PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PMEMD-HW: SIMULAÇÃO POR DINÂMICA MOLECULAR USANDO *HARDWARE* RECONFIGURÁVEL

ADILSON ARTHUR MOHR

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul

Orientador: Prof. Dr. Ney Laert Vilar Calazans

Porto Alegre
Janeiro de 2010

Dados Internacionais de Catalogação na Publicação (CIP)

M699P Mohr, Adilson Arthur
PMEMD-HW : simulação por dinâmica molecular usando hardware reconfigurável / Adilson Arthur Mohr. – Porto Alegre, 2010.
98 p.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Ney Laert Vilar Calazans.

1. Informática. 2. Arquitetura de Computador. 3. FPGA. 4. Simulação (Programação de Computadores). I. Calazans, Ney Laert Vilar. II. Título.

CDD 004.22

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "PMEMD-HW: Simulação por Dinâmica Molecular Usando Hardware Reconfigurável", apresentada por Adilson Arthur Mohr, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas Embarcados e Sistemas Digitais, aprovada em 22/03/10 pela Comissão Examinadora:

Prof. Dr. Ney Laert Vilar Calazans – Orientador — PPGCC/PUCRS

Prof. Dr. Fernando Gehm Moraes — PPGCC/PUCRS

Prof. Dr. Osmar Norberto de Souza — PPGCC/PUCRS

Prof. Dr. Eduardo Augusto Bezerra — UFSC

Homologada em 17/08/2010, conforme Ata No. 13/10 pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

AGRADECIMENTOS

Começo agradecendo a minha família e em especial aquela que hoje é minha esposa: Francielle. Obrigado por terem me estimulado a iniciar o mestrado; e, Amor, obrigado pelo amparo e suporte nos momentos de fraqueza durante o percurso do mestrado, que às vezes pareciam intransponíveis...

Gostaria de agradecer também ao meu orientador, Ney Calazans, por acreditar no meu trabalho, pela paciência e especialmente por permitir o desenvolvimento do mestrado mesmo sem que eu estivesse disponível em tempo integral – como eu preferia e tinha me proposto ainda durante as entrevistas. O progresso da obra não teve o mesmo andamento nem amplitude, mas chegamos lá – conseguimos a tão esperada integração com a Bioinformática e temos adquirido bastante conhecimento para continuar com essa produção!

Nesse ponto, meu obrigado ao professor Osmar Norberto de Souza por assinalar o problema que balizou este trabalho e pelo auxílio a compreendê-lo, e ao agora mestre Maicon Sartin, pela grande parceria nesta pesquisa. Graças ao empenho de todos, pudemos reunir as áreas do GAPH e LABIO.

Cumprimentos também aos colegas Taciano Rodolfo, Gerson Scartezzini e Edson Moreno (PUCRS) e Giovani Vizzotto (Digitel), todos a quem tanto interroguei durante o Curso e que sempre procuraram atender minhas sucessivas indagações.

Fico grato igualmente ao professor Fernando Moraes, por participar em todas as bancas que o convidei, apreciando meu trabalho sempre com contribuições importantes. A todos os demais professores e colegas de mestrado e do GAPH – foi um período de muito aprendizado e boas amizades.

Minha gratidão também à Digitel, pela flexibilização e diminuição de carga horária, e à DELL, pelo apoio financeiro dispensado para a realização desta Dissertação, entidades que, assim, contribuíram enormemente para que pudesse concretizá-la.

Obrigado a todos!

PMEMD-HW: SIMULAÇÃO POR DINÂMICA MOLECULAR USANDO *HARDWARE* RECONFIGURÁVEL

RESUMO

Sistemas de dinâmica molecular são definidos pela posição e energia das partículas que o compõe, assim como pelas interações entre estas. Tais sistemas podem ser simulados através de métodos matemáticos como o cálculo de forças eletrostáticas baseadas na Lei de Coulomb. Computar os estados através dos quais um sistema destes evolui, avaliando a interação de cada partícula, é tarefa computacionalmente dispendiosa, mesmo para um número pequeno de partículas. Portanto, podem-se obter benefícios ao se aplicar técnicas específicas para acelerar tais computações. Enquanto alguns estudos propõem o uso de algoritmos diferenciados, existem os que empregam processadores especiais ou *hardware* personalizado, a técnica abordada nesta Dissertação.

Descreve-se aqui o projeto e a prototipação de uma arquitetura de *hardware* com potencial para acelerar uma aplicação que computa forças eletrostáticas entre partículas não ligadas. Dá-se ênfase especificamente aos aspectos da integração entre o *hardware* e a aplicação-alvo empregada neste projeto, o programa PMEMD (*Particle Mesh Ewald Molecular Dynamics*), parte da plataforma AMBER (*Assisted Model Building with Energy Refinement*). Os cálculos mais onerosos deste programa foram identificados e movidos para uma implementação de hardware em FPGA, criando um co-processador específico – o PMEMD-HW. A escolha de um *hardware* reconfigurável se deve, entre outros motivos, à facilidade de fazer evoluir o processo de projeto e obter a aceleração almejada.

A principal contribuição deste trabalho é o domínio da tecnologia de uso de co-processadores de *hardware* para acelerar aplicações nas áreas de Biologia e Biofísica. Um protótipo funcional está disponível, utilizando uma plataforma comercial de prototipação de *hardware*. Esta prova de conceito demonstra a viabilidade de usar com sucesso as técnicas desenvolvidas.

Palavras-chave: simulação por dinâmica molecular, FPGA, aceleração em *hardware*, desempenho, comunicação *hardware-software*.

