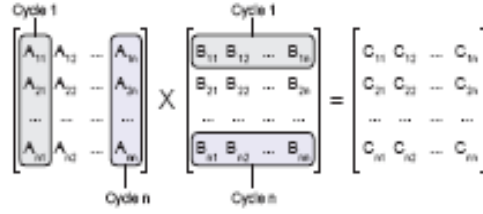


Figura 13 Sequência da multiplicação paralela da matriz

Na arquitetura proposta, apresentada na Figura 14, podemos observar a presença de elementos de memória RAM, implementados com blocos de RAM e elementos de processamento necessários quando a matriz A tem l elementos e a matriz B tem m elementos. Nesta abordagem os blocos de RAM, também chamados de BRAM, alimentam os PEs, que são feitos a partir de blocos DSP. Os blocos DSP contêm internamente um multiplicador e um acumulador. No momento em que os dados estão saindo do somador, novos dados estão saindo do multiplicador, fazendo assim as duas operações, de multiplicação e soma no mesmo ciclo de relógio.

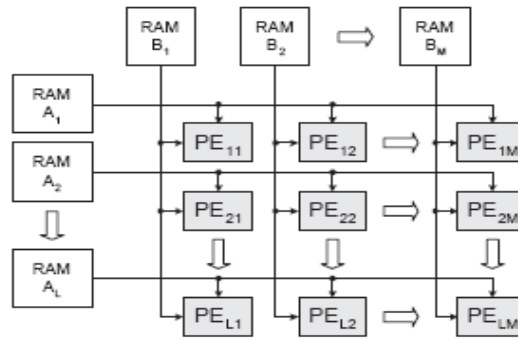


Figura 14 Arquitetura do array de multiplicação das matrizes

Com a configuração apresentada, $l \times m$ produtos são gerados a cada ciclo, a partir de $l + m$ elementos das matrizes de entrada. Sem perda da generalidade, os autores consideram que $m = l$ e que a ordem das matrizes n satisfaz a condição.

A estrutura de cada um dos PE é formada por duas entradas, para os elementos de A e B, um multiplicador acumulador (MAC), e uma FIFO de saída. A arquitetura do PE é apresentada na Figura 15.

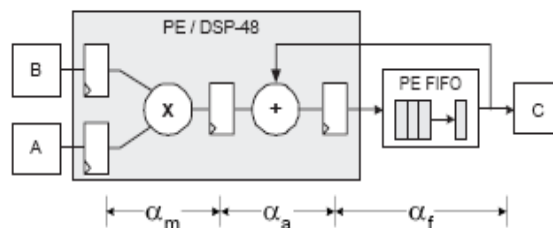


Figura 15 Estrutura do elemento de processamento - PE

Cada um dos elementos tem uma latência associada. Durante o cálculo do elemento da matriz de saída C_{ij} o produto do termo $A_{ik} \times B_{kj}$ deve estar disponível para a saída do somador, durante o mesmo ciclo de relógio em que o produto $A_{i(k+1)} \times B_{(k+1)j}$ é disponibilizado pelo multiplicador.

A estrutura de memória utilizada considera a utilização das *BRAMs* para armazenar os elementos das matrizes de entrada e de saída. Cada matriz de entrada é particionada em m bancos de memória. Cada banco A armazena $k = \frac{n}{m}$ palavras de cada coluna de A, para toda linha de A. Cada banco de B armazena $k = \frac{n}{m}$ palavras de cada linha de B, para todas as colunas de B. Com esta arquitetura, considerando que os dados estão disponíveis a cada ciclo de relógio, para realizar as n^3 operações da multiplicação de matrizes, com $p = m^2$ elementos de processamento, o tempo de computação é da ordem de $\frac{n^3}{p}$.

2.5 Conclusão

O particionamento de grandes matrizes é determinado pela capacidade de armazenamento interno do recurso reconfigurável. A latência da operação completa de multiplicação de grandes matrizes é analisada e determinada como um compromisso entre a largura de banda disponível entre o módulo *host* e os *PEs*, levando-se em conta o tempo de leitura dos dados, latência existente na transferência de dados entre o *host* e o *FPGA*, a capacidade de armazenamento interna do dispositivo reconfigurável, o que implica no fator de reuso de dados e conseqüentemente no particionamento dos dados da aplicação.

Outro aspecto importante discutido em [21] é que a melhor maneira de fazer o particionamento de código do problema (hw/sw codesign), dividir um código entre módulos de software e hardware pode não ser a mais intuitiva. A melhor solução pode depender da arquitetura em uso. No caso exposto, para a plataforma SRC-6 MAP a solução foi transferir todo o código para ser processado pelo *FPGA*.

Os recursos de armazenamento ora existentes nos *FPGAs*, como as *BRAM*, também devem ser explorados adequadamente otimizando o processamento interno do algoritmo, além da exploração de estruturas *pipelines* para paralelização no processamento interno.

[PAGINA INTENCIONALMENTE DEIXADA EM BRANCO]

Capítulo 3 – Fundamentação Teórica

Quando se trata de processamento de grandes massas de dados, que envolvam o desenrolar de grandes *loops*, como por exemplo, na multiplicação de grandes matrizes, existe a possibilidade destes dados não poderem ser acomodados dentro da estrutura de armazenamento dos co-processadores de hardware. Desta maneira, é necessária a decomposição dos dados para serem processados em partes menores.

Neste capítulo vamos abordar o conceito sobre particionamento de dados, o tamanho dos blocos deste particionamento, e o seu envio para os co-processadores em hardware. Também abordaremos o custo ocasionado pela movimentação de dados.

3.1 Multiplicação de Matrizes

Em matemática, uma matriz é uma tabela retangular de números delimitada por colchetes ou parentes e é indicada por uma letra maiúscula. Cada número da tabela é um elemento. As matrizes são classificadas pela sua dimensão, que é o número de linhas e colunas que elas possuem. Quando a quantidade de linhas e colunas é igual, a matriz é considerada uma matriz quadrada. Em programação, uma matriz é considerada como um agrupamento de variáveis. A Figura 16 mostra uma matriz genérica $m \times n$.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{pmatrix}$$

Figura 16 Representação de uma matriz genérica $m \times n$

O uso de matrizes é muito útil quando precisamos trabalhar com grandes conjuntos de dados. Podemos alterar o valor de cada elemento de uma matriz com operações como de soma, subtração e multiplicação, desde que sejam respeitadas as regras existentes para cada tipo de operação. O produto de duas matrizes é possível quando o número de colunas da primeira matriz (matriz da esquerda) é igual ao número de linhas da segunda matriz (matriz da direita). O produto é obtido pelo somatório do produto dos elementos da linha de A pelos elementos da coluna de B[24]. A fórmula abaixo demonstra esta equação.

$$\forall(i, j) [1 \leq i < n, 1 \leq j < p]: C(i, j) \Leftarrow \sum_{k=1}^m A(i, k) * B(k, j)$$

A multiplicação de matrizes requer *nmp* operações aritméticas de multiplicação, soma e acumulação. Fazendo uma análise assintótica, considerando que $n=m=p$, o algoritmo da multiplicação de matrizes é de ordem $O(n^3)$, ou seja, esta multiplicação possui um custo de n^3 operações. Um algoritmo para multiplicação de matrizes pode ser visto na Figura 17.

```

Para i = 1 até n faça
  Para j = 1 até p faça
    Para k = 1 até m faça
       $C(i, j) = C(i, j) + A(i, k) * B(k, j)$ 
    Fim para
  Fim para
Fim para

```

Figura 17 Algoritmo de multiplicação de matrizes

Na multiplicação de matrizes em hardware, é necessário que o *streams* de dados da matriz B sejam enviados de uma maneira tal que favoreça o algoritmo na busca dos dados para processamento. Com os dados das colunas da matriz B enviados como linhas, o algoritmo não precisará fazer cálculos e efetuar saltos para referenciar os elementos da coluna na multiplicação. Este rearranjo das colunas da matriz B se faz com a operação de transposição de matrizes. A equação abaixo mostra esta operação.

$$A^t_{(j,i)} = A_{(i,j)}$$

A Figura 18 mostra um exemplo de matriz $n \times m$ transposta.

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad A^t = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Figura 18 Exemplo de transposição de matriz $n \times m$

3.2 Particionamento e Escalonamento dos Dados

Quando os aplicativos exigem taxas de desempenho que não são facilmente obtidos com o processamento sequencial, o processamento paralelo oferece uma solução. Processamento paralelo é baseado em vários processadores trabalhando em conjunto para realizar uma tarefa [25]. A idéia básica é a de quebrar, ou particionar, a computação em unidades menores que são distribuídos entre os processadores. O processamento distribuído adequadamente nestes processadores pode reduzir significativamente o tempo total de execução do aplicativo. Mas esta parcela de diminuição do tempo está atrelada a parte

paralelizável do aplicativo, ou seja, quanto maior for a parcela do aplicativo que puder ser direcionada para ser processada em paralelo, maior será o ganho de desempenho.

A maioria dos algoritmos paralelos incorre em dois custos básicos: a computação relacionada com as operações lógico/aritméticas, e a comunicação, que inclui o custo com a movimentação de dados. Numa análise realista, ambos os fatores devem ser considerados.

Duas abordagens básicas são usadas para paralelizar um aplicativo:

- **Particionamento Funcional** – onde uma função do aplicativo é dividida entre os processadores existentes no sistema. Cada processador executa uma parte da função, ou seja, uma subfunção sobre o mesmo conjunto de dados. Em seguida estes mesmos dados passam para o próximo processador, enquanto um novo conjunto de dados entra no sistema para ser processado. Os dados seguem de processador em processador, como uma linha de montagem conhecida como *pipeline*. No final do processo consegue-se uma diminuição no tempo total de execução do aplicativo.
- **Particionamento de Dados** – onde cada processador executa a mesma função em diferentes blocos de dados ao mesmo tempo. Este princípio é conhecido como *Single Instruction Multiple Data (SIMD)* [24]. Desta maneira o tempo de execução é dividido pela quantidade de processadores participantes do processo. Esta abordagem exige algoritmos com paralelismo intrínseco como os de multiplicação de matrizes.

3.2.1 Particionamento em Blocos

Estratégias de particionamento de dados em blocos são freqüentemente adotadas em operações numéricas paralelas como forma de agregar desempenho às operações. Por causa disso, muitas propostas podem ser encontradas, especialmente para métodos de resolução de sistemas lineares [26]. A Figura 19 mostra este esquema de divisão de matrizes em blocos.

$$\begin{array}{|c|} \hline \textbf{A} \\ \hline \begin{array}{cccc} a_{11} & a_{12} & \dots & a_{1\frac{N}{b}} \\ a_{21} & a_{22} & \dots & a_{2\frac{N}{b}} \\ \dots & \dots & \dots & \dots \\ a_{\frac{N}{b}1} & a_{\frac{N}{b}2} & \dots & a_{\frac{NN}{bb}} \end{array} \\ \hline \end{array} \times \begin{array}{|c|} \hline \textbf{B} \\ \hline \begin{array}{cccc} b_{11} & b_{12} & \dots & b_{1\frac{N}{b}} \\ b_{21} & b_{22} & \dots & b_{2\frac{N}{b}} \\ \dots & \dots & \dots & \dots \\ b_{\frac{N}{b}1} & b_{\frac{N}{b}2} & \dots & b_{\frac{NN}{bb}} \end{array} \\ \hline \end{array} = \begin{array}{|c|} \hline \textbf{C} \\ \hline \begin{array}{cccc} c_{11} & c_{12} & \dots & c_{1\frac{N}{b}} \\ c_{21} & c_{22} & \dots & c_{2\frac{N}{b}} \\ \dots & \dots & \dots & \dots \\ c_{\frac{N}{b}1} & c_{\frac{N}{b}2} & \dots & c_{\frac{NN}{bb}} \end{array} \\ \hline \end{array}$$

Figura 19 Particionamento em submatrizes N/b

Para a divisão de matrizes quadradas em blocos é necessário que o tamanho da matriz seja múltiplo do tamanho do bloco que se quer obter. Quando dividimos uma matriz quadrada de dimensão N por blocos de dimensão b , obtemos um fator (ft) que nos dará a quantidade de blocos (qb) que será criado e também a quantidade de multiplicação (qm) que haverá entre os blocos para que a matriz seja multiplicada completamente.

A quantidade de blocos é dada pela equação $qb = \left(\frac{N}{b}\right)^2$, e a quantidade de multiplicações obtida na operação é dada pela equação $qm = \left(\frac{N}{b}\right)^3$.

Esta regra também se aplica em matrizes retangulares, desde que se encontre um denominador que seja comum para as duas dimensões das matrizes envolvidas na multiplicação. Neste caso teremos que dividir a quantidade de linhas e colunas, separadamente por este denominador comum, para encontrarmos o bloco da submatriz resultante.

A Figura 20 apresenta um exemplo de particionamento de matrizes retangulares. Considere uma matriz A $m \times n$ de dimensões 6×4 . Usando o fator 2, obteríamos blocos de submatrizes de dimensões 3×2 . O fator ft é substituído nas equações anteriores pelo $\frac{N}{b}$. Para matrizes retangulares $qb = (ft)^2$ e $qm = (ft)^3$.

$$\begin{array}{c} \text{A} \\ \left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 8 & 7 \\ 6 & 5 & 4 & 3 \end{array} \right] \Rightarrow \left[\begin{array}{cc} 1 & 2 \\ 5 & 6 \\ 9 & 1 \end{array} \right] \end{array}$$

Figura 20 Exemplo de matriz 6×4 dividida pelo fator 2

Para uma matriz B $n \times p$, de dimensões 4×4 , Figura 21, aplicando-se a mesma regra, usando o mesmo fator $ft = 2$, obtemos blocos de submatrizes de dimensões 2×2 . As submatrizes formadas pelos blocos de A , 3×2 e de B , 2×2 , são multiplicáveis entre si, pois a quantidade de colunas dos blocos da submatriz A é igual à quantidade de linhas dos blocos da sub-matriz B .

$$\begin{array}{c} \text{B} \\ \left[\begin{array}{cc|cc} 3 & 4 & 5 & 6 \\ 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 \\ 7 & 6 & 5 & 4 \end{array} \right] \Rightarrow \left[\begin{array}{cc} 3 & 4 \\ 2 & 3 \end{array} \right] \end{array}$$

Figura 21 Exemplo de matriz 4×4 dividida pelo fator 2

A matriz resultante desta multiplicação particionada por blocos será uma matriz, por exemplo, C, $m \times p$, que também deverá ser particionada pelo o mesmo esquema.

Neste exemplo, a matriz resultante C tem dimensões 6x4 como mostra a Figura 22.

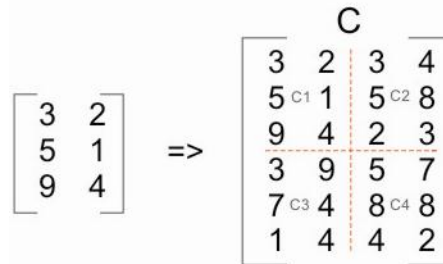


Figura 22 Exemplo de matriz 6x4 dividida pelo fator 2

Cada bloco da submatriz resultante C, de dimensões 3x2, será formado pela multiplicação dos blocos das submatrizes A e B na seguinte seqüência:

$$C1 = A1 * B1 + A2 * B2;$$

$$C2 = A1 * B3 + A2 * B4;$$

$$C3 = A3 * B1 + A4 * B2;$$

$$C4 = A3 * B3 + A4 * B4;$$

Cabe destacar que algoritmos como os de resoluções de sistemas lineares Linpack, utilizam técnicas de particionamento em blocos para explorar ao máximo as arquiteturas paralelas, servindo, por isso, como *benchmark* de avaliação de desempenho de máquinas paralelas [26].

3.2.2 Granularidade de Blocos

Após analisarmos as possibilidades da capacidade de armazenamento interno do algoritmo, no FPGA, pode-se decidir qual o melhor tamanho do bloco, ou granularidade, que trará a melhor solução custo benefício, com melhor adequação ao problema.

Como um exemplo, suponhamos a multiplicação de duas matrizes de dimensão 15x15, particionadas em blocos de submatrizes de dimensão 3x3 (Figura 23).

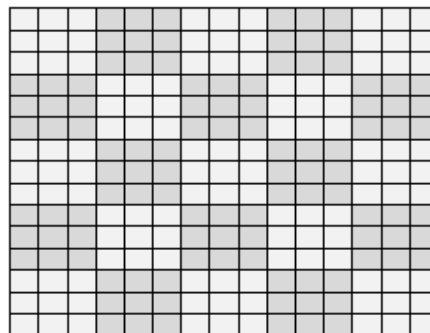


Figura 23 Matriz 15x15 particionada em blocos de 3x3

Aplicando a equação da divisão de quantidade de blocos temos:

$$qb = \left(\frac{N}{b}\right)^2 \Rightarrow \left(\frac{15}{3}\right)^2 = 5^2 = 25 \text{ blocos de submatrizes de tamanho } 3 \times 3.$$

Analisando a quantidade de multiplicações envolvidas temos:

$$qm = \left(\frac{N}{b}\right)^3 \Rightarrow \left(\frac{15}{3}\right)^3 = 5^3 = 125 \text{ multiplicações entre blocos que serão efetuadas. Isto}$$

significa que serão feitas 2 x 125 transferências de blocos para as submatrizes de A e B, e mais 125 transferências para trazer os blocos das submatrizes resultados, perfazendo um total de 375 transferências. Se a título de exemplificarmos os custos desta operação, colocássemos o valor de 3 ciclos de relógio para cada transferência de bloco, teríamos um total de 1125 ciclos de relógio.

Porém, se ao invés de blocos de dimensão 3x3, pudéssemos usar blocos de dimensão 5x5 como mostrado na Figura 24? Refazendo as mesmas contas anteriores, agora com $b=5$, teríamos 9 blocos de submatrizes de tamanho 5x5, com um total de 27 multiplicações entre blocos.

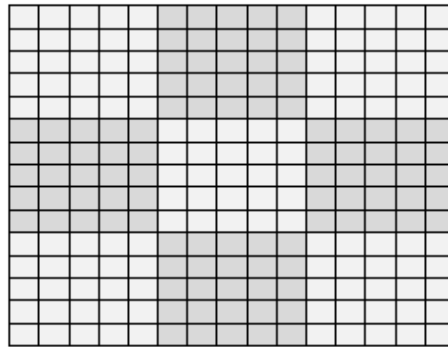


Figura 24 Matriz 15x15 particionada em blocos de 5x5

Como são blocos maiores, para exemplificar os custos, colocaremos o valor de 10 ciclos de relógio para cada transferência. Com um total de 81 transferências, 2 x 27 para transferir as submatrizes A e B e mais 27 transferência para buscar as submatrizes resultados, os custos seriam de 810 ciclos de relógio para completar as transferências da multiplicação.

Vê-se que no primeiro caso houve um gasto de 315 ciclos de relógio a mais que o segundo caso para efetuar a mesma quantidade de operações da multiplicação das matrizes 15x15. No primeiro caso, houve uma granularidade fina de blocos, ocasionando uma grande quantidade de multiplicação entre esses blocos pequenos. Com o aumento do tamanho dos

blocos, ou seja, com a diminuição da granularidade, conseguimos executar a mesma operação final com uma quantidade menor de blocos, levando a um custo menor nas transferências. Quanto maior for a dimensão das matrizes envolvidas na multiplicação e não havendo a contra partida com o tamanho dos blocos das submatrizes, ou seja, se estes blocos não puderem ter seu tamanho aumentado, maior será o custo com o aumento da quantidade de transferências. Este tipo de overhead poderia inviabilizar a execução desta operação no hardware.