PMEMD-HW: MOLECULAR DYNAMICS SIMULATION USING RECONFIGURABLE HARDWARE

ABSTRACT

Molecular dynamics systems are defined by the position and energy of their component particles, as well as by the interactions among these. Such systems can be simulated through mathematical methods like the computation of electrostatic forces based on the Coulomb Law. Predicting the states through which such system evolves by computing the interaction of each particle with its neighbors is a computationally costly task, even for a small number of particles. Thus, it can only be beneficial to apply specific techniques for accelerating these computations. While some studies propose the use of new algorithms, others advocate the use of specific processors or custom designed hardware, the later being the technique employed in this Dissertation.

This work describes the design and prototyping of a hardware architecture that has the potential to accelerate an application based on the computation of electrostatic forces among non-bonded particles. A special emphasis is given to the aspects of integration between the accelerating hardware and the modified target application, the PMEMD (Particle Mesh Ewald Molecular Dynamics) software, part of the AMBER (Assisted Model Building with Energy Refinement) platform. The costliest computations of PMEMD were identified and moved to an FPGA hardware implementation, creating a custom coprocessor – PMEMD-HW. The choice for reconfigurable hardware is due, among other reasons, to the ease with which it enables the evolution of the design towards the target acceleration.

The main contribution of this work is the mastering of the technology to design and analyze hardware coprocessors that target the acceleration of applications in Biology and Biophysics. A working prototype is available, using a commercial hardware prototyping platform. The proof-of-concept implementation demonstrates the viability of successfully using the proposed techniques.

Keywords: molecular dynamics simulation, FPGA, hardware acceleration, performance, hardware-software communication.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ACTG	Adenina, Citosina, Timina e Guanina
AMBER	*Assisted Model Building with Energy Refinement*
ASIC	*Application Specific Integrated Circuit*
BRAM	*Block RAM*
CI	Circuito Integrado
CPLD	*Complex Programmable Logic Device*
CPU	*Central Processing Unit*
DM	Dinâmica Molecular
DMA	*Direct Memory Access*
DSP	*Digital Signal Processor*
FACIN	Faculdade de Informática
FFT	*Fast Fourier Transform*
FPGA	*Field-Programmable Gate Array*
GAPH	Grupo de Apoio ao Projeto de *Hardware*
HDL	*Hardware Description Language*
HLL	*High Level Language*
IP	*Intellectual Property*
LJ	Lennard-Jones
LABIO	Laboratório de Bioinformática, Modelagem e Simulação de Biossistemas
LUT	*Look-Up Table*
MPP	*Massively Parallel Processing*
NaN	*Not a Number*
PC	*Personal Computer*
PME	*Particle Mesh Ewald*
PMEMD	*Particle Mesh Ewald Molecular Dynamics*
PMEMD-HW	*Particle Mesh Ewald Molecular Dynamics in Hardware*
PPGCC	Programa de Pós Graduação em Ciência da Computação

RAM	*Random Access Memory*
RNM	Ressonância Nuclear Magnética
RTR	*Run-Time Reconfiguration*
SMP	*Symmetric Multiprocessing*
SPLD	*Simple Programmable Logic Device*
SPME	*Smooth PME*
VV	*Velocity Verlet*

SUMÁRIO

1 INTRODUÇÃO

O conhecimento dos problemas da Biologia que podem ser tratados via ferramentas da Bioinformática é parte do novo enfoque de pesquisas e atividades para biólogos e profissionais das Ciências Exatas e da Computação que buscam desenvolver soluções para os questionamentos biológicos [AND05]. Nesse intuito, este projeto pretende contribuir para o desenvolvimento da Biologia e da Biofísica Molecular.

Segundo Hansson, Oostenbrink e van Gunsteren [HAN02], praticamente todas as estruturas biomoleculares obtidas por cristalografia de raios X ou espectroscopia por ressonância nuclear magnética (RNM) são refinadas por dinâmica molecular. Os dados obtidos em experiências com RNM são médias de todas as moléculas de um recipiente e de um dado tempo. Para conseguir reproduzir tais médias, as simulações devem ser longas o bastante para produzir uma amostra das conformações relevantes do sistema de um modo adequado.

Por exemplo, há propriedades como o cálculo da constante dielétrica de uma solução de proteínas, em que uma simulação por dinâmica molecular de cinco nanossegundos não se mostrou suficiente para alcançar resultados satisfatórios [HAN02]. Além disso, proteínas maiores precisam de um tempo de simulação mais longo. Quanto maior o tempo de simulação exigido, maiores serão os requisitos em termos de poder computacional. Assim, ao criar um *hardware* específico que propicie ganhos em tempo de simulação é possível habilitar a obtenção de resultados mais confiáveis e com maior rapidez.

Outro exemplo é o cálculo das energias livres, feito a partir de simulações por dinâmica molecular, que pode ser uma ferramenta poderosa na elaboração de drogas com auxílio de computadores [HAN02]. Nesse quesito, a farmacologia pode empregar técnicas de modelagem molecular, objetivando a predição da geometria e afinidade da ligação proteína-proteína ou da ligação de pequenas moléculas a proteínas. A docagem molecular tem por objetivo identificar o local de ligação na proteína, determinando a posição e orientação do ligante, e estimar a afinidade da ligação. Ao se considerar a possibilidade de flexibilidade em ambas (ligante e proteína) as partes haverá uma complicação significativa nos cálculos de docagem molecular [LES08] – mais uma razão para buscar a melhoria no desempenho de programas de docagem molecular, por meio de um *hardware* específico ou qualquer outro método.

O presente trabalho foi desenvolvido no contexto de dois grupos de pesquisa do Programa de Pós-Graduação em Ciência da Computação (PPGCC) da Faculdade de In-

formática (FACIN) desta universidade: o Grupo de Apoio ao Projeto de *Hardware* (GAPH) e o Grupo de Biofísica Molecular Computacional do Laboratório de Bioinformática, Modelagem e Simulação de Biossistemas (LABIO). A realização desse trabalho visa à capacitação para desenvolver soluções de alto desempenho para a execução da simulação por dinâmica molecular com o ambiente AMBER, e tem como premissa a criação de uma plataforma de aceleração baseada em *hardware* reconfigurável atuando como co-processador do programa PMEMD.