3.3 Conclusão

O problema da granularidade precisa ser bem equacionado para não provocar *overhead* excessivo no tempo total da aplicação. Quando o tempo de comunicação e sincronização para criar uma tarefa é maior que o tempo de execução, ou seja, tempo que esta tarefa gasta para executar determinadas operações, significa que é inviável criar esta tarefa para ser executada em hardware.

Enquanto que no estudo de particionamento de dados para processadores de uso geral, é visto que uma granularidade grossa pode afetar o desempenho da aplicação devido a capacidade da cache não suportar o tamanho do “*grão*”, para os coprocessadores de hardware, esta granularidade grossa, pode ser uma boa solução por requerer que mais processamento seja feito com menor quantidade de transferência de dados. Porém fica, também, na dependência da capacidade de armazenamento local interno.

O algoritmo de multiplicação de matrizes permite, com seu particionamento, uma paralelização de dados sem que os elementos de processamento necessitem de comunicar-se para cumprir a tarefa. Cada elemento recebe sua cota de dados, trabalha sobre ela, e depois de concluído, devolve para o local concentrador de resultados, onde todos os elementos de processamento deverão depositar seus resultados também.

Neste trabalho está sendo considerada apenas matrizes quadradas. O foco deste trabalho é o de particionar matrizes, verificar o envio de seus blocos para o processamento em hardware e, após a multiplicação, receber os blocos com as submatrizes multiplicadas e montar a matriz resultante.

[PAGINA INTENCIONALMENTE DEIXADA EM BRANCO]

Capítulo 4 – Plataformas Híbridas Reconfiguráveis

Plataformas híbridas reconfiguráveis são sistemas computacionais compostos por um computador de uso geral junto a um co-processador de hardware reconfigurável visando explorar os benefícios que cada um tem de melhor.

O processador de hardware reconfigurável é um dispositivo com características intermediárias entre os processadores de uso geral e os processadores de uso específico, os *ASICs*. Este tipo de arquitetura combina a flexibilidade dos processadores de uso geral com o alto desempenho dos *ASICs* [27]. Existem vários tipos de arquiteturas no mercado voltadas para aplicações de alto desempenho, dentre as quais GPU [28], CELL [29] e FPGA [30]. FPGA é o hardware foco deste nosso estudo.

Como o hardware reconfigurável pode ser customizado para resolver um problema específico, ele pode alcançar um desempenho melhor que um processador de uso geral. Esta eficiência se dá pelo fato de que o hardware executa diretamente as operações aritméticas, sem ter que decodificar nenhuma instrução previamente, como acontece no processador de uso geral. Além disso, sua estrutura interna permite a construção de estruturas paralelas, uso de *pipeline*, etc. possibilitando maior paralelismo e, em consequência, melhor desempenho. Isto é verdade principalmente em problemas com características de independência de dados. A Figura 25 mostra a estrutura de um computador reconfigurável genérico.

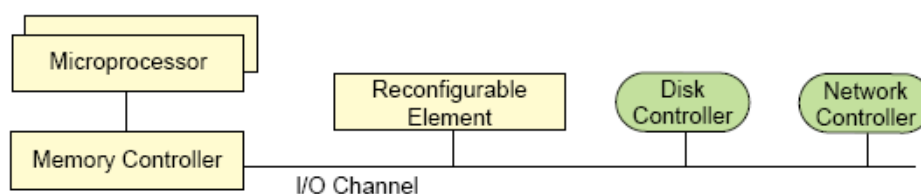


Figura 25 Esquema de um computador reconfigurável genérico

Outro ponto a ser colocado a favor do processador de hardware reconfigurável é que ele pode ser programado e reprogramado em campo, isto é, após a sua fabricação ele pode receber novas configurações fazendo com que ele passe a desempenhar outras funções. Este fato faz com que estes dispositivos apresentem um menor custo de projeto e maior flexibilidade que os *ASICs*.

No caso dos FPGA, eles são configurados e re-configurados através de lógica pré-fabricada que se encontram armazenadas em memórias de configuração do tipo SRAM, no dispositivo. Com a execução de um *bitstream* nesta memória, podemos construir a lógica que

programe determinado algoritmo, que seria específico para resolver um determinado problema. Os *FPGAs* são construídos como uma matriz de blocos lógicos configuráveis e interconexões internas e externas aos blocos. A arquitetura de um *FPGA* genérico é apresentada na Figura 26.

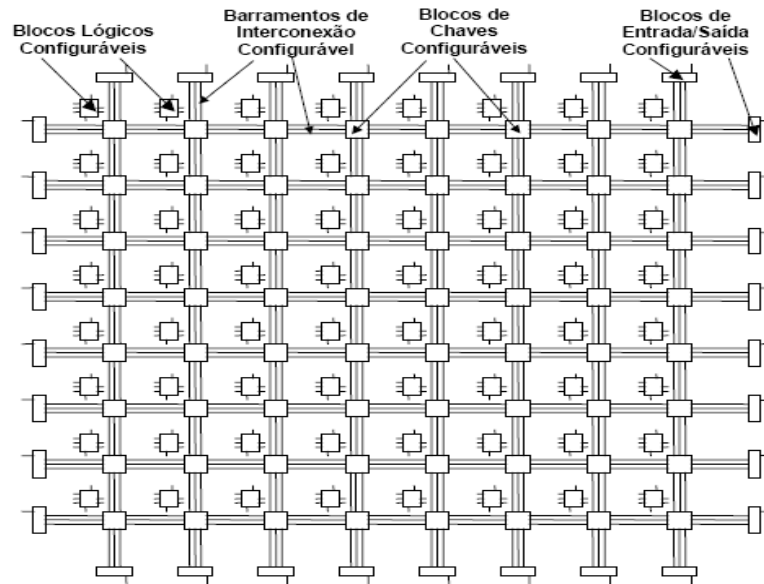


Figura 26 Arquitetura de um *FPGA* genérico

Os processadores de uso geral, com arquitetura *Von Neumann* e alta frequência de operação, são mais eficientes para tarefas de controle, que não exijam uma grande capacidade de computação intensiva.

Os sistemas de computadores reconfiguráveis de alto desempenho consistem assim, de um ou mais processadores de uso geral, hardware reconfigurável e uma arquitetura de interconexão de alto desempenho. As tarefas podem ser executadas somente nos processadores, ou no hardware reconfigurável ou em ambos. Este tipo de projeto cooperativo permite a exploração dos benefícios do paralelismo da computação de alto desempenho em conjunto com a aceleração em hardware reconfigurável [31], [32], [33].

A seguir plataformas híbridas serão apresentadas e discutidas em detalhes como parte de estudo de adequação de grandes problemas em sistemas reais.

4.1 Plataforma SRC MAPstation

O SRC-6 MAPstation é o sistema reconfigurável que integra um microprocessador Intel, executando o sistema operacional Linux, e um ambiente de hardware reconfigurável, denominado MAP (*Multi-adaptive Processor*) [34]. Cada processador MAP consiste em

dois *FPGAs Xilinx Virtex II Pro XC2VP100* e um controlador baseado em *FPGA*. Cada *FPGA* tem acesso a seis bancos de memória *SRAM* on-board denominados *OBM (On-Board Memory)*, de 4 MBytes cada, que provêem uma largura de banda total de 4.8 GB/s.

O controlador de *FPGA* facilita a comunicação e o compartilhamento de memória entre os microprocessadores e os *FPGAs*. Nesta arquitetura, o programador é explicitamente responsável pela transferência de dados para os, e dos, bancos de memória usando biblioteca de funções fornecidas pelo fabricante que são chamadas de dentro da aplicação em *FPGA*.

Múltiplos MAPstations podem ser conectados através de rede *Ethernet* formando um cluster. A Figura 27 mostra a arquitetura de hardware do processador MAP.

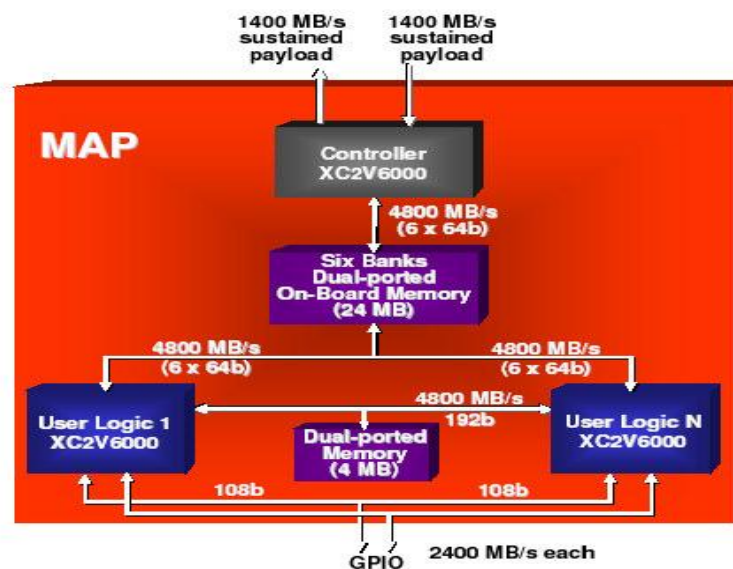


Figura 27 Arquitetura do hardware MAP da SRC

Na Figura 28, vemos a arquitetura do SRC-6 MAPstation, consistindo de uma placa-mãe com processador *Intel dual Xeon* de 2.8 GHz e o processador MAP, interconectados a 1.4 GB/s via um switch *Hi-Bar*. Nesta arquitetura, a placa de interface SNAP [35], plugada no slot DIMM, conecta a placa-mãe ao switch *Hi-Bar*.

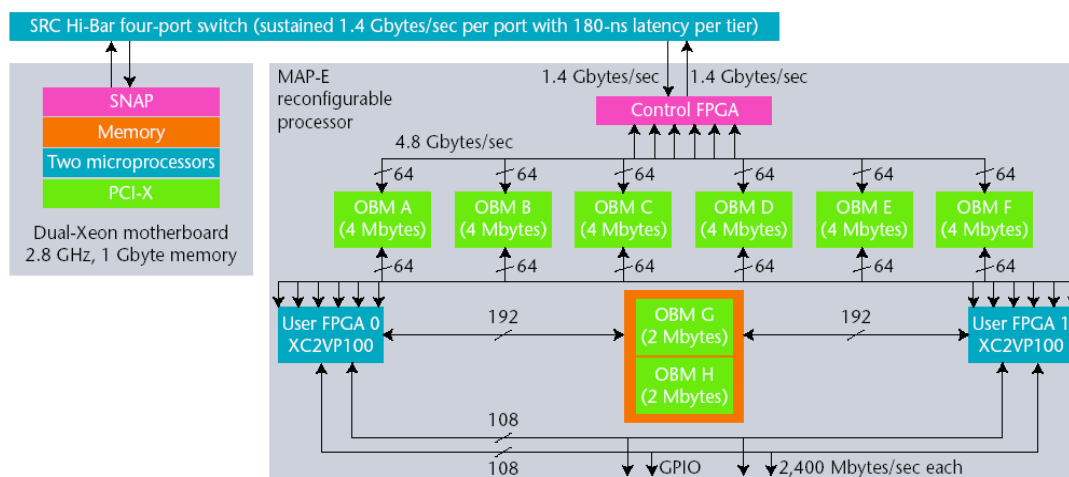


Figura 28 Arquitetura do hardware SRC-6 MAP

4.1.1 Usando Dispositivo FPGA para Acelerar Simulação Biomolecular

Um exemplo de desempenho desta plataforma é apresentado em [36]. A plataforma SRC MAPstation é utilizada para a análise de desempenho de *FPGAs* em simulações biomoleculares. O projeto foi desenvolvido utilizando linguagem de alto nível e como estudo de caso foi executado um método de simulação de dinâmica molecular. A plataforma da SRC disponibiliza uma pilha de software, permitindo que os algoritmos sejam escritos em linguagens tradicionais de alto nível, como Fortran ou C.

Para o estudo de caso, apenas uma parte do algoritmo de simulação, que compreende 80% do seu tempo de execução total, foi considerado. Para esta fração do algoritmo, a capacidade dos *FPGAs* da plataforma SRC permitiu uma aceleração da computação maior que 3x para dois sistemas biológicos da ordem de 24K e 62K átomos, utilizando uma aritmética de ponto flutuante precisão simples.

Uma importante contribuição apresentada neste trabalho foi a caracterização dos requisitos dos tempos de acesso à memória para os algoritmos em análise. Isto porque, as medidas de tempo foram divididas em três partes:

- ✓ tempo de inicialização,
- ✓ tempo de computação, e
- ✓ tempo de transferência dos dados, que é o tempo gasto para o *host* enviar os dados para o *FPGA* e tempo de busca dos dados de volta ao *host*.

Destes tempos, o terceiro foi o que causou maior impacto negativo no desempenho total.

Considerando somente o tempo de computação, chegou-se a uma aceleração de cerca de 3x. Entretanto, considerando o *overhead* na transferência dos dados, a aceleração foi menor que 1x, ou seja, houve uma degradação no desempenho. Realizando um estudo nos requisitos de acesso à memória, algumas técnicas foram avaliadas para redução dos tempos na transferência dos dados. As técnicas avaliadas foram as seguintes:

- ✓ Pré-busca e pré-armazenamento de dados para esconder as latências das transferências;
- ✓ Utilização de *pipeline*, enquanto dados buscados anteriormente estão sendo processados novos dados estarão sendo buscados;
- ✓ Analisar o padrão e comportamento de acesso aos dados e eliminar transferências desnecessárias.

A terceira abordagem foi empregada, porque ela consegue alavancar as outras. Com a utilização desta técnica, conseguiu-se manter a aceleração em 3,3x. Estas técnicas podem ser aplicadas em outros problemas para permitir mascarar (*overlapping*) os gargalos na comunicação entre host e *FPGA*.

4.2 Plataforma CRAY XD1

Cray XD1 [37] é uma das plataformas comerciais que combina computação reconfigurável com processador de propósito geral para alcançar aumento de desempenho. Trata-se de um cluster baseado no processador AMD *Opteron* [38] com uma interconexão proprietária *RapidArray* de alta velocidade. A unidade básica da sua plataforma é conhecida como *compute blade* e contém 2 processadores AMD *Opteron* e um *FPGA Xilinx Virtex II*. Cada *FPGA* tem acesso a quatro bancos de memória *SRAM QDR II* e podem acessar, também, a *DRAM* dos processadores *Opteron*, através dos processadores *RapidArray*, responsáveis por boa parte das funções de comunicação dentro do sistema Cray. A Figura 29 mostra as conexões do *FPGA* no sistema.

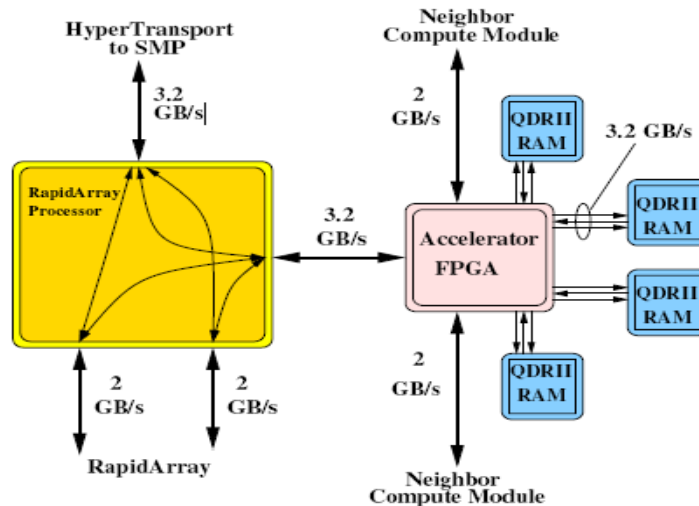


Figura 29 Conexões do módulo FPGA no Cray XD1

Um *switch crossbar RapidArray* permite a comunicação entre os *compute blade*, sendo que seis *compute blade* formam um chassis. Dentro dos chassis, os *FPGAs* são conectados em um *array* circular através de uma interface de comunicação de alta velocidade provida pelos *Transceivers Xilinx RocketI/O Multi-Gigabit* [39]. Os vários chassis são conectados usando os switches externos *RapidArray*. A Figura 30 apresenta a arquitetura da plataforma do Cray XD1.

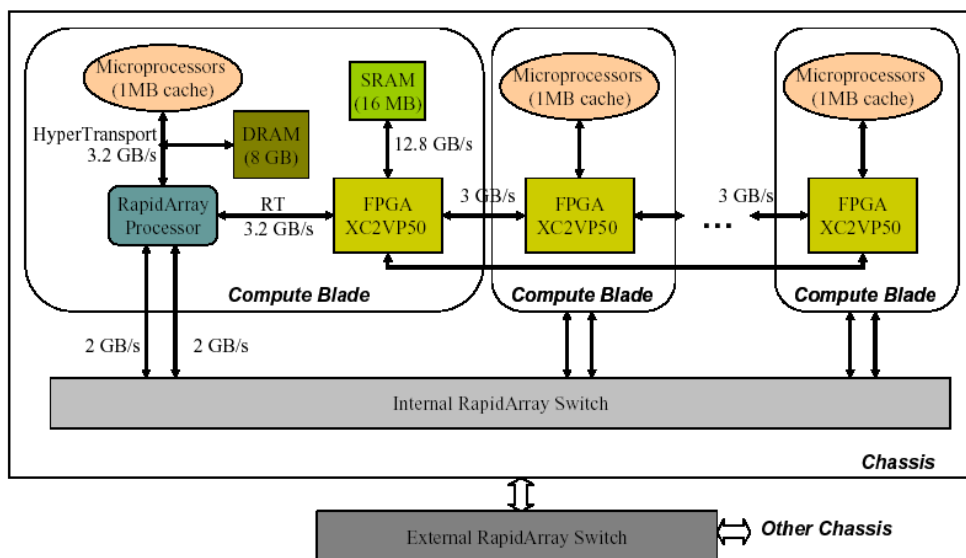


Figura 30 Arquitetura da plataforma Cray XD1

O sistema Cray é um sistema HPC (*High Performance Computer*), computação de alto desempenho, baseado na arquitetura DCP (*Direct Connected Processor*), onde dispositivos de memória e processadores estão diretamente conectados.

A arquitetura DCP do Cray [40] permite a otimização das aplicações de passagem de mensagens, ligando os processadores diretamente através de uma estrutura de interconexão de alto desempenho e eliminando, assim, a disputa em memória compartilhada e o *gargalo* do barramento. Com esta arquitetura, é possível integrar vários processadores em um sistema simples e unificado, alcançando os melhores níveis de desempenho das aplicações. A Figura 31 apresenta a arquitetura DCP do sistema Cray XD1.

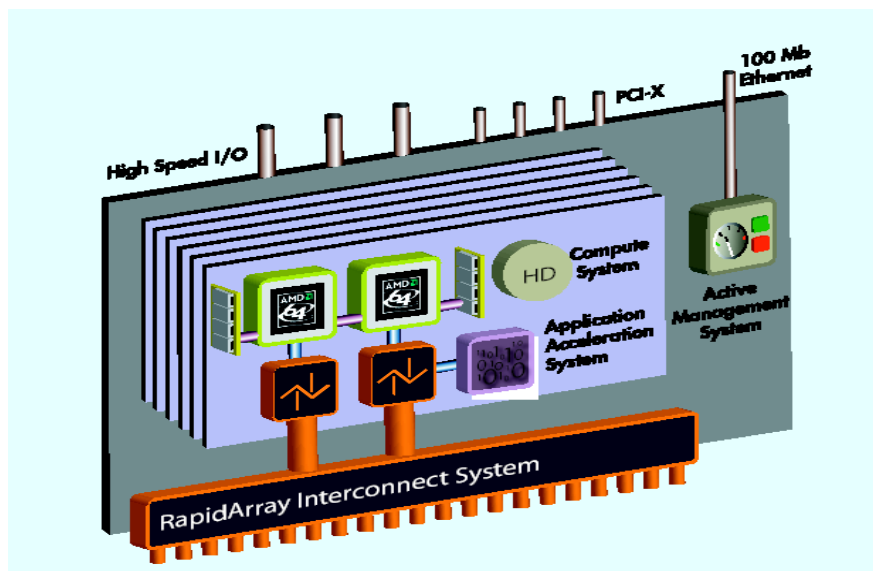


Figura 31 Arquitetura do processador DCP

A arquitetura do Cray XD1 inclui quatro subsistemas chaves:

- ✓ ambiente de computação,
- ✓ interconexão *RapidArray*,
- ✓ acelerador de aplicação, e
- ✓ gerenciamento ativo.

O ambiente de computação é composto pelos processadores AMD *Opteron* e o sistema operacional Linux. O motivo da escolha do processador *Opteron* pela Cray é o seu desempenho no acesso à memória. Como um processador de 64 bits, o *Opteron* consegue endereçar mais que um Terabyte de memória. Além disso, cada processador possui um controlador de memória embutido que provê uma largura de banda de 6,4 GB/s. Este desempenho escala linearmente para 12,8 GB/s quando dois processadores são acoplados. O uso do sistema Operacional Linux permite que os usuários tirem vantagem do grande número de aplicações código-aberto e comerciais.

Os processadores de comunicação *RapidArray* provêm a interface do *HyperTransport* [41] do processador *Opteron* com a interconexão *RapidArray*, criando um link de alta velocidade e baixa latência entre os processadores AMD. Permitem ainda, devido ao forte acoplamento dos processadores *Opteron* e switches, acelerar as funções de comunicação dos processadores AMD, liberando-os para executar as tarefas de computação e permitindo computação e comunicação concorrentes.

O subsistema acelerador de aplicações incorpora computação reconfigurável para alcançar uma maior aceleração das aplicações alvo. Cada um dos chassis Cray XD1 pode ser configurado com até seis processadores de aceleração de aplicação baseado nos *FPGAs Xilinx Virtex-II*. O recurso reconfigurável atua como um co-processador para os processadores AMD *Opteron*. Os algoritmos de computação intensiva têm grandes laços em seus procedimentos e por isto podem ter sua aceleração na execução paralela nesses dispositivos.

O sistema de gerenciamento ativo combina particionamento e características de inteligência de autoconfiguração para permitir que administradores visualizem e gerenciem centenas de processadores como um ou mais computadores lógicos. O particionamento divide o sistema Cray em computadores lógicos e os administradores operam estas partições. As características de autoconfiguração automaticamente traduzem detalhes de particionamento em detalhes de configurações do nó.

4.2.1 Scalable Hybrid Designs for Linear Algebra on Reconfigurable Computing Systems

Em [42], a plataforma XD1 foi avaliada para projetos de multiplicação de matrizes em ponto flutuante, considerada aplicação básica para computação científica e cujo desempenho é fundamental para muitas aplicações [43], [44], [45].

O foco deste trabalho foi o de investigar o desempenho das plataformas híbridas que utilizam *FPGA* e processador de propósito geral. Para isto foram considerados os problemas de particionamento hw/sw, a comunicação e cooperação entre *FPGAs* e processadores, o paralelismo dentro do *FPGA* e a escalabilidade para múltiplos nós. Foi proposto um projeto que pode ser executado em múltiplos nós do sistema, sendo o trabalho particionado dentro de cada nó usando um método de modelagem de desempenho baseado no balanceamento de cargas entre o *FPGA* e o processador. Além disso, a computação nos *FPGAs* e nos processadores foram coordenadas para evitar conflitos no acesso à memória.

O projeto alcançou uma aceleração de 1,4x quando comparado com execução apenas utilizando processador de propósito geral e de 2x quando comparado com projetos que

utilizam apenas *FPGA*. O resultado experimental alcançou 24,6 GFLOPs para a aplicação de multiplicação de matrizes utilizando seis processadores e seis *FPGAs*.

Dentre as principais conclusões do estudo estão o fato de que o desempenho da aplicação é escalável com o número de nós e de que o desempenho de cada nó pode ser aumentado com a utilização de unidades de ponto flutuante mais eficiente e/ou com o uso de dispositivos *FPGAs* maiores.

4.3 Conclusão

Os resultados para este trabalho mostraram o quanto o desempenho dos sistemas para as operações de produto escalar e de multiplicação vetor-matriz é dependente da largura de banda entre a memória e o *FPGA*. Neste sentido podemos afirmar que as operações de multiplicação vetor-matriz e produto escalar são considerados *I/O bound*, pois fazem uso intensivo de operações de entrada e saída. Mesmo que uma capacidade de processamento ilimitada seja disponibilizada, o desempenho destas operações é restringido pela largura de banda de memória disponível.

Diferente do que ocorre com as duas operações anteriores, que faz muito uso de acesso I/O, a operação de multiplicação de matrizes não é *I/O bound*, tendo em vista que cada elemento de uma operação $C_{n \times n} = A_{n \times n} * B_{n \times n}$ é utilizado n vezes. Sendo assim, o pico de desempenho deste projeto não é determinado pela largura de banda de memória disponível e sim pelos recursos de armazenamento disponíveis no *FPGA*, pela velocidade destes recursos e pela capacidade do algoritmo em reusar os dados.

[PAGINA INTENCIONALMENTE DEIXADA EM BRANCO]

Capítulo 5 – Ambiente de Desenvolvimento RASC

Neste capítulo é apresentado com mais detalhes o ambiente de desenvolvimento de projeto denominado *RASC*, alvo dos estudos na construção de *IP cores* para multiplicação de matrizes em hardware, associados a este trabalho.

5.1 Visão geral do Sistema RASC

O sistema *RASC* (*Reconfigurable Application Specific Computing*) [46], da empresa Silicon Graphics Inc, é composto pelo servidor SGI Altix 350 acoplado a uma ou mais placas aceleradoras RC100 construídas com *FPGA*. Neste sistema, considerado como a plataforma que melhor soluciona o problema de movimentação de dados [56], a interconexão entre os módulos é feita através da interface *NUMalink* [47] que provê uma largura de banda de 3,2 GB/s, conforme ilustra a Figura 32.

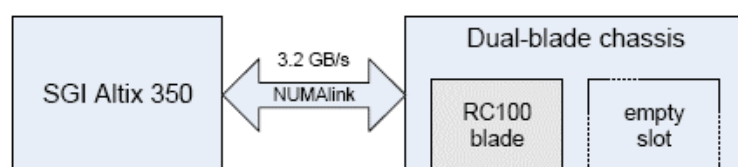


Figura 32 Sistema RASC

O SGI Altix 350 [48] compreende um processador *Intel Itanium2 dual core*, com 4 GBs de memória física. Já o módulo de hardware, o RC100, contém dois *FPGAs* computacionais, dois chips de I/O ponto a ponto (TIO) e um *FPGA* responsável pela carga dos *bitstreams* nos *FPGAs* computacionais.

Os dois chips *FPGAs* disponíveis ao usuário são da *Xilinx*, modelo *Virtex 4 LX200*. Cada um contém aproximadamente 200.000 células lógicas e trabalha com uma frequência máxima de 200 MHz. A Figura 33 apresenta um diagrama dos módulos de hardware. Os dois *FPGAs* computacionais são conectados aos respectivos chips de I/O via uma porta denominada SSP (*Scalable System Ports*) [49]. Cada *FPGA* é acoplado a até 5 bancos de memória QDR *SRAM* de 8MB cada, a uma largura de banda de 1.6 GB/s cada banco. As *SRAM* são configuradas como dois bancos para manter compatibilidade com a largura de banda da interface *NUMalink* que é de 3.2GB/s em cada direção.

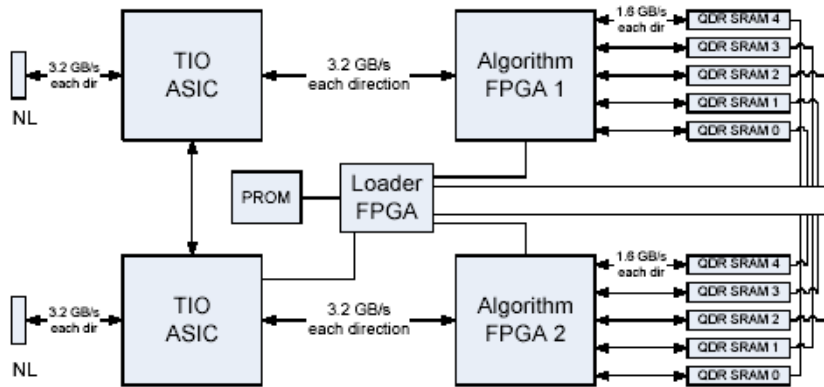


Figura 33 Módulos de hardware do RC 100

Apesar da quantidade limitada de bancos de memória, a arquitetura do *RASC* permite uma técnica de processamento em *streaming*, que reduz o *overhead* de transferência de dados por sobrepor a carga e descarga dos dados com o algoritmo em execução. Para o processamento *streaming* é necessário que o projetista defina previamente os bancos de memória, dentre os cinco bancos disponíveis, quais serão usados para leitura de dados e quais bancos de memória que serão usados para escrita de dados pelo algoritmo e pelo aplicativo. A Figura 34 apresenta um exemplo de segmentação de memória.

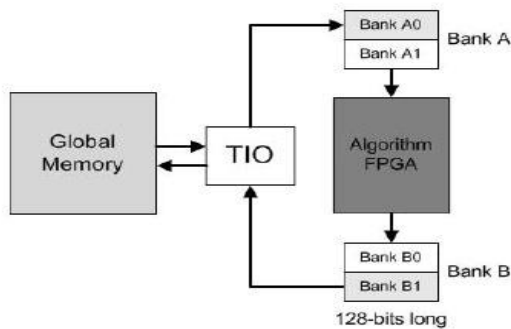


Figura 34 Exemplo de segmentação de memória

Na segmentação ilustrada na figura 34, dois bancos físicos de memória A0 e A1 formam um banco lógico A e são definidos como bancos de leitura (que serão lidos pelo algoritmo), outros dois bancos físicos de memória B0 e B1 formam um banco lógico B e são definidos como bancos de saída (que serão escritos pelo algoritmo). Com esta divisão, o algoritmo pode processar dados do banco A1 e escrever os resultados no banco B0, enquanto os próximos dados de entrada estão sendo escritos no banco A0 e os resultados previamente processados estão sendo descarregados do banco B1 para o *host*. Quando todos os dados do

banco A1 tiverem sido processados, o *FPGA* inicia a execução do segmento carregado (banco A0), enquanto o segmento A1 será carregado com os próximos dados de entrada.

Outra técnica disponível no *RASC* é a escalabilidade automática sobre múltiplos *FPGAs*. Um conjunto de dados pode ser automaticamente particionado e enviado a múltiplos *FPGAs* executando *bitstreams* idênticos. Ao final do processamento os resultados formarão, automaticamente, um único conjunto de dados.

No *RASC*, uma porção de cada chip é dedicada à lógica do *RASC Core Services* que é responsável, entre outras coisas, por prover uma interface entre o chip *TIO* e as *SRAM* ligadas ao *FPGA*. O restante do recurso do *FPGA* é dedicado à lógica do usuário. A Figura 35 ilustra um *FPGA* computacional com o algoritmo do usuário e os módulos do *RASC Core Services*.

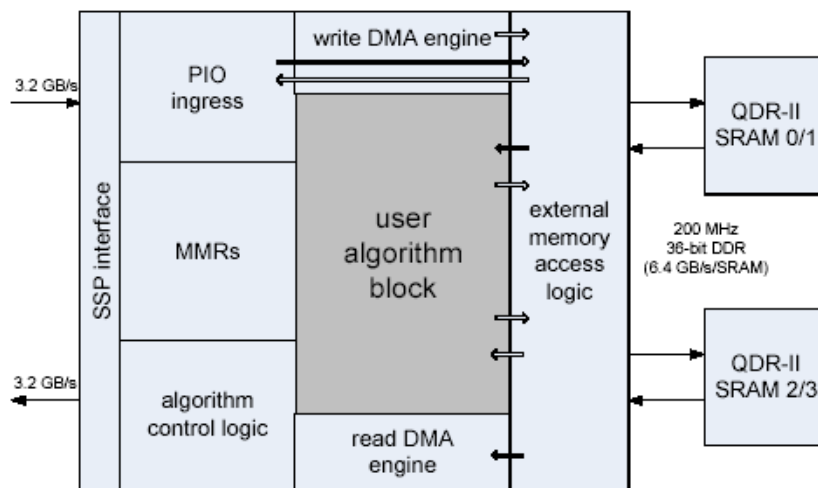


Figura 35 Algoritmo do usuário e módulos do Core Services

Além da interface SSP e a lógica para acessar os bancos de memória *SRAM*, o *RASC Core Services* também contém registradores mapeados em memória (MMRs) que podem ser acessados pela lógica do usuário.

Para obter o desempenho esperado nos sistemas híbridos é necessário que o mesmo possua uma interface com grande largura de banda e de baixa latência entre o processador de uso geral e o *FPGA*. No caso do *RASC*, esta interface é mantida pelo barramento de interconexão *NUMALink 4*. Com a conexão dos *FPGAs* ao *NUMALink*, o *RASC* coloca os recursos do *FPGA* dentro do domínio do sistema de computação, garantindo uma grande largura de banda, de 3,2 GB/s em cada direção e uma baixa latência. O *NUMALink* se conecta aos *FPGA* através do módulo de comunicação rápida que é baseado no chip *TIO*, como mostrado na Figura 36. Este chip suporta a interface SSP (*Scalable System Port*) que é usada

para conectar os *FPGA* ao sistema Altix, provendo uma interface de grande largura de banda e baixa *latência* para cada um dos *FPGA*.

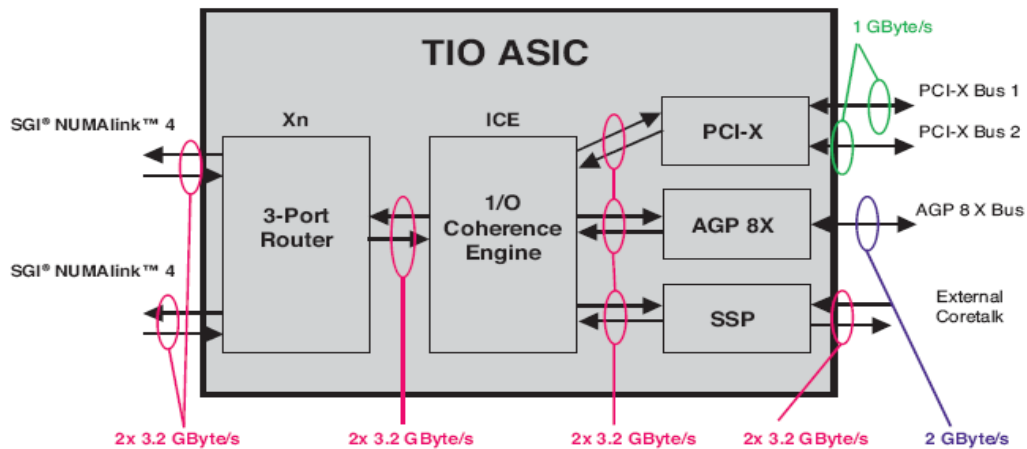


Figura 36 Esquema Interno do chip TIO

O ambiente de desenvolvimento de projeto do *RASC* inclui, também, um conjunto de *API* através do *RASCAL* (*RASC Abstraction Layer*) camada de abstração do *RASC*. Esta camada permite a abstração do hardware para prover vários níveis de escalabilidade, além do controle direto sobre cada elemento de hardware no sistema [52].

Também faz parte do ambiente *RASC* o *SSP Stub* que é uma ferramenta da verificação planejada para auxiliar o usuário a simular e verificar o algoritmo antes de ser carregado e executado no *FPGA*. Usado com o *VCS/Virsim* (Verilog Compiler Simulator) o *SSP Stub* permite que o usuário simule o envio e recepção pacotes de dados para e do algoritmo, como também iniciar e parar o algoritmo ou verificar o status da simulação. No *RASC* também existe uma versão do *GNU Debugger* (GDB) que é baseado na versão *GDB* do Linux e contém extensões para gerenciamento de vários *FPGA*.

5.2 Core Services

Para viabilizar a criação de projetos no *RASC* e para facilitar a execução do algoritmo do usuário no mesmo, a SGI desenvolveu em linguagem de descrição de hardware Verilog, o conjunto de controladores que permitem que o algoritmo do usuário acesse corretamente os recursos do sistema, o *RASC Core Services* [53]. Esse conjunto de controladores ocupa uma fração do *FPGA* e permite, entre outras coisas, a comunicação do algoritmo do usuário com a memória e com a porta *SSP*. Um algoritmo desenvolvido para ser executado no *RASC* deve ser integrado ao *RASC Core Services* para acessar todos os recursos disponíveis do sistema.

O *RASC Core Services* contém a implementação da interface física e do protocolo *SSP*, controle e geração de relógio global para o algoritmo, portas de leitura e escrita para cada um dos blocos físicos ou lógicos de memória, controle para execução passo a passo do algoritmo, *DMA* (acesso direto à memória) para leitura e escrita de dados, acesso as portas de debug e registradores do algoritmo, registradores de status, de controle e lógica de interrupção. A Figura 37 ilustra o diagrama em blocos do *RASC Core Services*.

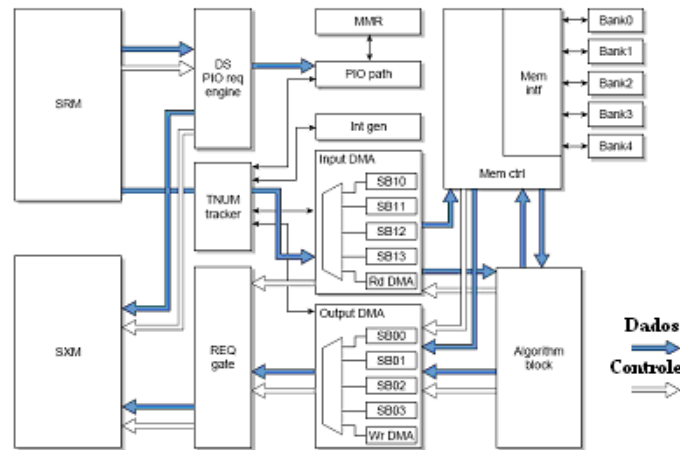


Figura 37 Diagrama de blocos do *RASC Core Services*

Os blocos *SRM* (*SSP Receive Module*) e *SXM* (*SSP Transmite Module*), representados na Figura 37, constituem, respectivamente, as lógicas de recepção e transmissão do protocolo *SSP*. O *SRM* está ligado ao bloco *PIO Request Engine*, o qual manipula os pedidos de escrita do *host* e requisições de leitura para palavras de 64 bits. O *SXM* está ligado ao bloco *REQ Gate* que forma os pacotes de escrita do *FPGA* e os pacotes de pedido de leitura na memória principal.