1.1 Objetivos

Esta Dissertação teve como objetivo primário o desenvolvimento de uma arquitetura de *hardware* em FPGA que executará as funções mais exigentes em termos de recursos computacionais do programa PMEMD de simulação por dinâmica molecular – os cálculos das funções de energia. A contribuição está centrada no projeto e implementação do referido *hardware,* da comunicação entre esse *hardware* e o *software* de simulação e na análise do tempo total de execução com e sem o co-processador.

Trabalhos paralelos, tais como os revisados no Capítulo 5, têm como alvo abordar a adaptação de *software* padrão de simulação por dinâmica molecular ao uso de *hardware* dedicado para sua aceleração. Neste âmbito, o *software* de simulação empregado nesta Dissertação é o ambiente AMBER [CAS05]. Em particular, visa-se o uso do programa PMEMD desse ambiente, que dá suporte à execução das simulações mencionadas. Os objetivos estratégicos atingidos no escopo deste mestrado compreendem:

1) O domínio do processo de projeto de *hardware* para a aceleração de sistemas de biofísica molecular computacional;
2) O conhecimento do processo de comunicação *software* x *hardware* em sistemas compostos de um computador hospedeiro e um *hardware* dedicado, com o protocolo de comunicação proposto;
3) Proposta e implementação de uma infra-estrutura para habilitar a aceleração de atividades de simulação por dinâmica molecular via *hardware* dedicado baseado em FPGAs.

Para alcançar tais objetivos, as seguintes metas foram atingidas:

- ✓ Domínio dos algoritmos do programa PMEMD, identificando as partes migráveis de maneira eficiente para o *hardware*;
- ✓ Compreensão das necessidades de dados a serem transferidos entre o PMEMD e o *hardware* dedicado;

- ✓ Projeto do *hardware* dedicado para executar os algoritmos indicados no passo acima;
- ✓ Definição da comunicação (transferência de dados e informações de controle) entre a plataforma criada e o ambiente de DM;
- ✓ Implementação de mecanismos de comunicação e controle;
- ✓ Obtenção de dados específicos de desempenho na comunicação entre o computador hospedeiro e plataforma co-processadora;
- ✓ Disponibilização dos dados de desempenho na comunicação;
- ✓ Disponibilização dos dados de desempenho global com comunicação e processamento na plataforma desenvolvida.

1.2 Estruturação do Trabalho

Uma vez introduzida a motivação deste trabalho e os objetivos a que se propõe, expõe-se a seguir como o restante está apresentado. O Capítulo 2 define como a biologia e a computação se interligam, formando a bioinformática. Já o Capítulo 3 trata de um problema específico dessa área, a simulação por dinâmica molecular. Traz a história da técnica e as leis matemáticas que a dirigem.

O Capítulo 4 faz uma abordagem sobre como um melhor desempenho pode beneficiar aplicações de DM, mostrando as vantagens da aceleração por meio de *hardware* especializado. Ainda, explica a respeito das necessidades para projetar um *hardware* reconfigurável, contextualizando as necessidades de planejamento para a implementação. O Capítulo 5 faz uma revisão dos artigos mais atuais envolvendo as técnicas de aceleração de DM, relacionados aos objetivos deste mestrado.

Os Capítulos 6 e 7 tratam especificamente desta pesquisa. O primeiro elenca os recursos escolhidos mediante análises justificadas no próprio Capítulo. A segunda mostra a maior contribuição deste mestrado, um co-processador criado para executar uma parte do programa PMEMD. Descreve a arquitetura criada, detalhando as necessidades e peculiaridades de cada módulo. Por fim, entrega os resultados do protótipo desenvolvido, validando sua funcionalidade. O último Capítulo traz considerações sobre este trabalho e apresenta diversos tópicos passíveis de pesquisa a partir deste estudo.

2 BIOLOGIA E COMPUTAÇÃO – A BIOINFORMÁTICA

A biologia é definida, segundo Gibas [GIB01], como o estudo dos seres vivos, desde a interação das espécies e populações às funções dos tecidos e células em um organismo individual. Quanto à bioinformática, sua definição é dada por Luscombe [LUS01] como a conceitualização da biologia em termos de moléculas e a aplicação de técnicas computacionais (como matemática aplicada, ciências da computação e estatísticas) para entender e organizar as informações associadas às macromoléculas biológicas.

A bioinformática se mostra como um elo entre Biologia e Computação, despontando como uma proposta concreta para a montagem e interpretação dos segredos da vida (Alexandre Queiroz *apud*: [AND05]). É um subconjunto de um campo maior da biologia computacional – a aplicação de técnicas analíticas quantitativas à modelagem de sistemas biológicos.

O surgimento da bioinformática tem duas datas consideradas. Alguns ponderam seu nascimento com a publicação do *Atlas of Protein Sequences and Structure* em 1965 – um catálogo que continha todas as seqüências de proteínas conhecidas até então, criado a partir dos trabalhos multidisciplinares sobre evolução biomolecular feitos por Margaret Oakley Dayhoff. Outros crêem que foi mais tarde, quando o TIGR (*The Institute for Genomics Research*) publicou o genoma completo da bactéria *Haemophilus influenza* em 1995. Tal diferença de datas vem de uma longa espera até ser possível o entendimento do alfabeto do DNA nos anos noventa – o que só foi conseguido graças à criação de computadores eficientes para a leitura do enorme volume de seqüências A, T, C e G, que gerou grande impulso para o desenvolvimento da bioinformática [AND05].