O *Input DMA* contém até quatro máquinas *streams* de *DMA* que transferem dados vindos da memória principal para o algoritmo e uma máquina que transfere para a *SRAM* os dados vindos da memória principal.

O *Output DMA* contém até quatro máquinas *streams* de *DMA* que transferem dados do algoritmo para a memória principal e uma máquina de escrita que transfere os dados da memória *SRAM* para a memória principal.

O bloco *MMR* (*Memory Mapped Register*) contém os registradores usados para controle e uso do *FPGA*.

O *Int gen* gera um pacote de dados proprietário para interromper o *host* caso uma transferência de *DMA* tenha sido concluída ou o algoritmo do usuário tenha finalizado.

5.3 Fluxo de Projeto na Plataforma *RASC*

Uma vez conhecidas a arquitetura e funcionalidade do *RASC Core Services* e como os seus blocos interagem com o algoritmo, podemos definir uma metodologia para desenvolvimento de projetos no *RASC*. A primeira característica do projeto a ser definida é a alocação de memória. É necessário determinar quais bancos de memória serão alocados para armazenar os operando de entrada e os operando de saída do algoritmo. Uma recomendação dada pela SGI, para distribuição de memória, é que os dados de entrada e saída pertençam a bancos de memória diferentes. Esta distribuição é importante quando o projeto utiliza *DMA*. Fazendo deste modo, o algoritmo sempre lê de um banco de memória em que o controlador de *DMA* faz sua escrita com o dado vindo da memória do *host*. O mesmo raciocínio é usado para a operação contrária, no banco que é escrito pelo algoritmo e lido pelo *DMA*.

Como foi visto no capítulo 4, um banco de memória lógica é formado por dois bancos de memória físicos de 8 MB cada. Os bancos de memória lógica podem ser particionados em segmentos para melhor organização dos dados. O tamanho de cada segmento pode variar de 16 KB até metade da capacidade de armazenamento do banco lógico. Uma vez definida a distribuição de memória, o projetista deve desenvolver o seu algoritmo. Para aplicações que processam grande quantidade de dados, se recomendada o uso da multi-buferização de dados, provendo um contínuo e paralelo fluxo de dados de e para o algoritmo. Os dados são carregados, processados e descarregados como em uma arquitetura *pipeline*.

Após o desenvolvimento do algoritmo, o projeto pode ser integrado ao *RASC Core Services*. Nesta integração, o projetista atribui os sinais de *debug* às variáveis que devam ser analisadas e associa as portas de acesso à memória aos sinais do controlador de memória. A partir da integração, o projeto já está pronto para ser simulado no ambiente VCS [54]. Esta simulação garante que os protocolos de comunicação com o *RASC Core Services* estão sendo respeitados e torna possível a visualização dos sinais do algoritmo através da ferramenta Virsim [55].

Após a validação na simulação, e antes da implementação em *FPGA*, é necessário a inserção, no projeto, de diretivas especiais que permitam que informações sobre a organização dos dados sejam transmitidas para a camada de software. Essas informações são passadas através de diretivas *extractor*, ou comentários em *Verilog* ou *VHDL*. A partir dessas diretivas um Script *python* é executado e são gerados dois arquivos de configuração que contém as informações necessárias para que a camada de software comunique com o hardware corretamente. Esses arquivos são chamados *core_services.cfg* e *user_space.cfg*.

O arquivo *core_services.cfg* contém, por exemplo, os endereços das portas de memória e das portas de debug utilizadas no projeto. O arquivo *user_space.cfg* contém a especificação dos arrays utilizados pelo algoritmo, como, tamanho, banco de memória associado, tipos de dados utilizados. A Figura 39 mostra a declaração de dois arrays de entrada de tamanho 16KB, localizados na *SRAM0*. Esses arrays são organizados em 2048 palavras de 64 bits.

```
##Array name
##  # of elements in array
##      Bit width of element
##
##          SRAM location
##
##              Byte offset (within given SRAM)
##
##                  Direction
##
##                      Type
##
##                          Stream flag
// extractor SRAM:input_a 2048 64 sram[0] 0x000000 in unsigned stream
// extractor SRAM:input_b 2048 64 sram[0] 0x004000 in unsigned stream
```

Figura 39 Exemplo de extractor na declaração de arrays de entrada

Após a inserção dos comentários *extractor* e da geração dos arquivos de configuração, o projeto deve ser sintetizado e poderá ser enviado para o *FPGA*. A Figura 40 sintetiza o fluxo de desenvolvimento do algoritmo no *RASC*.

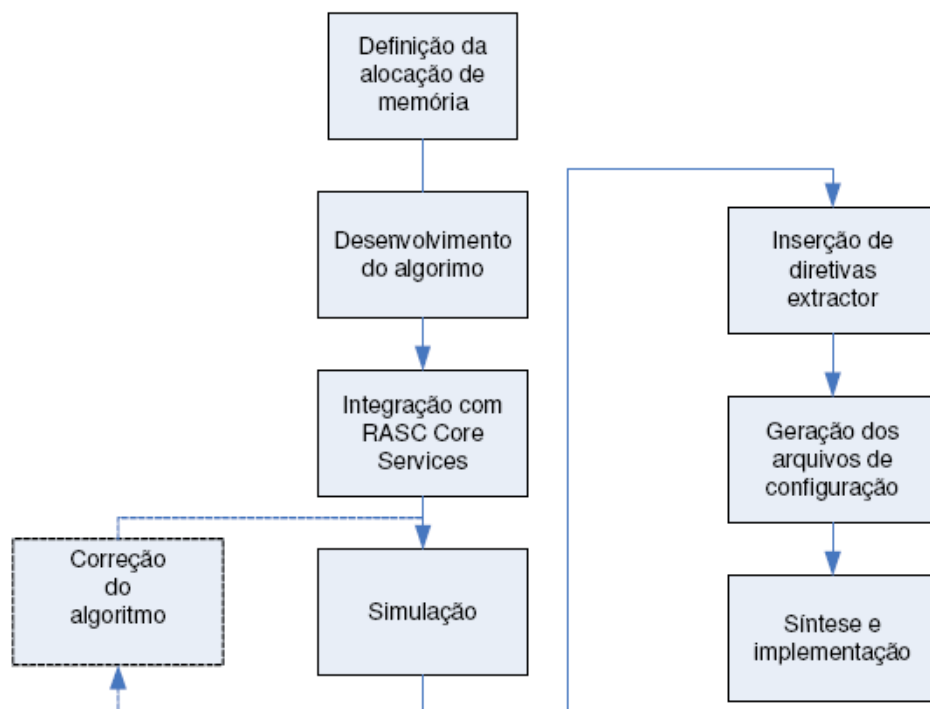


Figura 40 Fluxo de desenvolvimento de projeto no RASC

5.4 Camada de Abstração do RASC

A camada de abstração do RASC provê uma interface de programação da aplicação (*API*) com funções padrões *open*, *close*, *read*, *write* e *ioctl* para o aplicativo acessar através do *kernel* os recursos do hardware. Esta camada é implementada em dois níveis: o nível mais baixo, denominado nível *COP* (COProcessador), provê chamadas de funções para dispositivos individualmente, ou seja, pode-se dirigir uma chamada, por exemplo, de leitura (*read*), para um específico dispositivo. O nível mais alto, denominado nível de algoritmo, trata uma coleção de dispositivos como um único. A camada de abstração gerencia a movimentação de dados de e para os dispositivos e a distribuição de tarefas através dos múltiplos dispositivos, permitindo a escalabilidade. A Figura 41 mostra um diagrama em blocos dos níveis de abstração do projeto do RASC.

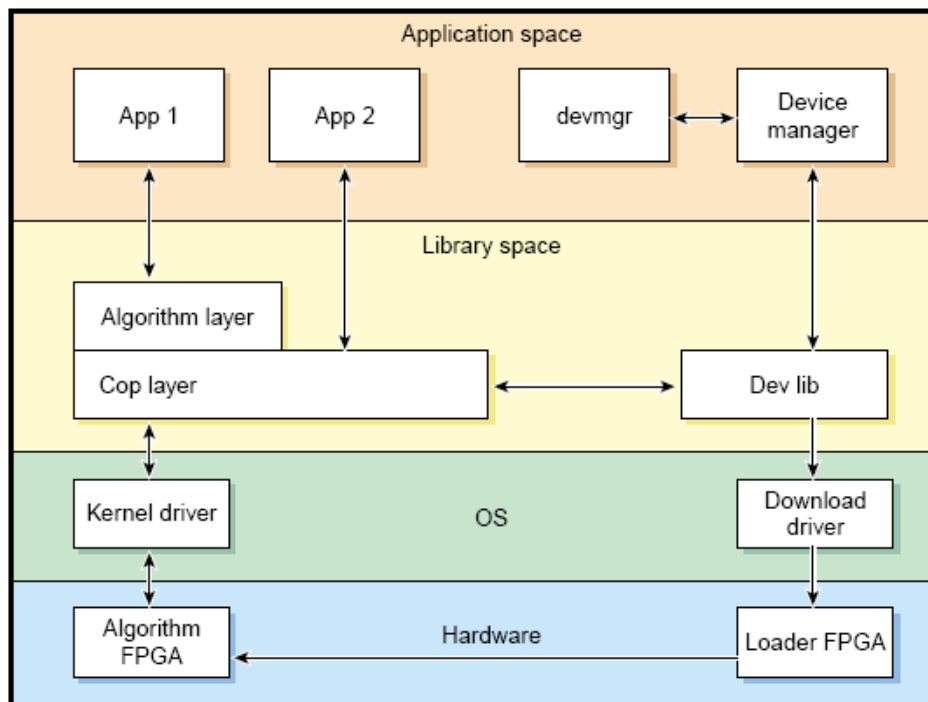


Figura 41 Camadas de abstração do RASC

As funções da camada de abstração do RASC estão disponíveis em linguagem em C e Fortran90 e devem ser chamadas de dentro da aplicação desenvolvida pelo usuário. Estão disponíveis funções para reserva e configuração dos dispositivos, associação dos dispositivos configurados como um único dispositivo lógico, inicialização da computação, envio de dados para o dispositivo, recepção de dados vindos do dispositivo, entre outras.

5.5 Ambiente de Simulação VCS/VIRSIM

A SGI disponibiliza um ambiente de verificação para os projetos desenvolvidos no RASC. Este ambiente chamado de *SSP Stub*, consiste em módulos escritos em *Verilog* que permitem a simulação de envio e recepção de pacotes *SSP* de e para o algoritmo do usuário. Neste ambiente é possível inicializar os modelos de simulação de memória através de arquivos de inicialização denominados *arquivos.dat*, e, também, ao fim da simulação, extrair as informações dos modelos de memória de saída.

Durante a execução do *Testbench*, o *SSP Stub* recebe pacotes de comandos *SSP* lidos de um arquivo de entrada, chamado diagnóstico. Um arquivo diagnóstico contém comandos para inicialização do *Core Services*, inicialização do algoritmo, operações de leitura de *DMA* para envio de dados para o *FPGA*, operações de escrita de *DMA* para recepção dos dados do *FPGA*, checagem de status e verificação de *flag* de erros e *pooling* para verificar se a operação no algoritmo foi finalizada.

A simulação é executada no simulador *VCS* da *Synopsys* [54]. A cada finalização do diagnóstico, um arquivo chamado *vcdplus.vpd* é gerado no diretório do *Testbench*. Este arquivo *vpd* é utilizado como entrada para visualização das formas de onda na ferramenta *Virsim* da *Synopsys* [55]. A Figura 42 mostra o ambiente de simulação de projeto para o RASC.

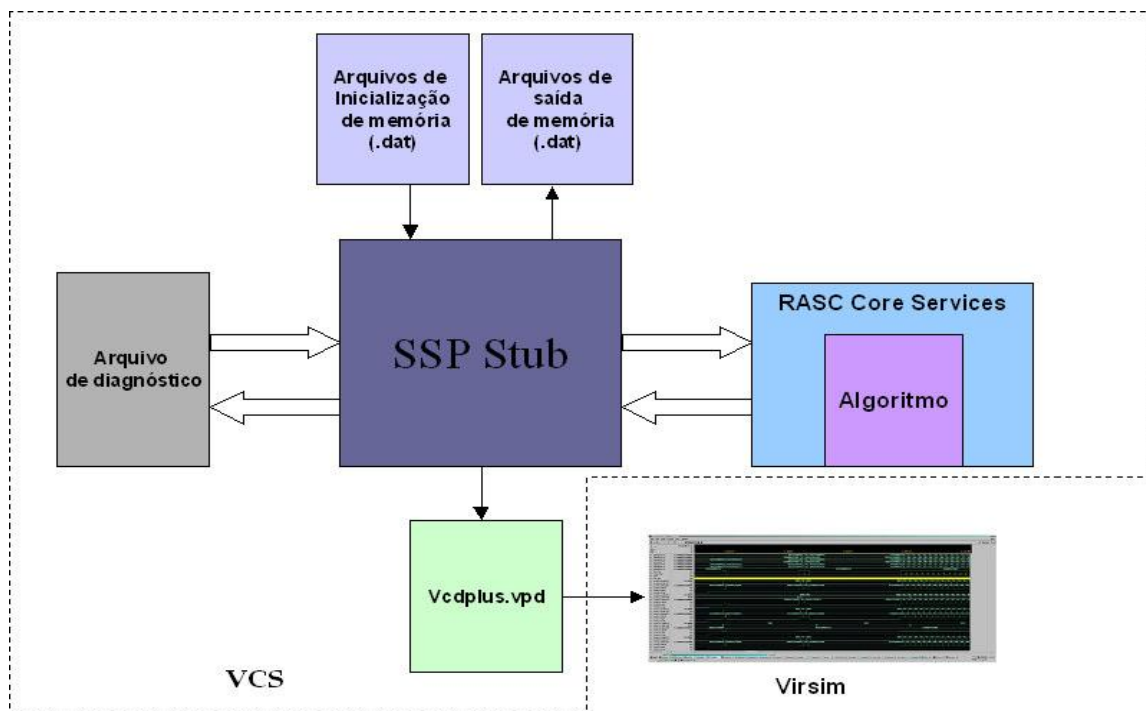


Figura 42 Ambiente de simulação do RASC

5.6 Avaliação da Metodologia Mitrion-C em Computação Reconfigurável de Alto Desempenho.

Em [50], o desempenho da plataforma *RASC* foi avaliada para algumas funções básicas da computação científica. Dois algoritmos foram construídos, sendo o primeiro sobre multiplicação de densas matrizes-vetor em ponto flutuante de precisão simples (DMVM – Dense Matrix-Vector Multiplication), e o segundo estudou as condições de limite esféricas na dinâmica molecular (SB – Spherical Boundary). Nesta avaliação foi utilizada uma linguagem de alto nível, *Mitrion-C* [51], para desenvolver o algoritmo. O ambiente de desenvolvimento Mitrion, da empresa Mitronics AB, inclui o Mitrion Virtual Processor (MVP) e Mitrion Software Development Kit com Mitrion-C compilador. Neste ambiente o desenvolvedor tem todas as ferramentas para desenvolver software paralelo, simular e fazer depurações (debugger) do código. A Figura 43 apresenta o fluxo de desenvolvimento de projeto com a plataforma Mitrion.

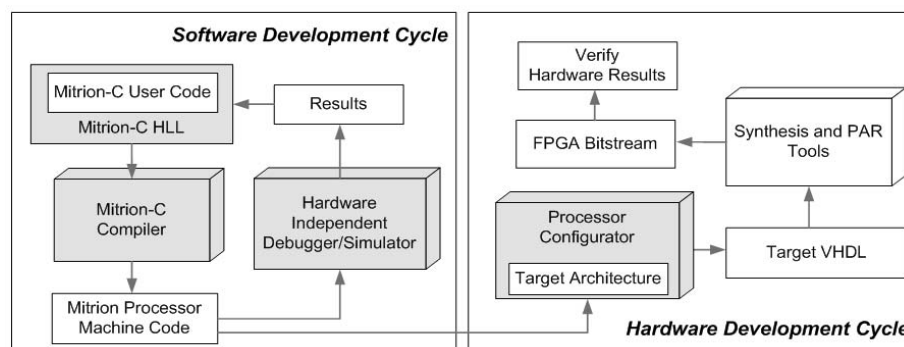


Figura 43 Fluxo de projeto da Mitrion

O sistema Altix 350 é composto por oito processadores *Itanium2* com frequência de trabalho de 1,5 GHz e 16 GB de memória compartilhada conectados pela rede *NUMALink* ao hardware. O hardware possui duas placas aceleradoras RC 100 com dois *FPGAs Virtex-4 LX200* e 5 bancos de memória, de 8 MB por banco, acoplados a cada um dos *FPGA*. A frequência de relógio de operação dos *FPGA* foi de 100 MHz.

A multiplicação de matrizes 2048 x 2048 foi executada como 2048 operações de multiplicação matriz-vetor. Os bancos de memória foram divididos de forma que dois bancos físicos formem um banco lógico. A Figura 44 mostra a formação dos bancos lógicos.

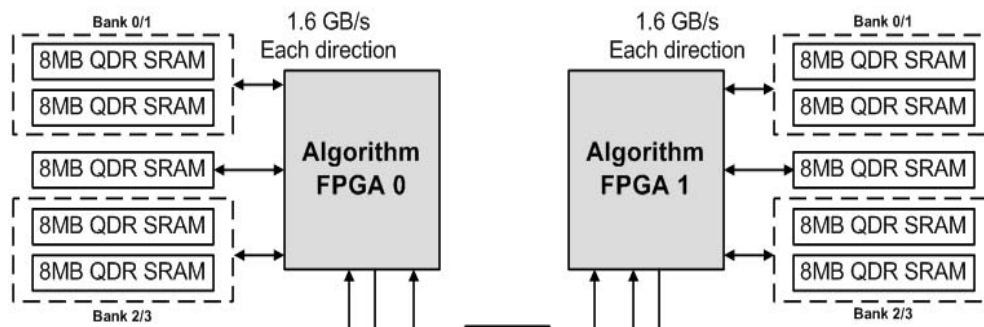


Figura 44 Formação dos bancos lógicos de memória

Em um dos bancos lógicos é armazenada a matriz e em outro banco um vetor coluna. Com esse particionamento da memória, o tamanho da palavra lida de cada banco de memória é de 128 bits, sendo possível, a cada ciclo de relógio, a leitura de 4 palavras da matriz e 4 palavras do vetor coluna. Deste modo, dada a limitação da largura de banda e visando explorar o paralelismo, foram implementados 4 multiplicadores em *pipeline*, que realizam a cada ciclo uma porção da multiplicação de uma linha da matriz pelo vetor coluna. Para calcular um elemento é necessária a acumulação de 512 parcelas dessa multiplicação. A Figura 45 mostra o esquema de particionamento e multiplicação das submatrizes.

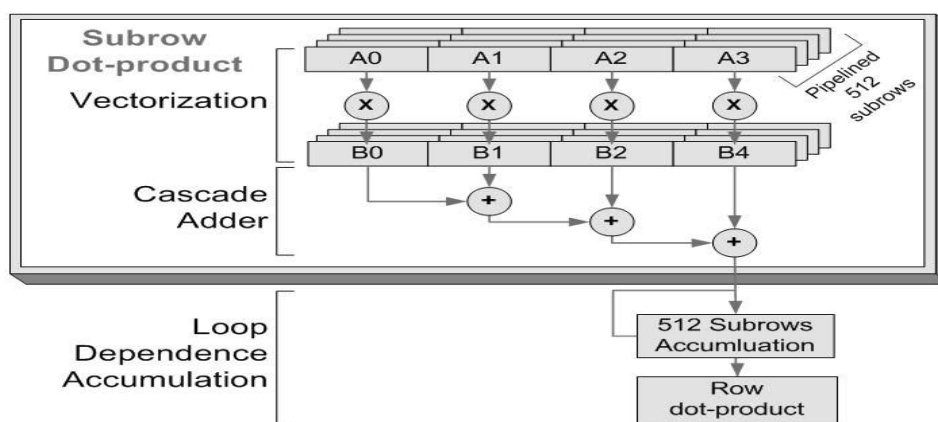


Figura 45 Esquema de particionamento e multiplicação

Com esta arquitetura e a utilização de um *FPGA* o projeto alcançou uma aceleração de 21 vezes quando comparado a um processador Itanium2.

Procurando melhorar o desempenho, um segundo projeto foi desenvolvido com duas otimizações do código *Mitrion-C*. Primeiramente foram paralelizadas as multiplicações, porque agora iria usar mais de um *FPGA*. Em segundo lugar os 3 níveis de soma foram reduzidos para dois níveis. Deste modo, os quatros resultados de saída em paralelo do multiplicador passaram a ser 16 fazendo com que as 512 parcelas da multiplicação fossem reduzidas a 128. A Figura 46 mostra a otimização feita na estrutura do multiplicador.

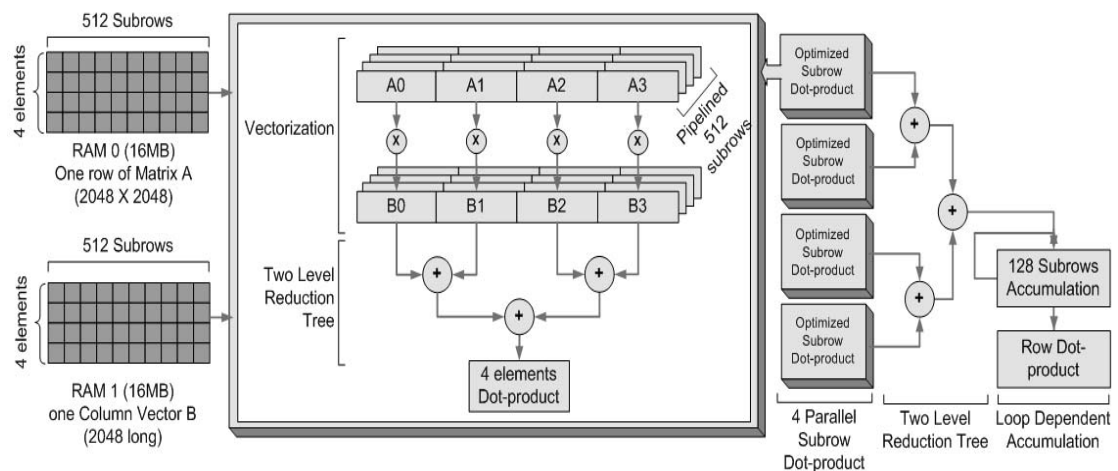


Figura 46 Esquema otimizado para maior desempenho

O projeto otimizado foi escalonado para 1, 2 e 4 *FPGA* alcançando uma aceleração de 80 vezes para 4 *FPGA* com o algoritmo da multiplicação matriz-vetor. Neste algoritmo, a razão de aceleração foi de 2x, quando passou a usar 2 *FPGAs*, enquanto o algoritmo *SB* nesta mesma situação teve um aumento de aceleração de 1,6x, como está demonstrado nas Tabelas 1 e 2.

Tabela 1 Tempo de execução de Software e Hardware

	DMVM	SB
Software	0.900 s	1.607 s
Non-optimized Hardware	0.105 s (8× Speedup)	-
Optimized Hardware	0.042 s (21× Speedup)	0.158 s (10× Speedup)

Tabela 2 Tempos de execução com vários hardware

Number of FPGAs	DMVM	SB
1	0.042 s (21× speedup)	0.158 s (10× speedup)
2	0.022 s (41× speedup)	0.101 s (16× speedup)
4	0.011 s (80× speedup)	0.061 s (26× speedup)

Também em [50] a diferença de aceleração entre os dois aplicativos está no fato de que multiplicação de matriz vetor não é *I/O bound* tanto quanto o algoritmo *SB*, que faz uso de entradas e saídas cada iteração do laço interno.

5.7 Conclusão

Neste capítulo, foi apresentada uma visão geral do *RASC* com algumas de suas características mais importantes. Estas características devem ser levadas em conta no momento do desenvolvimento dos projetos para esta plataforma.

Na plataforma *RASC*, o uso de memórias lógicas formadas por dois bancos físicos proporciona ao desenvolvedor a possibilidade de poder fazer um *pipeline* com os dados sendo armazenados pelo aplicativo em um segmento e o algoritmo buscando dados para processar em outro segmento ao mesmo tempo.

O *Core Services* provê uma camada de abstração sobre as maneiras de acessar os recursos disponíveis na plataforma. Através dele o projetista não precisa se preocupar com tempos de acesso á memória, ou seja, basta pedir um dado para a porta de requisição e no tempo certo, após ser avisado, o dado será entregue. E para ter certeza de que o algoritmo irá funcionar quando for prototipado, o ambiente de teste disponibiliza uma grande quantidade de recursos para conferir se o algoritmo está dentro dos padrões exigido pelo *FPGA* em questão.

Em sua nova versão 2.20, o *RASC* possui uma interface gráfica para auxiliar o desenvolvedor no momento de criar um projeto.

Capítulo 6 - Estratégias para Particionamento de Grandes Matrizes

Neste capítulo é abordada a estratégia de particionamento de dados de grandes matrizes, utilizada nesta dissertação.

6.1 Metodologia de Desenvolvimento

O método de particionamento desenvolvido neste trabalho foi proposto por [57] em 1995 e é denominada *PCAM* (*Partitioning, Communication, Agglomeration e Mapping*).

Esta estrutura metodológica é composta por quatro fases, a saber: particionamento, comunicação, aglomeração e mapeamento. As duas primeiras fases têm foco na concorrência e na escalabilidade. Nas duas fases finais o foco é dirigido à localidade e outros aspectos relacionados ao desempenho. A Figura 47 sintetiza os quatro estágios desta metodologia.

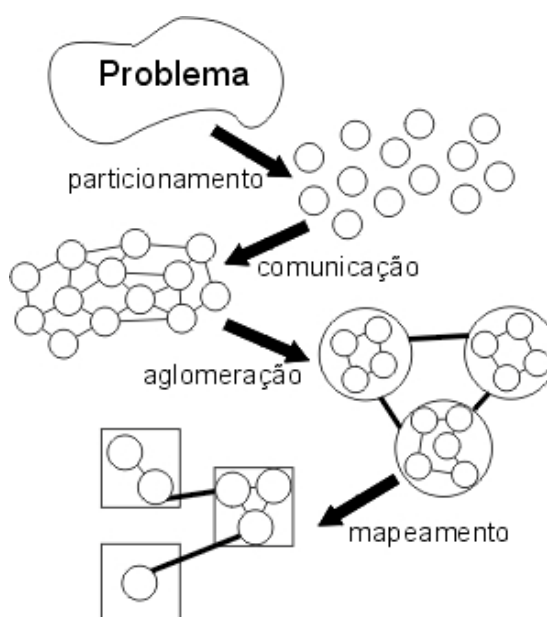


Figura 47 Estrutura da Metodologia PCAM - Foster, 1995

Cada fase desta pode ser descrita como:

- ✓ **Particionamento** - a computação a ser executada e o seu respectivo conjunto de dados, devem ser decompostos em pequenas tarefas. Como a abordagem é genérica, ignoram-se questões como o número de processadores e características específicas de máquinas em particular. O enfoque neste passo é reconhecer as oportunidades para execução paralela.

- ✓ Comunicação - nesta fase determina-se a comunicação necessária para coordenar a execução das tarefas e as estruturas mais apropriadas para a realização desta tarefa.
- ✓ Aglomeração - avaliam-se as estruturas das tarefas e das comunicações, definidas anteriormente, com respeito às exigências de desempenho e aos custos de implementação. Havendo necessidade, as tarefas podem ser combinadas em tarefas maiores para aumentar o desempenho e/ou reduzir os custos de projeto/implementação.
- ✓ Mapeamento - cada tarefa é endereçada para um processador com o intuito de satisfazer os objetivos de maximizar a utilização de cada um deles e minimizar os custos de comunicação. Este mapeamento pode ser especificado de forma estática, ou determinado em tempo de execução, por algoritmos de balanceamento de carga.

Esta metodologia é bastante apropriada para os propósitos de desenvolvimento de algoritmos de multiplicação de matrizes em hardware. Na multiplicação de matrizes, para o particionamento, levamos em conta o tamanho do armazenamento disponível no algoritmo em hardware. A comunicação deverá ser minimizada. No nosso caso não haverá comunicação entre os elementos de processamentos existentes no algoritmo. Cada elemento de processamento receberá parte dos dados que lhe forem endereçados e executará as transformações sobre eles independente dos outros elementos de processamentos. A quantidade de dados e a quantidade de elementos de processamentos também definirão o tamanho do particionamento podendo ser agrupados mais dados ou não, para aumentar o desempenho. Na multiplicação de matrizes o mapeamento é feito de forma estática, ou seja, antes de iniciar, o algoritmo recebe seus dados.

6.2 Computação Intensiva

A estratégia de particionamento baseado em um modelo *hardware/software codesign* [58] é determinada por suas necessidades de entrada e saída (I/O), sua capacidade de armazenamento, entre outros atributos, e pela complexidade das operações de suas tarefas.

A complexidade de um algoritmo consiste na quantidade de trabalho necessário para a execução de uma tarefa, expressa em função das operações fundamentais que variam de acordo com o algoritmo e com o volume de dados que este processa. Com o aumento na complexidade e no tamanho dos problemas ora sob demanda, em várias áreas do conhecimento, torna-se cada vez mais importante o desenvolvimento de algoritmos mais eficientes, aliados a máquinas com maior desempenho computacional.

Esta complexidade computacional poderá determinar se a implementação de um determinado algoritmo é ou não viável, para uma determinada arquitetura, em função de seus requisitos de desempenho.

A importância da complexidade pode ser observada no exemplo na tabela abaixo, que mostra 5 algoritmos, de A_1 a A_5 , para resolver problemas de mesmo tamanho, porém com diferentes complexidades. Supomos, para este exemplo, que uma operação leva 1 milésimo (ms) para ser efetuada. A tabela 3 mostra o tempo necessário para cada um dos algoritmos.

Tabela 3 Tempos de execução variando a entrada e a complexidade

n	A_1 $T_1(n) = n$	A_2 $T_2(n) = n \log n$	A_3 $T_3(n) = n^2$	A_4 $T_4(n) = n^3$	A_5 $T_5(n) = 2^n$
16	0.016s	0.064s	0.256s	4s	1m4s
32	0.032s	0.16s	1s	33s	46 Dias
512	0.512s	9s	4m22s	1 Dia 13h	10^{137} Séculos

Fonte:

A complexidade computacional pode ainda ser classificada como: complexidade espacial, representada pelo espaço de memória que o algoritmo usa quando em execução e temporal, que é o mais usado e se divide em tempo real, tempo necessário para a execução do algoritmo, e do número de instruções necessárias para a sua execução.

Para o estudo da complexidade são usadas três perspectivas: pior caso, melhor caso e caso médio. No nosso trabalho, para análise de como particionar um algoritmo de multiplicação de matrizes entre componentes de software e de hardware, tomaremos como exemplo a análise do pior caso.

Este método é normalmente representado pela notação $O(\)$ [59]. Por exemplo: sejam f e h funções reais positivas de variável inteira n . Diz-se que f é $O(h)$ – representando desta forma $f = O(h)$, quando existir uma constante $c > 0$ e um valor inteiro n_0 , tal que

$$n > n_0 \Rightarrow f(n) \leq c \cdot h(n)$$

Ou seja, a função h atua como um limite superior para valores assintóticos da função f .

Quando se considera o número de passos efetuados por um algoritmo, as constantes aditivas e multiplicativas podem ser desprezadas. Também, os termos de menor grau serão desconsiderados, levando em conta somente o termo assintótico que represente o maior crescimento da função. Assim, em um algoritmo que possua um número de passos igual a n^2

+ n , será aproximado para n^2 . Se houver um valor de passos igual a $6n^3 + 4n - 9$ serão transformados em n^3 . A análise do pior caso é o método mais usado sendo o mais fácil de determinar.

Quando as tarefas têm operações de baixa complexidade assintótica, como $O(1)$ ou $O(n)$, por exemplo, estas tarefas são apropriadas para ser executadas em um processador de propósito geral. Isto porque são tarefas de baixo custo computacional. Por outro lado, quando as tarefas possuem operações de alta complexidade como $O(n^2)$ ou $O(n^3)$, operações consideradas computacionalmente intensivas, sugere-se que componentes, como co-processadores especiais, possam ser usados na execução de módulos intensivos em processamento.

6.3 Algoritmo de Particionamento dos Dados

Para conseguir a necessária reutilização dos dados na memória local, vários métodos para a computação envolvendo matrizes e outros arrays de dados foram desenvolvidos [60]. Tipicamente, um algoritmo que trata de grandes problemas, como grandes matrizes, onde existe o processamento massivo de dados, pode ter uma versão distribuída que opera na forma de clusters, processando subarrays de dados, denominados de blocos ou submatrizes. A vantagem dessa abordagem é que os pequenos blocos podem ser movidos e processados com maior eficiência em níveis de memória mais rápidas (memória cache), com um maior reuso de dados. Esta sugestão requer, no entanto, mecanismos de controle posterior para recuperação do resultado final do algoritmo.

A técnica *strip mining* é usada para transformar um algoritmo seqüencial em um algoritmo por bloco [60]. Uma seqüência de transformação é feita nos laços do algoritmo seqüencial, colocando mais um laço externo junto a cada laço existente. Cria-se, desta maneira, uma nova variável para percorrer as linhas (ou colunas) até o tamanho da matriz original. Neste laço externo as linhas e colunas da matriz são particionadas em blocos do tamanho que se deseja. Este tamanho é o passo que cada laço executará após cada iteração. Abaixo segue um exemplo de uma linha de laço externo ao laço i do algoritmo seqüencial:

Para $i0 = 1$ até n de passo b faça.

Neste exemplo, $i0$ é a nova variável que comandará a variável do laço i , e b é o tamanho das tiras deste particionamento.

O laço i do algoritmo seqüencial que será transformado receberá, para iniciar, o valor da variável $i0$ do novo laço, que agora o comanda. Este laço não mais contará até o tamanho

final da matriz original. Ele é restrito ao tamanho do particionamento que foi criado por b no novo laço externo. Um exemplo deste novo laço i é mostrado em seguida.

Para $i = i_0$ até $\min(i_0+b-1, n)$ faça.

Para fazer a parada do laço, uma função $\min(x,y)$ determina o momento de parada. Quando o valor de n for menor do que o valor do primeiro termo da equação, a execução do laço termina. A Figura 48 mostra um exemplo dos 6 laços que fazem o *strip mining* no algoritmo sequencial.

```

Para  $i_0 = 1$  até  $n$  de passo  $b$  faça
  Para  $i = i_0$  até  $\min(i_0+b-1, n)$  faça
    Para  $j_0 = 1$  até  $n$  de passo  $b$  faça
      Para  $j = j_0$  até  $\min(j_0+b-1, n)$  faça
        Para  $k_0 = 1$  até  $n$  de passo  $b$  faça
          Para  $k = k_0$  até  $\min(k_0+b-1, n)$  faça

```

Figura 48 Strip mining no algoritmo sequencial

Porém, esta técnica não é suficiente para fazer um algoritmo por blocos. A ordem dos laços não favorece a formação dos blocos. Devido a isto, uma nova transformação é feita com os laços. Os laços que fazem o particionamento (i_0, j_0, k_0) são deslocados para cima, ou seja, para o lado de fora do algoritmo, e os laços que fazem a multiplicação (i, j, k) são deslocados para baixo. A Figura 49 mostra o código com a nova posição dos laços.

```

1) Para  $i_0 = 1$  até  $n$  de passo  $b$  faça
2)   Para  $j_0 = 1$  até  $n$  de passo  $b$  faça
3)     Para  $k_0 = 1$  até  $n$  de passo  $b$  faça
4)       Para  $i = i_0$  até  $\min(i_0+b-1, n)$   faça
5)         Para  $j = j_0$  até  $\min(j_0+b-1, n)$  faça
6)           Para  $k = k_0$  até  $\min(k_0+b-1, n)$  faça
7)              $c[i, j] = c[i, j] + a[i, k] * b[k, j];$ 
8)           Fim para
9)         Fim para
10)       Fim para
11)     Fim para
12)   Fim para
13) Fim para

```

Figura 49 Algoritmo sequencial de multiplicação em blocos

É este intercambio dos laços que causa as repetidas referencias aos blocos da submatriz. Neste algoritmo da Figura 49, das linhas de 1 a 3, estão os laços externos do código que divide as matrizes e indicam para os laços internos, representados nas linhas de 4 a 6, o tamanho dos blocos, ou submatriz, envolvidos na multiplicação. Quando os laços internos alcançam seus limites, dados aqui pela função $\min(x,y)$, retornam o comando para os laços externos que indicam os novos blocos a serem multiplicados. Este movimento se repete até que os laços externos cheguem aos seus limites, aqui imposto por n . Neste momento as matrizes estarão multiplicadas e a matriz resultante poderá ser apresentada.

Esta é uma técnica empregada para melhorar o desempenho final do algoritmo por particionar as matrizes e estas poderem se ajustar a capacidade da cache da arquitetura computacional em uso.

A partir deste algoritmo da Figura 49, podemos efetuar o particionamento das matrizes para a multiplicação em co-processadores em hardware. Para este particionamento precisamos levar em consideração as diversas formas existentes de multiplicar matrizes, e qual destas formas será empregada neste caso. Isto se prende ao fato de que a forma escolhida deverá nos indicar a melhor abordagem a ser seguida.