A fusão desses tópicos é possível, pois a vida pode ser considerada como uma tecnologia da informação: a fisiologia dos organismos é determinada pelos genes, que podem ser vistos, basicamente, como informação digital. Concomitantemente, o poder de processamento, armazenagem e a internet revolucionaram os métodos de acesso e troca de dados [LUS01], o que foi de extrema importância para a bioinformática.

Como se pode observar, a bioinformática não seria possível sem os avanços nas áreas de *hardware* e *software*. A capacidade de processamento cada vez maior e os meios de armazenamento rápidos e de grande volume se unem a métodos matemáticos e estatísticos sofisticados, necessários para se obter resultados com maior precisão e velocidade [LES08] [LUE04]. Os programas na área de bioinformática são utilizados para:

- Fazer inferências a partir de dados obtidos da biologia molecular moderna;

- Fazer conexões entre elas;
- Derivar predições importantes e relevantes.

Com tal progresso, existe hoje um crescimento exponencial de conhecimento, observável pela quantidade de dados que estão disponíveis em bases como o GenBank (com 106,5 bilhões de pares base de DNA e 108,4 milhões de seqüências genéticas em 2009) [NCB10] e o Protein Data Bank (PDB, com quase 63.000 estruturas em 2010) [PDB10] – Figura 1. Além dessas, existem diversas outras bases como o NDB, CATH, SCOP, SWISS-PROT, EMBL, e o TrEMBL [LUS01], [AND05].

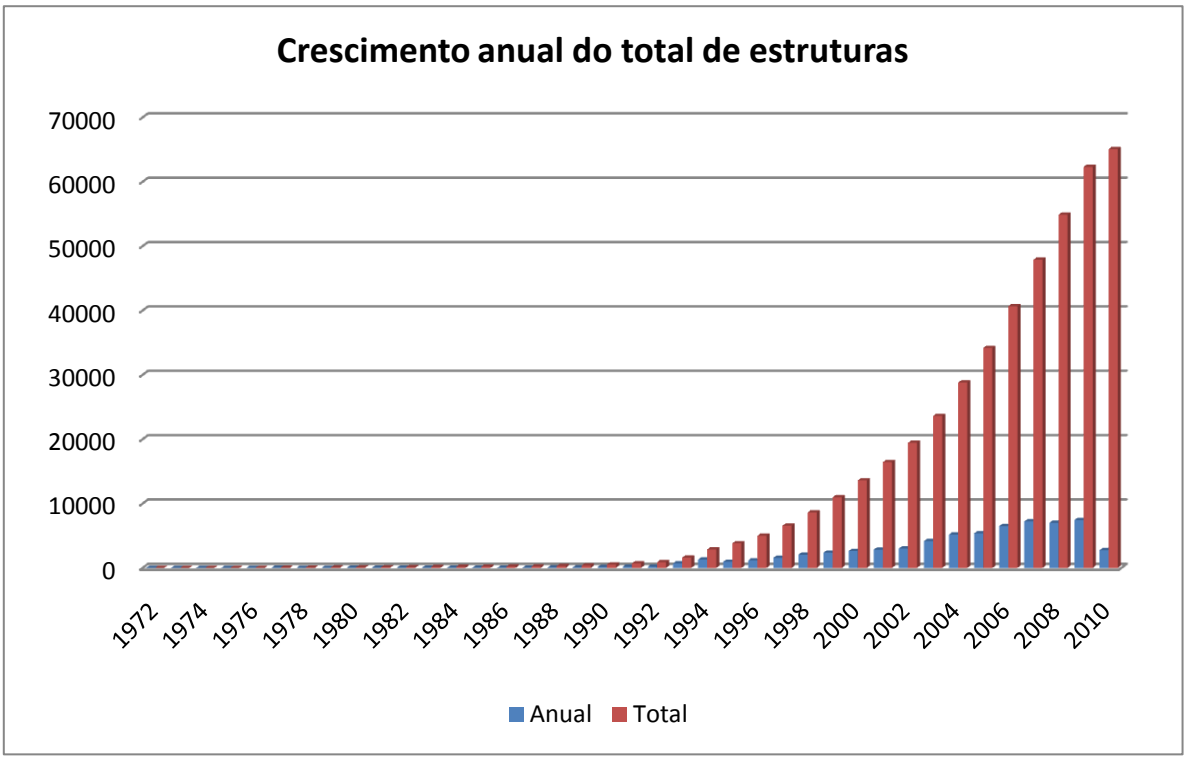


Figura 1 – Crescimento anual do total de estruturas disponíveis no PDB [PDB10].

Com tantos dados, métodos computacionais se tornaram indispensáveis nas investigações biológicas. O que começou com uma simples análise de seqüências, hoje engloba as mais diversas matérias. De um modo principal, são duas abordagens: comparação e agrupamento de informações e seu estudo para inferir e compreender suas relações. Como conseqüência do auxílio computacional, a bioinformática alcançou sucesso ao aumentar a profundidade das investigações biológicas [LUS01].

Está claro que, independentemente da aplicação, sempre há uma crescente demanda por métodos de simulação, sua precisão e exatidão [KAR02]. O campo da simulação molecular, assim como qualquer outro, se beneficia enormemente de avanços no *hardwa-*

re computacional. A título de comparação, Hansson [HAN02] citava, em 2002, o artigo de Duan e Kollman (*Pathways to a protein folding intermediate observed in a 1-microsecond simulation in aqueous solution*), de 1998, com simulações na ordem de um microssegundo. Hoje, Shaw *et al.* [SHA07] buscam criar uma máquina para simulações que cheguem a ordem de um milissegundo. Utilizando um computador convencional (como um Pentium Core 2 Duo), apenas um nanossegundo pode levar 10 dias ou mais para ser simulado.