Para particionar matrizes, não é necessário o laço k e $k0$, usados somente para multiplicação. Um trecho do algoritmo de particionamento da matriz A é apresentado na Figura 50. Na linha 6, está o comando que carregará os dados deste bloco para um arquivo previamente definido. Este arquivo armazenará todos os blocos do particionamento.

```

1) Para  $i0 = 1$  até  $n$  de passo  $b$  faça
2)   Para  $r = 1$  até  $n$  de passo  $b$  faça
3)     Para  $j0 = 1$  até  $n$  de passo  $b$  faça
4)       Para  $i = i0$  até  $\min(i0+b-1, n)$  faça
5)         Para  $j = j0$  até  $\min(j0+b-1, n)$  faça
6)           arquivo_al  $\leftarrow A[i, j]$ ;
7)         Fim para
8)       Fim para
9)     Fim para
10)   Fim para
11) Fim para

```

Figura 50 Algoritmo de particionamento da matriz A

Um ponto a destacar no algoritmo da figura 50 é o comando da linha 2. Sem o comando desta linha nesta posição, a matriz A seria particionada corretamente, em sua quantidade de blocos, mas exigiria um trabalho posterior na hora de enviar os blocos para o hardware. Isto porque seria necessária a repetição do envio do mesmo bloco mais de uma vez e em um determinado momento. Com isto teríamos que usar de algum mecanismo para poder repetir os blocos e na devida seqüência da multiplicação. Com o comando da linha 2 nesta posição, os blocos da matriz A serão armazenados na quantidade certa e na seqüência correta de repetição que serão enviados. Por isto este índice é denominado de r , r de repetição. No apêndice encontram-se arquivos exemplos com o particionamento de matrizes de 8×8 em blocos de 4×4 das matrizes A , B e C .

Na construção dos blocos da matriz B , ocorre de maneira diferente que a construção dos blocos da matriz A . Enquanto que na matriz A o particionamento é no sentido horizontal, na matriz B os blocos são formados no sentido vertical. Na linha 6 do algoritmo os índices i e j são colocados invertidos. Com esta estratégia fazemos a matriz transposta como explicado no capítulo 3. Os blocos desta matriz são armazenados por colunas, ou seja, as colunas se transformam em linhas, e esta construção poderá facilitar o uso pelo hardware. O índice r do laço de repetição encontra-se no laço mais externo do bloco de particionamento. Nesta posição, após criar todos os blocos da matriz B , ele repete a criação dos blocos novamente. Um trecho do algoritmo de particionamento da matriz B é mostrado na Figura 51.

```

1) Para  $r = 1$  até  $n$  de passo  $b$  faça
2)   Para  $i0 = 1$  até  $n$  de passo  $b$  faça
3)     Para  $j0 = 1$  até  $n$  de passo  $b$  faça
4)       Para  $i = i0$  até  $\min(i0+b-1, n)$  faça
5)         Para  $j = j0$  até  $\min(j0+b-1, n)$  faça
6)           arquivo_b1  $\leftarrow B[j, i];$ 
7)         Fim para
8)       Fim para
9)     Fim para
10)   Fim para
11) Fim para

```

Figura 51 Algoritmo de particionamento da matriz B

No particionamento da matriz C , Figura 52, o sentido da partição é idêntico ao da matriz A , porém com o índice r no laço mais interno do bloco de particionamento, fazendo

com que se repitam cada bloco em seguida à sua criação. Esta é a seqüência que a matriz C receberá os blocos dos resultados das multiplicações entre as submatrizes de A e de B .

```

1) Para i0 = 1 até n de passo b faça
2)   Para j0 = 1 até n de passo b faça
3)     Para r = 1 até n de passo b faça
4)       Para i = i0 até min(i0+b-1,n) faça
5)         Para j = j0 até min(j0+b-1,n) faça
6)           arquivo_c1 ← C[i,j];
7)         Fim para
8)       Fim para
9)     Fim para
10)   Fim para
11) Fim para

```

Figura 52 Algoritmo de particionamento da matriz C

Neste esquema, usando o algoritmo seqüencial por blocos mostrado na Figura 49, as linhas de 1 a 3 não são consideradas de computação intensiva. Todas elas percorrem n , mas a passos de b , na execução do algoritmo. A função destas três linhas é somente a de dividir o código que será levado para os outros três laços mais internos tratar.

As linhas de 4 a 7 fazem parte da computação intensiva, pois elas executam, $2\frac{n^3}{b}$ operações de multiplicações e somas, e manuseiam $3\frac{n^2}{b}$ dados a cada vez que são chamadas a executar. Neste algoritmo estas linhas são as candidatas a ser levadas para execução no coprocessador em hardware.

6.4 Fluxograma do Escalonamento dos Blocos para Multiplicação em Hardware

O fluxograma do particionamento apresentado na Figura 53 mostra o esquema desta integração de hardware e software para a multiplicação de grandes matrizes em hardware.

Para realizar este fluxo apresentado na figura 53, um aplicativo na linguagem C conteria as funções de leitura das matrizes, os três algoritmo de particionamento destas matrizes, que após particionadas teriam suas submatrizes armazenadas em arquivos. Outra função neste aplicativo seria encarregada de ler seqüencialmente os blocos das duas matrizes que serão enviadas ao coprocessador em hardware. Após receber a confirmação de término da

multiplicação do hardware, o componente em software (sw) busca o bloco da matriz resultado e adiciona ao respectivo bloco da matriz *C*, que também foi particionada para receber os blocos na sequência correta. Um contador de blocos da matriz *C* seria o responsável por averiguar se todos os blocos esperados da multiplicação foram recebidos adequadamente. Se o contador de blocos recebidos não for igual ao de quantidade de multiplicação a ser efetuadas, o algoritmo encarregado de ler os blocos envia outra sequência de blocos das matrizes *A* e *B* até a finalização da multiplicação.

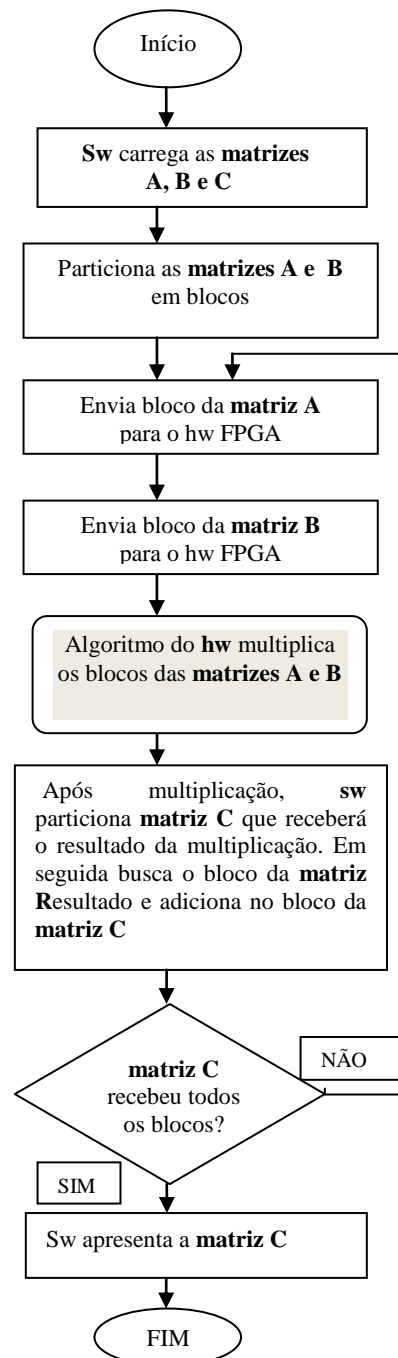


Figura 53 Fluxograma do Particionamento e Multiplicação em Hardware

Esta sucessão de eventos ocorre até o contador de blocos da matriz C se igualar ao contador de quantidade de multiplicação. Quando isto ocorrer, a multiplicação estará completa e o software apresentará a nova matriz C .

6.5 Arquitetura de Multiplicação de Matrizes para a Plataforma RASC

No trabalho realizado em [18], foi construída uma arquitetura de multiplicação para grandes e densas matrizes de ponto flutuante de precisão dupla (64 bits). O objetivo é construir um multiplicador de matrizes de 100x100. Usando a plataforma do RASC, apresentada no capítulo 5, foi implementada a construção de 2 bancos lógicos de memória, construídos através de 2 bancos físico de 8MB cada. Nestes bancos lógico de 16MB foram feitos segmentos de tamanho de 256KB. Esta capacidade do segmento é suficiente para armazenar matriz de dimensão 100, que teriam cerca de 80KB cada. Na Figura 54 é mostrada esta construção.

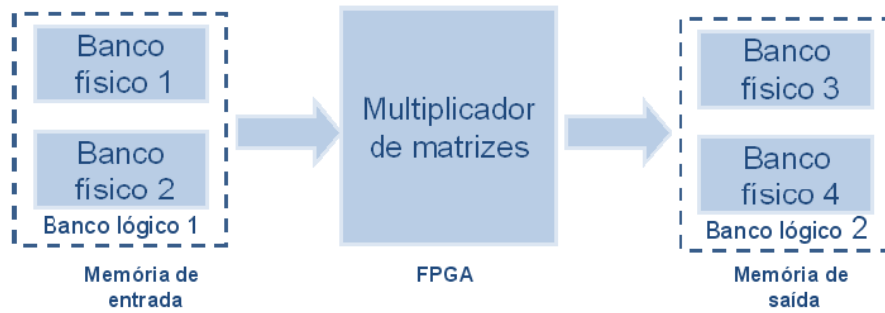


Figura 54 Construção dos bancos de memória [18]

O engenho de multiplicação de matrizes faz esta operação multiplicando vetores das colunas da matriz A por vetores das linhas da matriz B . Quando termina a multiplicação destes 2 (dois) vetores, os dados parciais deverão ser armazenados para somarem com dois próximos vetores. No caso de uma matriz de dimensão 100, esta operação só termina após 100 multiplicações entre vetores A . A Figura 55 mostra como foi executado esta operação.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} \\ a_{21} \\ \dots \\ a_{n1} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \end{bmatrix} + \begin{bmatrix} a_{12} \\ a_{22} \\ \dots \\ a_{n2} \end{bmatrix} \cdot \begin{bmatrix} b_{21} & b_{22} & \dots & b_{2n} \end{bmatrix} + \dots + \begin{bmatrix} a_{1n} \\ a_{2n} \\ \dots \\ a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{n1} & b_{n2} & \dots & b_{nn} \end{bmatrix}$$

$$A_{n \times n} * B_{n \times n} = A_{n \times 1}^1 * B_{n \times 1}^1 + A_{n \times 1}^2 * B_{n \times 1}^2 + \dots + A_{n \times 1}^n * B_{n \times 1}^n$$

Figura 55 Multiplicação dos vetores das matrizes [18]

Uma coluna da matriz A é armazenada em FIFO nas entradas dos multiplicadores denominados de MAC, construído com blocos de DSP. Linhas da matriz B são armazenadas em FIFO específica na entrada de cada multiplicador. Na saída outra FIFO armazena os resultados, aguardando o final da multiplicação para descarregar os dados no segmento de memória de saída. A Figura 56 mostra este esquema.

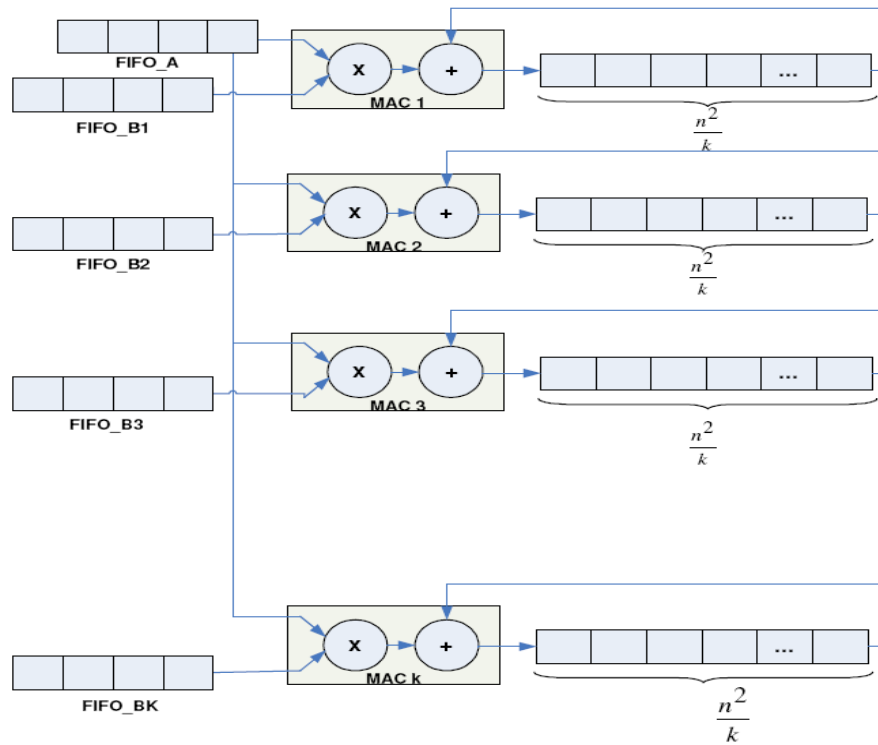


Figura 56 Arquitetura do multiplicador [18]

Para alimentar esta arquitetura, através dos dados de um aplicativo, a única exigência é de que os dados da matriz A sejam armazenados nos segmentos de memória de entrada em colunas, ao invés de linhas como seria tradicionalmente. Neste caso então, ao invés de fazermos a matriz B transposta, é a matriz A que deverá ser feita transposta para facilitar o engenho de busca do algoritmo em questão.

6.6 Conclusão

A metodologia PCAM, para desenvolvimento de algoritmos paralelos, é considerada uma metodologia bem estruturada, pois inicia com a especificação do problema, desenvolve uma estratégia de particionamento, determina as necessidades de comunicação e se for o caso, aglomera as tarefas para finalmente realizar o mapeamento destas para os processadores.

No algoritmo do hardware não temos comunicação entre os *PEs*, pois cada um recebe seus dados e trabalham de forma independente. A comunicação é realizada apenas entre os

PEs e o *host*. A estratégia de particionamento é dependente da quantidade de elementos de armazenamento existente na estrutura do *FPGA*.

É sabido que todas estas metodologias, técnicas, abordagens de uso, reuso e de proximidades dos dados surgiram das necessidades de desempenho observadas em sistemas gerenciados por processadores de software. Mas todos estes conceitos são muito úteis também para uso nestes novos coprocessadores em hardware. O hardware não tem uma “cache” em seu algoritmo, mas o conceito de cache, que nos leva a localidade dos dados é bem observável para o hardware. O armazenamento em blocos de *RAMs*, as *BRAMs*, em *FPGA*, seria considerado como cache, pois estas estão localizadas muito próximo aonde os dados serão manuseados. Elas são as memórias locais internas do algoritmo. A *SRAM* é a memória local externa.

Capítulo 7 – Estudo de Caso

Este capítulo apresenta um estudo de caso baseado em estimativas obtidas considerando as características de três plataformas comerciais distintas, tais como: largura de banda da comunicação com o host, largura de banda com a memória e tempo de execução da multiplicação de matrizes em hardware. O tempo de execução em hardware foi estimado pelo modelo proposto em [18]. Este modelo fornece o número de ciclos de relógio necessários para realizar a multiplicação de matrizes de tamanho n em FPGA, considerando características como: frequência de operação do FPGA, largura de banda com a memória, latência no acesso aos dados e número de elementos de processamento.

O estudo consiste em estimar o tempo total de execução da multiplicação de duas matrizes grandes e densas considerando que as mesmas serão particionadas pelo host em blocos de matrizes menores (100 x 100) que serão enviadas para o hardware reconfigurável. Os tempos obtidos nas plataformas reconfiguráveis são então comparados com o tempo obtido para realização da multiplicação completamente em software. As latências no acesso aos dados no barramento e na memória assim como o *overhead* inserido pelo sistema operacional não foram considerados.

7.1 Descrições das plataformas utilizadas nas estimativas

As plataformas comerciais reconfiguráveis de alto desempenho Cray XD1 e SGI RASC descritas nos capítulos 4 e 5, e a plataforma comercial genérica Xilinx ML555 apresentam as larguras de banda de acesso a memória e de comunicação com o host como descrito na Tabela 4.

Tabela 4 - Largura de banda de acesso a memória e de comunicação com o host

Plataforma	Largura de banda com o host (GB/s)	Largura de banda com a memória (GB/s)
Cray XD1	3,2	6,4
SGI RASC	3,2	3,2
Xilinx ML555	1,05	3,2

As duas primeiras apresentam grandes vantagens em relação a terceira. Maior largura de banda na comunicação com o *host* e os múltiplos canais de acesso a memória são exemplos. Os múltiplos canais de acesso permitem que a carga e descarga de dados da memória com o *host* ocorram independente do acesso a memória realizado pelo algoritmo em processamento. Para atenuar esta desvantagem é possível utilizar na plataforma ML555 a técnica de *overlap*. A técnica de *overlap* consiste em armazenar os dados que serão utilizados no processamento, internamente no FPGA utilizando seus meios de armazenamento interno, como os blocos de RAMs (BRAMs), liberando o único canal de acesso a memória para que o *host* possa voltar a acessá-lo, durante o processamento do algoritmo.

Esta técnica só pode ser empregada com sucesso quando o tempo de processamento for maior que o tempo necessário para a transferência de dados. O *host* pode utilizar o tempo de processamento, se este for dominante, para realizar todas as cargas e descargas necessárias. As estimativas apresentam o desempenho da plataforma Xilinx ML555 com e sem o emprego da técnica de *overlap*.

A plataforma de desenvolvimento ML555 é equipada com um FPGA Virtex 5 (XC5VLX50T) da Xilinx e conecta-se ao *host* através de um barramento PCI Express de 8 lanes. Esta plataforma vem equipada com memória DDR2 SDRAM de 256 MB, com palavras de 64 bits. A Figura 57 apresenta um diagrama em bloco desta plataforma.

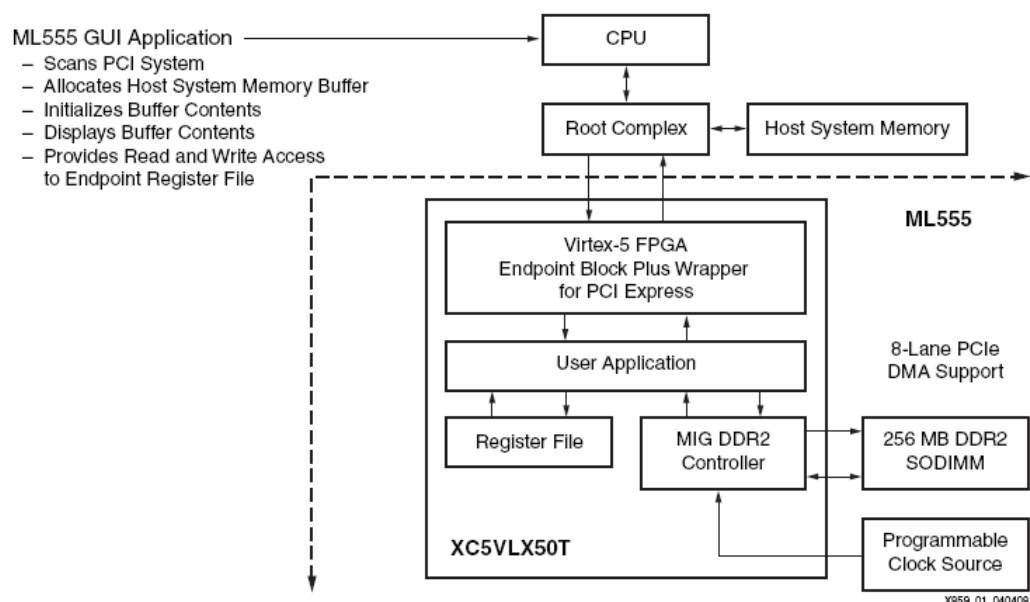


Figura 57 Diagrama em bloco da plataforma Xilinx ML555

A Xilinx fornece as ferramentas necessárias para criar e executar um projeto nesta plataforma. Ferramentas de terceiros também são disponibilizadas como é o caso do *device*

driver para a interface PCIe (Peripheral Component Interconnect Express) gerado pelo Jungo, WinDriver Ltd. para acessar o dispositivo no barramento.