3 SIMULAÇÃO POR DINÂMICA MOLECULAR

Se for possível (e viável) alterar um sistema para verificar seu funcionamento sob novas condições, normalmente será desejável fazê-lo. Entretanto, muitas vezes o custo é proibitivo, o que demanda um modelo do sistema, que pode ser físico (como um modelo em escala de um edifício) ou matemático (que representa a realidade com expressões lógicas e quantitativas que são manipuladas por métodos matemáticos para saber como o sistema deve reagir). Para o segundo, existem soluções analíticas (que produzem o resultado exato) ou simulações (quando o sistema é muito complexo para admitir uma solução analítica) [LAW00]. No caso da dinâmica molecular, além do alto custo e da complexidade, a velocidade com que se dão as alterações nas conformações moleculares é tal (simulações na ordem de nano ou microssegundos, com passos na ordem de femtossegundos [SCR06]) que apenas sua simulação pode trazer respostas a determinados problemas.

A Dinâmica Molecular (DM) é uma técnica para modelar movimentos e interações entre átomos ou moléculas, que está em evolução desde 1957 (Alder e Wainwright *apud* Sheraga, Khalili e Liwo [SCH07]). Um dos marcos em seu desenvolvimento foi a simulação por DM de uma proteína em 1977 (McCammon e Karplus *apud* Hansson, Oostenbrink e van Gunsteren [HAN02] e Karplus e Mccammon [KAR02]) – os resultados desse trabalho mudaram a visão de proteínas como estruturas rígidas, mostrando que são sistemas dinâmicos onde os movimentos internos e as mudanças conformacionais resultantes possuem um papel importante nas funções das proteínas.

Atualmente, essa técnica se tornou uma ferramenta estabelecida no estudo de biomoléculas, complementar às técnicas experimentais, auxiliando no entendimento das bases físicas das estruturas e funções biológicas de macromoléculas [KAR02] [HAN02]. Conforme apontado anteriormente, para a DM é preferível utilizar simulações a experimentos, pois se o intuito é alcançar questões específicas sobre as propriedades do sistema a ser modelado, simulações podem prover uma grande quantidade de detalhes a respeito do movimento das partículas. Um exemplo para esse caso é a busca por inibidores, ativadores e outros ligantes que permitam o desenvolvimento de fármacos [LES08] [PRO02].

As abordagens de simulação por dinâmica molecular padrão modelam proteínas como um conjunto de massas pontuais (átomos) conectados por ligações, em estruturas tridimensionais. As ligações, ângulos e forças entre átomos formam um sistema molecular – onde cada conjunto de átomos possui uma posição definida no espaço tridimensional –

no qual se adiciona energia por meio de aquecimento simulado. Então, regras de mecânica newtoniana e molecular e cálculos eletrostáticos são aplicados.

Segundo as leis newtonianas, a energia adicionada ao sistema oferece uma força oposta que move os átomos na molécula para fora de suas conformações padrão. As ações e reações de centenas de átomos em um sistema molecular podem ser simuladas usando essa abstração [GIB01] [LUS01].

As simulações por DM requerem a execução de um grande número de tarefas, como o cálculo de forças dos átomos que estão quimicamente ligados (em inglês: *bonded*) e dos átomos que não estão ligados covalentemente (todos os outros – em inglês: *non-bonded*), assim como ângulos, torção e atualização das velocidades e posições de cada átomo ou molécula do sistema sendo simulado. Dentre essas tarefas, o cálculo de forças dos átomos não ligados tem alta demanda computacional, pois um sistema de tamanho médio pode chegar a milhões de iterações para átomos não ligados, comparada a algumas milhares de iterações para os ligados e para ângulos [ALA07] [GU06] [YAN07].

Cada uma dessas tarefas permite calcular a energia potencial (V) do sistema, baseada nas leis de mecânica clássica de Newton. Sua fórmula pode ser vista em (1), a qual é dividida em várias partes, que são explicadas citando principalmente o trabalho de Pascutti em [PAS02]. A Figura 2 e a Figura 3 demonstram onde tais cálculos são aplicados.

$$
\begin{aligned}
V(r_i) = V(r_1, r_2, \ldots, r_{Nat}) = & \sum_{n=1}^{N_b} 1/2 K_{bn}(b_n - b_{0n})^2 + \sum_{n=1}^{N_\theta} 1/2 K_{\theta n}(\theta - \theta_{0n})^2 + \\
& + \sum_{n=1}^{N_\xi} 1/2 K_{\xi n}(\xi - \xi_{0n})^2 + \sum_{n=1}^{N_\varphi} K_{\varphi n}[1 + \cos(n_n \varphi_n - \delta_n)] + \\
& + \sum_{i<j}^{N_{at}} [(C_{12}(i,j)/r_{ij})^{12} - (C_6(i,j)/r_{ij})^6 + q_i q_j / 4\pi\varepsilon_0 \varepsilon r_{ij}]
\end{aligned}
\tag{1}
$$