Esta plataforma também possui um DMA para ler e escrever as transações iniciadas pelo *device driver* do PCIe. O controlador de memória utilizado para que o algoritmo implementado no FPGA acesse a memória disponível na placa é gerado automaticamente através do aplicativo MIG (Memory Interface Generator) que também é fornecido pela Xilinx.

7.2 Estimativa do custo da movimentação de dados

Considerando as larguras de banda apresentadas na Tabela 4, podemos realizar uma estimativa do custo, em tempo, para movimentar os dados do host para as plataformas reconfiguráveis. A Tabela 5 apresenta os tempos estimados nas plataformas, Cray XD1, RASC RC100 e Xilinx ML555. É importante destacar que as plataformas comerciais que apresentam foco em alto desempenho possuem barramentos especiais com grande largura de banda e baixa latência, como apresentado nos capítulos 4 e 5. A plataforma Xilinx ML555 possui largura de banda menor conectando-se ao host através do barramento PCIe de 8 *lanes*.

Tabela 5 - Estimativas de custo em tempo da movimentação de dados nas plataformas comerciais

Plataforma	Largura de Banda	Custo / Movimentação escrita de A e B e leitura de R em ms
Cray XD1	3,2 GB/s	0,0698
SGI RASC	3,2 GB/s	0,0698
Xilinx ML555	1,05 GB/s	0,1935

A carga de trabalho foi a multiplicação de matrizes de ponto flutuante de precisão dupla (64 bits), de dimensão 100 x 100. Portanto, para esta carga é necessário transferir para o FPGA as duas matrizes que serão multiplicadas, A e B, e por fim, descarregar a matriz resultante R. Então teremos no total $3 \cdot n^2$ dados de 64 bits transferidos entre o *host* e as plataformas reconfiguráveis.

Os custos em tempo para movimentação destes dados foi calculado pela equação $\frac{N^2 \times bit}{BW}$, onde N é a dimensão das matrizes, bit é o tamanho da palavra de cada elemento das matrizes e BW é a largura de banda dada em bit/s.

7.3 Estimativa do Custo de Processamento

A estimativa do custo de processamento em hardware da multiplicação de duas matrizes de dimensão 100 x 100 foi realizada baseando-se no modelo proposto em [18]. Segundo este modelo, é possível estimar em número de ciclos o tempo de processamento da multiplicação em hardware, se for possível saber antecipadamente a frequência de operação do FPGA, a largura de banda com a memória, a latência no acesso aos dados e o número de elementos de processamento. A equação que permite a obtenção dessa estimativa é $T_{exec} = T_{init} + 100 * T_{mult} + T_{write}$, onde: T_{init} é o tempo de acesso aos primeiros dados contidos na SRAM que permitem o início do processamento, T_{mult} é o tempo de uma multiplicação vetorial e T_{write} é o tempo de escrita dos dados na memória SRAM.

Baseado neste modelo e considerando que o multiplicador em hardware utiliza 10 MACs (multiplicadores acumuladores) e que os FPGAs estão trabalhando numa frequência única de 200 MHz, foram obtidos os dados da Tabela 6 que apresenta os tempos de processamento em hardware nas plataformas citadas anteriormente. Também é apresentado na Tabela 6 o tempo total de execução adicionando-se os custos com a movimentação de dados apresentados na Tabela 5 para cada plataforma.

Tabela 6 - Estimativas do custo de processamento e do tempo total considerando a movimentação de dados nas plataformas comerciais

Plataforma	Processamento (ms)	Custo de Movimentação escrita de A e B e leitura de R em ms	Processamento (ms) c / movimentação de dados
Cray XD1	0,5702	0,0698	0,6400
SGI RASC	0,5704	0,0698	0,6402
Xilinx ML555	0,5350	0,1935	0,7285

Os resultados mostram que os tempos da movimentação de dados são menores que o tempo de processamento em todas as situações. Isto significa que nesta proporção de tempos, processamento e movimentação de dados, um algoritmo eficiente conseguiria fazer overlap da movimentação de dados sobre o processamento. O tempo das transferências só seria computado para o primeiro processamento e para o último, sendo 1ª carga e última descarga. Todas as outras cargas e descargas seriam realizadas dentro do tempo de processamento.

7.4 Análise do Desempenho das Plataformas ML 555, RASC e Cray

Baseado no modelo de estimativas, diversas comparações foram feitas entre as plataformas apresentadas. É sabido que a diferença de largura de banda em favor das plataformas focadas em alto desempenho poderia levar a plataforma ML555 a um desempenho bastante inferior. Porém foi constatado que com uma boa estratégia de acesso a memória, procurando explorar as oportunidades de uso do overlap, a plataforma Xilinx ML555 teve um desempenho muito próximo das plataformas comerciais reconfiguráveis focadas em alto desempenho. A seguir são apresentados alguns gráficos que demonstram os resultados comparativos obtidos.

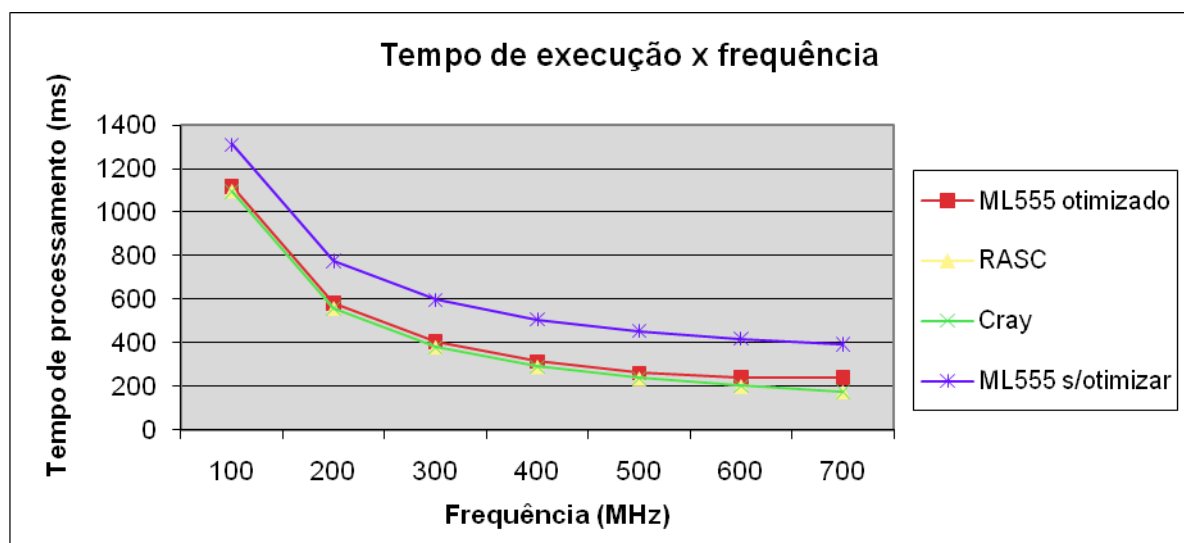


Figura 58 Desempenho do tempo de execução x frequência

O gráfico da Figura 58 apresenta a relação tempo de processamento versus frequência de operação para multiplicação de matrizes 1000 x 1000 particionadas em matrizes menores de ordem 100 utilizando 10 MACs no recurso reconfigurável. Pode-se observar que o desempenho da plataforma ML555 quando utilizando o algoritmo otimizado é bem próximo

das plataformas de alto desempenho. Para uma frequência de 200 MHz, a diferença entre a plataforma ML555 as plataformas comerciais focadas em alto desempenho é de cerca de 4%. Essa diferença de desempenho aumenta quando a frequência aumenta porque o tempo de processamento diminui e a largura de banda disponível passa a ser significativa no tempo total de processamento.

Observa-se também que o uso da otimização ocasiona uma melhoria de desempenho que pode chegar a 42% quando comparado ao algoritmo sem otimização. Esta taxa é extraída do tempo total de execução com *overlap* (otimizado) sobre o tempo total de execução sem *overlap*. O tempo de processamento, como esperado, diminui com o aumento da frequência, porém, no caso da implementação com *overlap* essa queda se estabiliza numa determinada frequência (aproximadamente 600 MHz para 10 MACs). Isto porque, o tempo de carga e descarga dos dados passa a se sobrepor sobre o tempo de processamento. Neste caso, para melhorar o desempenho seria necessário aumentar a largura de banda.

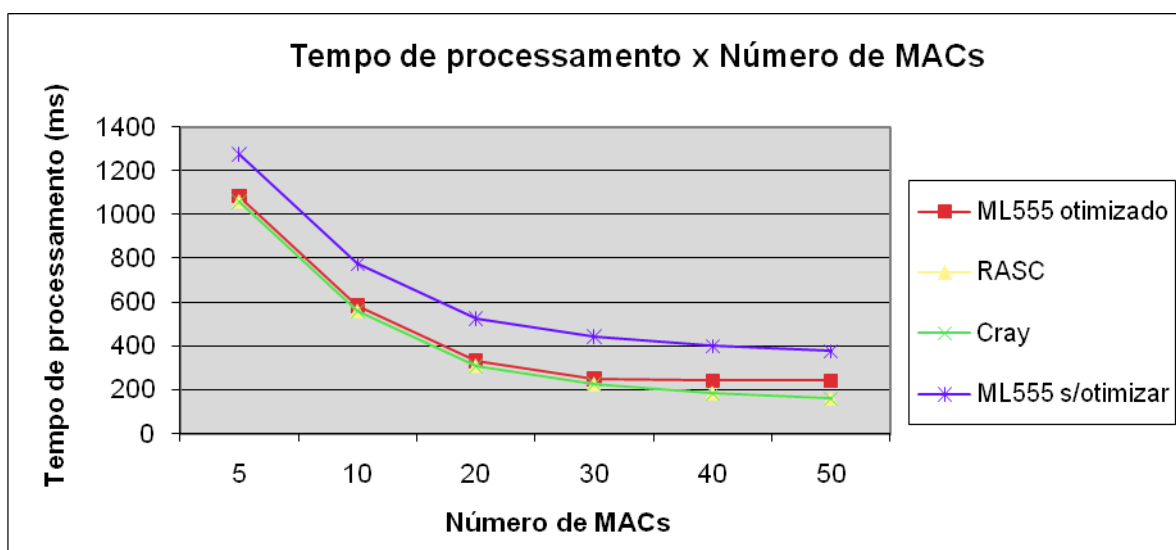


Figura 59 Desempenho do tempo de processamento x quantidade de MACs

O gráfico da Figura 59 mostra os tempos de processamento das plataformas em questão para uma mesma frequência de operação (200MHz), considerando uma variação do número de MACs (multiplicadores acumuladores). A partir do gráfico podemos ressaltar, mais uma vez, a melhoria de desempenho da plataforma ML555 ocasionada pela utilização do algoritmo otimizado, esta melhoria chega a cerca de 40%. Entretanto, observa-se também que a partir de certo número de MACs (entre 30 e 40), o tempo de processamento do algoritmo otimizado se estabiliza e não melhora como era de se esperar. Quando o número de elementos

de processamento aumenta, o tempo efetivo de processamento diminui e mais uma vez, o tempo de carga e descarga passa a se sobrepor ao tempo de processamento, limitando o desempenho da plataforma ML555 e aumentando a diferença entre os tempos de processamento dela em relação as outras plataformas comerciais já que elas possuem uma maior largura de banda. Esta diferença chega a 34%.

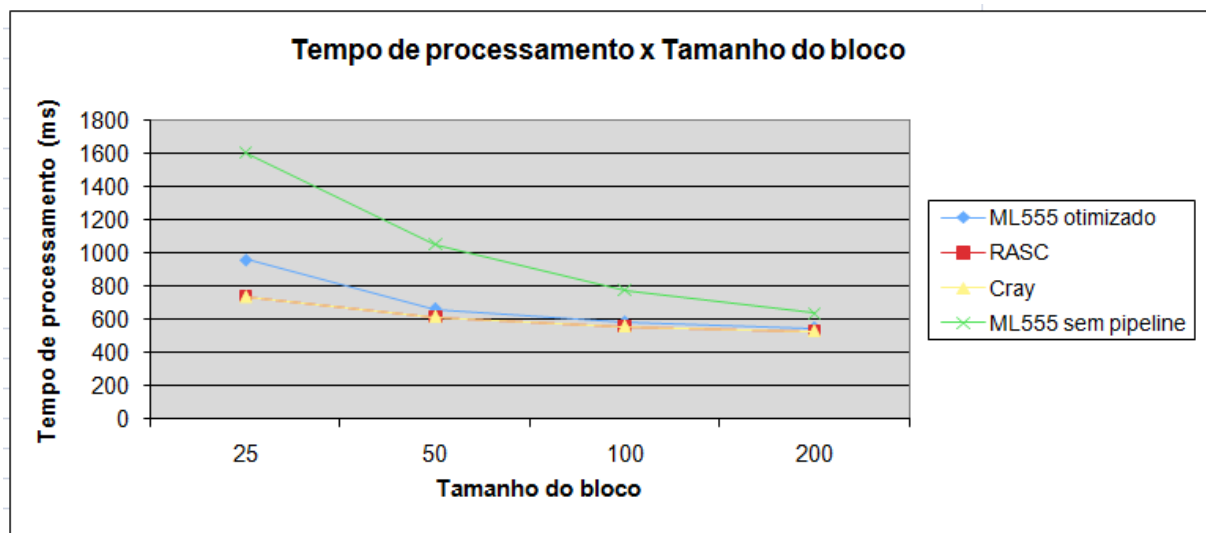


Figura 60 Desempenho do tempo de processamento x tamanho do bloco

O gráfico da Figura 60 mostra como o tempo de processamento total da multiplicação de duas matrizes pode variar dependendo do tamanho do bloco no qual essas matrizes foram particionadas. Consideramos duas matrizes de ordem 1000 particionadas em submatrizes de tamanho 25, 50, 100 e 200.

Observamos que para um tamanho de bloco muito pequeno, há uma diferença significativa de desempenho entre a plataforma ML555 e as plataformas comerciais de alto desempenho, isto acontece porque, como o bloco é pequeno, o tempo de processá-lo é curto e o tempo de carga e descarga passa a dominar o tempo total de processamento. No início esta diferença fica em torno de 24%.

Com o aumento do tamanho do bloco, a diferença diminui e o desempenho da plataforma ML555 começa a se aproximar das outras plataformas devido ao fato do tempo de processamento passar a ser dominante, sobrepondo-se ao tempo de carga e descarga dos dados.

Para um tamanho de bloco igual a 200, a diferença da plataforma ML555 para as demais plataformas se aproxima do mínimo (cerca de 3%).

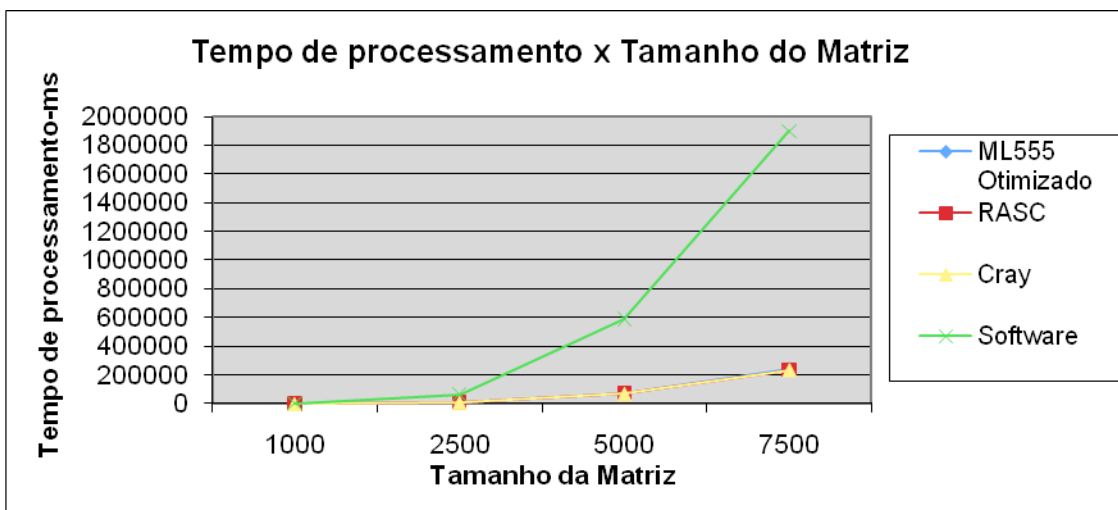


Figura 61 Desempenho do tempo de processamento x tamanho da matriz

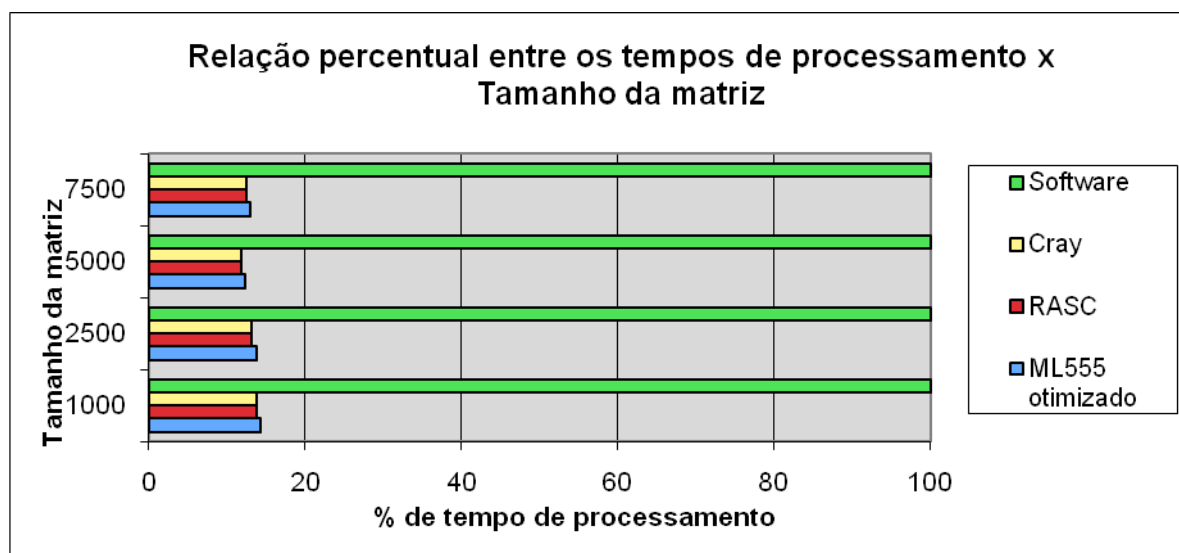


Figura 62 Relação entre os tempos de processamento x tamanho da matriz

Os gráficos 61 e 62 apresentam a comparação entre o desempenho das plataformas reconfiguráveis em relação às implementações em software, um processador Dual core AMD Athlon 64 X2 4400+ com 2.300 MHz, cache L1 de 2x 128 KB, L2 de 2x 512 KB e 2 GB de memória, para multiplicação de matrizes de ordem 1000, 2500, 5000 e 7500 particionadas em submatrizes de tamanho 100.