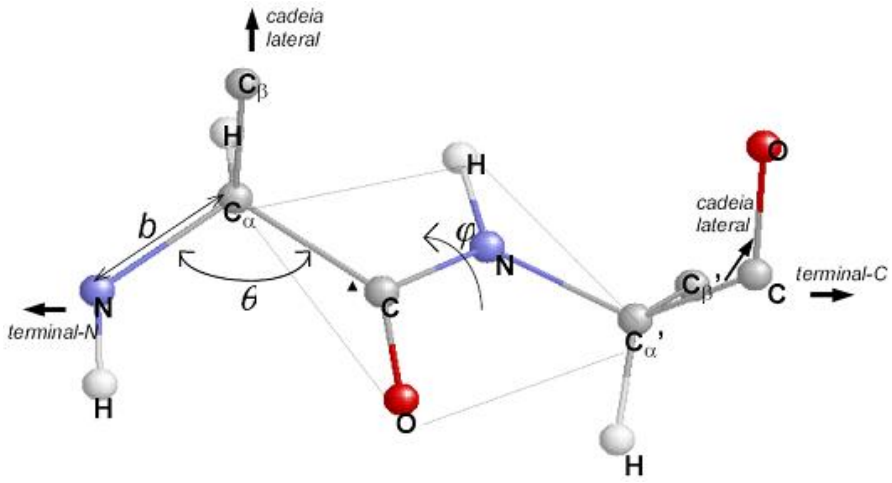


Figura 2 – Esqueleto peptídico para demonstração dos cálculos de DM.
Onde *b* é o comprimento das ligações químicas, *θ* é o ângulo entre duas ligações consecutivas e *φ* é o ângulo torcional para ligações com liberdade de rotação [PAS02].

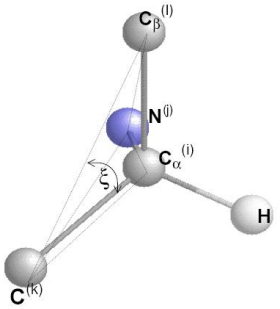


Figura 3 – Estrutura tetraédrica do carbono-*α*.
Onde *ξ* é o ângulo entre os planos *i_j_k* e *j_k_l* [PAS02].

Inicialmente são calculadas as oscilações de comprimento das ligações químicas, conforme (2), onde K_b é a constante de Hook, *b* é o comprimento da ligação e b_0 seu comprimento de equilíbrio. Da mesma forma, as oscilações dos ângulos de tais ligações podem ser descritas por um potencial harmônico (3), onde *θ* é o ângulo entre duas ligações químicas consecutivas e K_θ a constante de Hook para retornar ao ângulo de equilíbrio θ_0. Para um conjunto de quatro átomos é necessário um terceiro potencial harmônico (4), onde *ξ* é o plano de fundo formado pelos átomos *C*, C_α e C_β e o plano formado pelos átomos *C*, C_β e *N* na Figura 3, e K_ξ é a constante para a restituição ao ângulo de equilíbrio ξ_0 entre os planos. Considerando ainda que existam rotações, ou torções, em torno das ligações químicas, deve-se calcular o Potencial Diedral Próprio. Esse potencial é definido por (5), em que "K_φ é a constante que determina a altura da barreira de rotação, *n* o número de mínimos para a torção de uma ligação química específica, *φ* o ângulo diedral para a ligação central em uma seqüência de quatro átomos e *δ* é a defasagem no ângulo diedral que pode colocar na posição *φ* j igual a zero um ponto de máximo ou de mínimo" [PAS02].

$$V_b = \frac{1}{2} K_b (b - b_0)^2 \qquad (2)$$

$$V_\theta = \frac{1}{2} K_\theta (\theta - \theta_0)^2 \qquad (3)$$

$$V_\xi = \frac{1}{2} K_\xi (\xi - \xi_0)^2 \qquad (4)$$

$$V_\varphi = K_\varphi [1 + \cos(n\varphi - \delta)] \qquad (5)$$

As funções de (2) a (5) se referem apenas às interações entre átomos ligados covalentemente. Para todas as outras, os potenciais são dados pela lei de atração de van der Waals, avaliada pelo Potencial de Lennard-Jones – LJ (dado por (6), onde "ε é a profundidade do poço entre a barreira atrativa e a repulsiva e σ é o diâmetro de Lennard-Jones" e r é a distância entre os núcleos), e pela Lei de Coulomb (em (7), onde "q_i e q_j são as cargas residuais sobre os átomos i e j, separados pela distância r, ε_0 é a permissividade do espaço livre e ε é a constante dielétrica que corrige ε_0 para considerar a polarizabilidade do meio."). O potencial LJ é calculado para cada possível par de partículas que interagem, obtendo a força que age sobre cada partícula. Já a "interação eletrostática varia com o inverso da distância de separação entre os átomos, sendo, portanto, de longo alcance" [PAS02].

$$u_{LJ}(r) = 4\varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] \qquad (6)$$

$$V_{Coul.} = \frac{q_i q_j}{4\pi\varepsilon_0 \varepsilon r} \qquad (7)$$

Como potenciais e forças são aplicados às moléculas próximas, quando uma força tem curto alcance é inserido um ponto de corte, a partir do qual se considera que as partículas não interagem. Esses cálculos são $O(n^2)$ para um dado número de partículas n – se tornando promissor para aceleração. Para forças de longo alcance também pode ser utilizado um ponto de corte, que diminui consideravelmente a quantidade de cálculos neces-

sários ao mesmo tempo em que introduz um erro – que pode ser aceitável (como nas forças de LJ) ou não (como nas forças de Coulomb) [GU07b]. Para resolver esse problema existem métodos como o *Particle Mesh Ewald* (PME), que divide os somatórios infinitos usados em Coulomb em duas partes – ambas convergindo rapidamente [CAS05] – ou o método *multigrid*, que tem a vantagem de não precisar das transformadas rápidas de Fourier (FFT) [GU07a].

Outro ponto que pode ser analisado é o algoritmo de atualização de velocidade e posição de Verlet (VV). Apesar desse algoritmo também ser $O(n^2)$, o que pode sugerir a necessidade de aceleração, o VV é executado apenas uma vez por partícula e não caracteriza um gargalo, do mesmo modo que a inicialização dos dados, a construção da lista de vizinhos e os cálculos de termos covalentes e ligações de hidrogênio (que são $O(n)$).

Apesar dos artifícios apresentados acima para cálculos de forças de átomos não ligados, esses ainda contribuem para quase a totalidade do tempo de execução de uma simulação. Assim, pela lei de Amdhal, são candidatas para aceleração por *hardware*. Os meios para obter um melhor desempenho são endereçados nos capítulos seguintes.