Podemos observar, na Figura 61, que a estimativa de execução em hardware apresenta um desempenho bastante superior em comparação à implementação em software, sobretudo quando a ordem das matrizes aumenta.

O gráfico da Figura 62 é um gráfico percentual e considera que o tempo de processamento em software vale 100%. Desta forma, podemos relacionar percentualmente os demais tempos para as outras plataformas de hardware. O que podemos observar é que os tempos de processamento em hardware não alcançam 20% do tempo de processamento em software. Por exemplo, para matrizes de tamanho 1000 o tempo de implementação em hardware é cerca de 15% do tempo de implementação em software.

7.5 Conclusão

Neste capítulo foi apresentada uma análise de desempenho baseada em estimativas de duas plataformas comerciais reconfiguráveis de alto desempenho e uma plataforma comercial reconfigurável de propósito geral, na execução da operação de multiplicação de matrizes em diversas condições de operação e particionamento.

Fazendo uma análise inicial das características da plataforma ML555 pode-se observar que esta apresenta uma largura de banda entre o host e a memória SRAM inferior a SGI RASC RC100 e a Cray XD1. Além disso, a plataforma ML555, diferente das plataformas comerciais de alto desempenho, não possui memória com múltiplos canais de acesso. Analisando de uma forma superficial estas diferenças poderiam ocasionar uma perda de desempenho considerável da plataforma ML555.

Entretanto, a partir do estudo e definição da melhor forma de particionar e sincronizar os processamentos e transferência de dados, tirando proveito da técnica de *overlap*, foi possível obter um resultado de desempenho bastante aceitável em relação as plataformas de alto desempenho e de maior custo.

Obviamente, deve-se ressaltar que estas estimativas representam um valor esperado de desempenho, pois, overhead como latências do sistema operacional e do barramento não foram consideradas.

[PAGINA INTENCIONALMENTE DEIXADA EM BRANCO]

Capítulo 8 – Conclusão

Neste capítulo é apresentada a conclusão sobre os resultados obtidos e os possíveis trabalhos futuros.

8.1 Resultados Alcançados

Este trabalho teve como principal resultado a definição de um algoritmo de particionamento capaz de fragmentar o problema de multiplicação de matrizes grandes e densas em submatrizes menores que possam ser calculadas em um hardware reconfigurável especializado na execução desta operação. As análises apresentadas no decorrer do trabalho discutiram os problemas envolvidos na criação destes fragmentos em especial o overhead de comunicação e distribuição de dados.

Os algoritmos de particionamento são essenciais na execução de problemas de grande porte principalmente em aplicações científicas. Este particionamento permite que a carga de processamento seja distribuída em vários elementos de processamento diminuindo o tempo total de processamento.

Estes algoritmos também favorecem a escalabilidade, caso o problema abordado aumente de escala um particionamento bem definido pode distribuir esta carga adicional para novos elementos de processamento mantendo um desempenho total aceitável.

As análises realizadas nas três plataformas comerciais Cray XD1, SGI RASC e Xilinx ML555 mostraram que quando o tempo total de execução da operação é dominado pelo tempo de processamento e não pela comunicação, a técnica de particionamento apresenta uma grande melhoria de desempenho.

Também foi possível observar que apesar de múltiplos canais de memória propiciar um melhor desempenho no tempo total de execução da operação, o emprego de técnicas que tiram proveito do *overlap* podem trazer ganhos de desempenho significativos.

Outra característica importante deste trabalho é que as técnicas aqui propostas não dependem de plataforma. Portanto, podem ser empregadas nos mais diferentes cenários.

8.2 Trabalhos Futuros

O confronto dos resultados obtidos através de estimativas com dados de execução real nas plataformas discutidas será de grande importância para o engrandecimento do trabalho.

Outro ponto a ser trabalhado é a execução simultânea em mais de um hardware reconfigurável aumentando ainda mais o desempenho em relação às soluções aqui propostas.

O particionamento de matrizes de vários formatos, que não quadrada, também ensejam um estudo de viabilidade para implementação em plataformas de hardware

Referências Bibliográficas

[1] Windows HPC Server 2008 - Mercado de Capitais.

<http://www.microsoft.com/brasil/servidores/windowsserver2003/ccs/finserv/capital.mspx>

[2] Pessoa et al – Uma ferramenta de processamento de imagens médicas dentro do Sistema HeMoLab. VII Workshop de Informática Médica - WIM 2007

<http://www.ime.uerj.br/professores/cecas/AnaisWIM2007/sessao4/2-28305.pdf>

[3] Microsoft – High-Performance Computing Goes Mainstream in E&P

<http://www.microsoft.com/industry/manufacturing/oilandgas/businessvalue/HPCmainstream.mspx>

[4] CASPUR (Consorzio interuniversitario per le Applicazioni di Supercalcolo per Università Ricerca). Italian Supercomputing Center Reaches New User Communities with Windows-Based HPC.

<https://www.microsoft.com/casestudies/casestudy.aspx?casestudyid=4000002595>

[5] IBM - <http://www.ibm.com/br/systems/intellistation/pro/zpro/index.phtml>

[6] HP Alarga Portfolio Líder de Servidores com Novos Processadores Dual-core de Alto Desempenho.

http://h41131.www4.hp.com/pt/pt/press/HP_Alarga_Portfolio_Lder_de_Servidores_com_Novos_Processadores_Dual-core_de_Alto_Desempenho.html

[7] SGI – Altix XE Servers and Clusters. <http://www.sgi.com/products/servers/altix/x/>

[8] TOP 500 – Supercomputer Sites. <http://www.top500.org/>

[9] Portal Fator Brasil - Universidade Federal de Santa Catarina inicia tecnologia SGI para produção elevada de petróleo e gás.

http://www.revistafator.com.br/ver_noticia.php?not=49944

[10] HP – Rápido e Econômico.

http://h41131.www4.hp.com/pt/pt/stories/HP_Feature_Story_Computao_de_Alto_Desempenho_Janeiro_de_2006.html

- [11] Agência FAPESP – Poderoso Netuno. Servidores DELL.
<http://www.agencia.fapesp.br/materia/8811/noticias/poderoso-netuno.htm>
- [12] Inovação Tecnológica - UFRJ ganha supercomputador para estudar energia e infraestrutura. <http://www.inovacaotecnologica.com.br/noticias/noticia.php?artigo=ufrj-ganha-supercomputador-para-estudar-energia-e-infra-estrutura>
- [13] Fatahalian, K. et al - Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. Conference on Graphics hardware. Agosto de 2004. HWWS 04.
<http://portal.acm.org/results.cfm?coll=ACM&dl=ACM&CFID=22211227&CFTOKEN=75535599>
- [14] Daniel Chavarría-Miranda, Andrés Márquez - Assessing the Potential of *Hybrid HPC* Systems for Scientific Applications: A Case Study. Pacific Northwest National Laboratory.
<http://delivery.acm.org/10.1145/1250000/1242558/p173-chavarria.pdf?key1=1242558&key2=0601384321&coll=ACM&dl=ACM&CFID=22211227&CFTOKEN=75535599>
- [15] XILINX <http://www.xilinx.com/>
- [16] ALTERA <http://www.altera.com/>
- [17] Woods, Nathan - The Architecture and Implementation Perspective - Integrating FPGAs in High-Performance Computing. <http://delivery.acm.org/10.1145/1220000/1216941/p132-woods.pdf?key1=1216941&key2=0750682421&coll=ACM&dl=ACM&CFID=35456699&CFTOKEN=68287410>
- [18] Souza, Viviane Lucy Santos de – Implementação de uma Arquitetura para Multiplicação de Matrizes Densas em Sistemas Reconfiguráveis de Alto Desempenho. Dissertação de Mestrado. Agosto de 2008. Acessado em janeiro de 2009. Disponível em:
www.bdt.ufpe.br/tedeSimplificado//tde_busca/arquivo.php?codArquivo=5201
- [19] BLAS - <http://www.netlib.org/blas/>
- [20] Dou, Y., Vassiliadis, S., Kuzmanov, G. K., and Gaydadjiev, G. N. 2005. 64-bit

floating-point FPGA matrix multiplication. In Proceedings of the 2005 ACM/SIGDA 13th international Symposium on Field-Programmable Gate Arrays (Monterey, California, USA, February 20 - 22, 2005). FPGA '05. ACM, New York, NY, 86-95

[21] Volodymyr Kindratenko Code Partitioning for Reconfigurable High-Performance Computing: A Case Study. In Proc. The International Conference of Reconfigurable Systems and Algorithmus (ERSA '06), June 26 – 29, 2006, Las Vegas, Nevada

[22] Zhuo, L. Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on Reconfigurable Computing Systems. *IEEE Trans. Parallel Distrib. Syst.* 18, 4 (Apr. 2007), 433-448.

[23] Campbell, S. J. and Khatri, S. P. 2006. Resource and delay efficient matrix multiplication using newer FPGA devices. In Proceedings of the 16th ACM Great Lakes Symposium on VLSI (Philadelphia, PA, USA, April 30 - May 01, 2006). GLSVLSI '06. ACM, New York, NY, 308-311.

[24] Laurenz Christian Buri - Studies of classical HPC problems on fine-grained and massively parallel computing environment based on reconfigurable hardware. Master of Science Thesis. Stockholm, Sweden 2006

[25] Michele Alves de Freitas Batista, Lamartine Nogueira Frutuoso Guimarães. Algoritmos Genéticos em Ambientes Paralelos. Instituto Nacional de Pesquisas Espaciais Anais do V WORCAP, INPE, São José dos Campos, 26 e 27 de outubro de 2005

[26] Schepke, Claudio Distribuição de Dados para Implementações Paralelas do Método de Lattice Boltzmann - Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação - Porto Alegre, março de 2007

[27] M.D. Galanis, A. Milidonis, G. Theodoridis, D. Soudris, C.E. Goutis - A Partitioning Methodology for Accelerating Applications in Hybrid Reconfigurable Platforms. Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)

[28] James C. Phillips, John E. Stone, and Klaus Schulten - Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters. SC2008 November 2008, Austin, Texas, USA

[29] Catherine H. Crawford, Paul Henning, Michael Kistler, Cornell Wrigth - Accelerating Computing with the Cell Broadband Engine Processor. Conference On Computing Frontiers

Proceedings of the 5th conference on Computing frontiers *CF'08*, May 5–7, 2008, Ischia, Italy.

[30] Keith D. Underwood, K. Scott Hemmert - Closing the gap: CPU and FPGA Trends in sustainable floating-point BLAS performance. Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04) - Napa, California 20 - 23 April, 2004

[31] Daniel Chavarría Miranda, Andrés Márquez Assessing the Potential of Hybrid HPC Systems for Scientific Applications: A Case Study. *CF'07*, May 7–9, 2007, Ischia, Italy. Copyright 2007 Association for Computing Machinery. ACM

[32] Laurenz Christian Buri, Studies of Classical HPC Problems on fine-grained and massively parallel computing environment based on reconfigurable hardware, Msc. Thesis, Department of Microelectronics and Information Technology IMIT KTH, 2006.

[33] Smith, M. C. and Peterson, G. D. 2005. Parallel application performance on shared high performance reconfigurable computing resources. *Perform. Eval.* 60, 1-4 (May. 2005), 107-125.

[34] SRC Computers, Inc., <http://www.srccomp.com/>

[35] SNAP - <http://www.srccomp.com/techpubs/snap.asp>

[36] Sadaf R. Alam, Pratul K. Agarwal, Melissa C. Smith, Jeffrey S. Vetter, David Caliga, "Using FPGA Devices to Accelerate Biomolecular Simulations," *Computer*, vol. 40, no. 3, pp. 66-73, Mar., 2007

[37] CRAY XD1– <http://www.cray.com>

[38] AMD Opteron - <http://www.amd.com/us/products/server/opteron/>

[39] RocketI/O -

Xilinx.http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/rocketio.htm

[40] Arquitetura DCP - <http://www.inf.ufrgs.br/procpa/disc/cmp134/trabs/T1/061/t1-mvneves-CrayXD1-relatorio.pdf>

[41] Link HyperTransport - <http://www.hypertransport.org/>

- [42] Ling Zhuo , Viktor K. Prasanna, Scalable Hybrid Designs for Linear Algebra on Reconfigurable Computing Systems, Proceedings of the 12th International Conference on Parallel and Distributed Systems, p.87-95, July 12-15, 2006.
- [43] Zhuo, L.; Prasanna, V.K., "Hardware/Software Co-Design for Matrix Computations on Reconfigurable Computing Systems," *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, vol., no., pp.1-10, 26-30 March 2007
- [44] Ling Zhuo and Viktor K. Prasanna High Performance Linear Algebra Operations on Reconfigurable Systems. Proceedings of the 2005 ACM/IEEE SC'05 Conference (SC'05) November 12-18, 2005, Seattle, Washington, USA
- [45] Ling Zhuo and Viktor K. Prasanna Optimizing Matrix Multiplication on Heterogeneous Reconfigurable Systems. In Proceedings of Parallel Computing 2007, September 2007, Germany
- [46] RASC - Reconfigurable Application-Specific Computing
<http://www.sgi.com/products/rasc/>
- [47] Interconexão NUMALINK <http://www.sgi.com/products/servers/altix/numalink.html>
- [48] Altix 350 - www.sgi.com/products/remarketed/altix350/
- [49] SSP Stub Users Guide. http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=linux&db=bks&fname=/SGI_EndUser/RASC_UG/apb.html
- [50] J. Koo, D. Fernandez, A. Haddad, and W. J. Gross, "Evaluation of a High-Level Language Methodology for High-Performance Reconfigurable Computers," Proceedings of the IEEE 18'th International Conference on Application-Specific Systems, Architectures and Processors, Montreal, Canada, pp. 30-35, July 8-11, 2007
- [51] Mitronics AB <http://www.mitrion.com/>
- [52] RASCL - RASC Abstraction Layer Calls - http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/linux/bks/SGI_EndUser/books/RASC_UG/sgi_html/ch04.html
- [53] RASC Core Services Silicon Graphics Inc., Mountain View, CA, Reconfigurable Application-Specific Computing User's Guide, 2006.

[54] Synopsys VCS/VCSi Tutorial.

ftp://ftp.xilinx.com/pub/documentation/M3.1i_tutorials/ug108.pdf

[55] Virsim - User Guide. <http://www.stanford.edu/class/ee271/docs/virsim.pdf>

[56] Smith, M. C., Vetter, J. S., and Liang, X. 2005. Accelerating Scientific Applications with the SRC-6 Reconfigurable Computer: Methodologies and Analysis. In Proceedings of the 19th IEEE international Parallel and Distributed Processing Symposium (Ipdps'05) - Workshop 3 - Volume 04 (April 04 - 08, 2005). IPDPS. IEEE Computer Society, Washington, DC, 157.2.

[57] FOSTER, I. T. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Publishing Company. 1994. Livro on line.

[58] Justin L. Tripp, Anders A. Hanson, Maya Gokhale - Partitioning Hardware and Software for Reconfigurable Supercomputing Applications: A Case Study. Proceedings of the 2005 ACM/IEEE SC05 Conference (SC'05) - November 12-18, 2005, Seattle, Washington, USA

[59] Jayme Luiz Szwarcfiter, Lilian Markenzon – Estruturas de Dados e seus Algoritmos – Editora LTC S.A. 2ª Edição Revista – Rio de Janeiro - RJ (livro)

[60] Robert Schreiber, Jack J. Dongarra - Automatic Blocking of Nested Loops - May1990

APÊNDICE

APÊNDICE I – Arquivo Exemplo com os Dados do Particionamento da Matriz A

APÊNDICE II – Arquivo Exemplo com os Dados do Particionamento da Matriz B

APÊNDICE III – Arquivo Exemplo com os Dados do Particionamento da Matriz C

APÊNDICE I – Arquivo Exemplo com os Dados do Particionamento da Matriz A

Matriz 8x8

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Blocos 4x4 da Matriz A Particionada – Os blocos se repetem nas posições de leitura do engenho de busca para a multiplicação com os blocos da matriz B.

1	2	3	4
9	10	11	12
17	18	19	20
25	26	27	28

5	6	7	8
13	14	15	16
21	22	23	24
29	30	31	32

1	2	3	4
9	10	11	12
17	18	19	20
25	26	27	28

5	6	7	8
13	14	15	16
21	22	23	24
29	30	31	32

33	34	35	36
41	42	43	44
49	50	51	52
57	58	59	60

37	38	39	40
45	46	47	48
53	54	55	56
61	62	63	64

33	34	35	36
41	42	43	44
49	50	51	52
57	58	59	60

37	38	39	40
45	46	47	48
53	54	55	56
61	62	63	64

APÊNDICE II – Arquivo Exemplo com os Dados do Particionamento da Matriz B

Matriz 8x8

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Blocos 4x4 da Matriz B Particionada – Os blocos se repetem nas posições de leitura do engenho de busca para a multiplicação com os blocos da matriz A.

1 9 17 25
2 10 18 26
3 11 19 27
4 12 20 28

33 41 49 57
34 42 50 58
35 43 51 59
36 44 52 60

5 13 21 29
6 14 22 30
7 15 23 31
8 16 24 32

37 45 53 61
38 46 54 62
39 47 55 63
40 48 56 64

1 9 17 25
2 10 18 26
3 11 19 27
4 12 20 28

33 41 49 57
34 42 50 58
35 43 51 59
36 44 52 60

5 13 21 29
6 14 22 30
7 15 23 31
8 16 24 32

37 45 53 61
38 46 54 62
39 47 55 63
40 48 56 64

APÊNDICE III – Arquivo Exemplo com os Dados do Particionamento da Matriz C

Matriz 8x8

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Blocos 4x4 da Matriz C Particionada – Os blocos se repetem nas posições de leitura do engenho de busca para a recepção dos blocos das matrizes multiplicadas.

1	2	3	4
9	10	11	12
17	18	19	20
25	26	27	28

1	2	3	4
9	10	11	12
17	18	19	20
25	26	27	28

5	6	7	8
13	14	15	16
21	22	23	24
29	30	31	32

5	6	7	8
13	14	15	16
21	22	23	24
29	30	31	32

33	34	35	36
41	42	43	44
49	50	51	52
57	58	59	60

33	34	35	36
41	42	43	44
49	50	51	52
57	58	59	60

37	38	39	40
45	46	47	48
53	54	55	56
61	62	63	64

37	38	39	40
45	46	47	48
53	54	55	56
61	62	63	64