3.1 Evolução

Conforme Scheraga [SCH07], já se passaram 50 anos desde a primeira simulação por dinâmica molecular. Segundo ele, pode-se considerar que a era moderna dos cálculos de dinâmica molecular, utilizando computadores, começou com os trabalhos de Alder e Wainwright, que escreveram os artigos *Phase transition for a hard-sphere system* e *Molecular dynamics by electronic computers* em 1957 e 1958. Em tais artigos são calculadas as propriedades de equilíbrio e não-equilíbrio de centenas de partículas, obtendo, finalmente, a equação de estado (pressão e volume) e a distribuição de velocidade de Maxwell-Boltzmann. Na seqüência, Scheraga cita o trabalho de Rahman (*Correlations of the motion of atoms in liquid argon*), que faz a primeira simulação por DM de um sistema real ao estudar a dinâmica do argônio líquido em 1964.

Já a publicação da primeira simulação por dinâmica molecular de uma proteína foi feita por McCammon e Karplus na revista Nature, em 1977, com o artigo *Dynamics of folded proteins*. Tal simulação envolveu uma molécula de proteína no vácuo e durou apenas 9.2 ps. (Hansson [HAN02] afirma que foram 8.8 ps.). Apesar do período curto, os resultados mudaram a visão de proteínas como estruturas rígidas, mostrando que são sis-temas dinâmicos onde os movimentos internos e as mudanças conformacionais resultantes possuem um papel importante nas funções das proteínas [HAN02], [KAR02], [SCH07].

Desde então, diversos avanços em direção a sistemas maiores e tempos de equilíbrio mais longos foram feitos ao incluir uma descrição mais realística do ambiente. Dentre eles, Karplus [KAR02] cita quatro:

- Inclusão de moléculas solventes explícitas ao redor da proteína (usualmente, água);
- Adição de contra-íons (*counterions*);
- Tratamento mais realístico dos limites do sistema;
- Tratamento mais preciso das forças eletrostáticas de longo alcance.

Em muitos casos, é mais fácil utilizar simulações do que experimentos para determinar certas propriedades de um sistema a ser modelado, pois simulações podem prover uma grande quantidade de detalhes sobre o movimento das partículas. Inclusive, em diversos aspectos das funções biomoleculares, são justamente esses os detalhes de interesse. Outra característica significativa da simulação é que os potenciais usados estão sob controle do usuário, de modo que, ao removê-los ou alterá-los, suas contribuições em determinar certa propriedade podem ser examinadas [KAR02].

Entretanto, de modo algum os experimentos devem ser esquecidos. A experimentação tem um papel fundamental na validação, comparando os resultados obtidos no experimento com o que foi simulado para testar a precisão da simulação e prover critérios de melhorias na metodologia [KAR02].

3.2 Aplicações

Simulações por dinâmica molecular possuem diversas aplicações. Hansson [HAN02] cita três:

- “Dar vida” a estruturas moleculares, trazendo a compreensão da dinâmica natural de biomoléculas;
- Oferecer médias térmicas das propriedades moleculares – após a simulação de uma molécula com seu ambiente por um período de tempo, pode-se dispor das propriedades moleculares médias que se aproximam das médias obtidas experimentalmente;
- Descobrir quais conformações de uma molécula ou complexo estão termicamente acessíveis – usado em aplicações de docagem de ligandos (*ligand-docking*), por exemplo. Como já mencionado, aplicações que se preocupam com docagem de proteínas são particularmente importantes na fabricação de

fármacos. Inclusive, o principal método para descobrir os processos de algumas doenças é a simulação por DM [GU06].

Ele ainda menciona o fato de que, se os dados de experimentos forem traduzidos de modo a guiar os cálculos de dinâmicas, a simulação pode combiná-los com as propriedades gerais das estruturas moleculares. Curiosamente, [KAR02] também cita três aplicações gerais para a simulação por DM:

- Usar apenas como um meio para amostragem de configuração espacial;
- Obter a descrição de um sistema em equilíbrio, incluindo propriedades estruturais e de movimento e os valores dos parâmetros de termodinâmica;
- Analisar a dinâmica realmente, visualizando o desenvolvimento do sistema durante o tempo.

Para as duas primeiras áreas pode-se utilizar simulação de Monte Carlo ou dinâmica molecular. Já a última, onde os movimentos e seu desenvolvimento no tempo são o interesse principal, apenas a DM pode ser empregada. Os três tópicos têm demandas crescentes por métodos de simulação mais precisos e fidelidade [KAR02].

Simulações por DM também são usadas para encontrar caminhos entre conformações abertas e fechadas, que é impossível de determinar experimentalmente. Entretanto, a transição entre uma estrutura aberta e fechada leva em torno (acredita-se) de um milissegundo, o que inviabiliza uma simulação direta desse movimento [KAR02].

O processo de simulação do enovelamento de proteínas em formatos específicos é tão computacionalmente dispendioso que a tarefa foi incumbida para a intuição humana, com a criação de um jogo. Pesquisadores liderados por David Backer criaram o jogo FoldIt (http://fold.it/), afirmando que "as pessoas, usando a sua intuição, poderão ser capazes de encontrar o caminho para a resposta correta muito mais rapidamente" do que milhares de computadores. Outras iniciativas incluem projetos como o folding@home (http://folding.stanford.edu/) ou o rosetta@home (http://boinc.bakerlab.org/rosetta/), que aproveitam o tempo ocioso de processamento dos computadores – incluindo até mesmo *chips* gráficos do vídeo-game PlayStation 3 – com os mesmos objetivos do FoldIt, nos moldes do Seti@home.

4 SOLUÇÕES PARA ACELERAÇÃO

Como qualquer outro ramo da ciência, o campo da simulação por dinâmica molecular se beneficia dos ganhos de desempenho obtidos com as melhorias no *hardware* dos computadores. Se antigamente só era possível executar tais simulações em supercomputadores, hoje essas são efetuadas em computadores de mesa comuns. A título de comparação, Hansson, Oostenbrink e van Gunsteren [HAN02] mostraram simulações na ordem de um microssegundo em 1998, enquanto atualmente há pesquisas como a de Shaw *et al.* [SHA07] que buscam criar máquinas que permitam atingir a ordem de um milissegundo de simulação. Outro exemplo é o uso de um computador convencional (possuindo um processador Pentium Core 2 Duo), com o qual é necessário um período de 10 dias ou mais para executar a mesma tarefa de Shaw *et al*. Para a simulação por DM, tal progresso permite tempos de prova e equilíbrio maiores. Isso possibilita a otimização dos campos de força para reproduzir propriedades mais complexas que são baseadas em médias calculadas nas simulações (ver Capítulo 3).

Com o aumento na capacidade de processamento, sistemas maiores e mais complexos se tornam acessíveis [HAS07]. Assim, todo o ganho de tempo é reinvestido no estudo de sistemas maiores que incluem, por exemplo, um solvente explícito e/ou um ambiente com membrana. Ainda graças ao fator tempo, pode-se executar diversas vezes a mesma simulação para obter estimativas de erros de estatísticas [KAR02].

Para melhorar ainda mais o desempenho das aplicações, é possível utilizar um recurso que é cada vez mais comum: a aceleração por *hardware* dedicado. Hasan, Al-Ars e Vassiliadis [HAS07] a definem como o uso de um componente especializado para executar alguma função mais rápido do que é possível com um *software* sendo executado em uma CPU de propósito geral. Se tal *hardware* for uma unidade separada da CPU, pode ser chamado de *hardware* de aceleração. O objetivo, nesse caso, é diminuir a carga do processador principal, obtendo ganhos de desempenho.

4.1 Dados vs. CPU

Existem basicamente dois tipos de algoritmos que podem ser levados em conta quando se pensa em aumento de desempenho: os que têm troca intensiva de dados (*data intensive*) e os que necessitam de computação intensiva (*CPU intensive*). Algoritmos que têm cálculos simples e curtos, mas que trabalham com uma quantidade muito grande de dados, sobrecaem no primeiro grupo. Seu gargalo está no tempo que é perdido lendo informações em disco e salvando os resultados finais. Já algoritmos que levam muito mais

tempo calculando do que buscando e guardando dados ficam no segundo grupo (também chamado de *compute bound*) [LUE04] – caso das simulações por dinâmica molecular, segundo Gu e Herbordt [GU07b].

Quando há muitas operações de entrada e saída (ou essas são consideravelmente lentas), algoritmos ficam amarrados a sistemas que provêem sinais externos – por isso são chamados *I/O bound*. Programas de troca intensiva de dados são bons exemplos (pois há muito acesso ao disco), sem serem únicos – se os dados são coletados de sensores, por exemplo, o tempo de acesso a suas informações faz com que o programa fique muito tempo em espera, caracterizando-se como *I/O bound*. Nesse caso, métodos de aceleração têm pouco valor [LUE04]. Ao contrário de Gu e Herbordt, Yang, Mou e Dou [YAN07] não classificam simulações por DM apenas como tendo computação intensiva, pois há igualmente porções de troca intensiva de dados, tornando-as ideais para FPGAs.

4.2 Possibilidades de Aceleração

No caso de programas que precisam de muito poder computacional, pode-se obter aceleração com a otimização do uso de memória, maior eficiência do algoritmo, melhores bibliotecas, etc. Quanto ao *hardware*, uma maneira de se atingir um melhor rendimento é dividir os dados entre diversos processadores idênticos (o que é conhecido como *Symmetric Multiprocessing* – SMP) ou diversos computadores (o que é conhecido como agregado ou grade, dependendo da topologia, velocidade da rede e sua administração). Nesse caso, o programa deve ser modificado para que se quebre o problema em pequenos pedaços que serão enviados para os elementos processadores, devolvidos e combinados [LUE04]. Esse tipo de abordagem é conhecida como “dividir para conquistar”.

Agregados e grades são flexíveis e têm alta vazão, mas exigem um maior cuidado em sua coordenação e manutenção. SMPs têm tempo de resposta individual baixo, mas são caros comparados com um agregado que tenha uma vazão similar, na opinião de Luethy e Hoover [LUE04]. Hasan, Al-Ars e Vassiliadis [HAS07] citam um artigo que demonstra que SMPs têm a capacidade de solucionar problemas em biologia molecular que exijam computação intensa de modo eficiente e econômico. Seguindo a visão de Luethy e Hoover, percebe-se que há um espaço aberto para a aceleração com o uso de processamento especializado, como aquele que pode ser obtido pelo uso de *hardware* específico utilizando dispositivos como FPGAs ou ASICs. Tais *chips* são criados para efetuar cálculos específicos rapidamente – até três ordens de magnitude mais rápidos do que CPUs de uso geral, mas podem implicar custos adicionais de desenvolvimento e aquisição de materiais.

Uma vantagem de FPGAs é ser mais flexível do que o ASICs, pois o mesmo *chip* pode ser reconfigurado para receber uma nova funcionalidade tantas vezes quanto se deseje. Entretanto, tal flexibilidade tem um custo – seu desempenho é menor se comparado a um circuito implementado em um ASIC, já que os sinais elétricos não precisam passar por tantos transistores e alcança uma maior frequência de processamento.

Do lado financeiro, os preços de FPGAs e microprocessadores de ponta são comparáveis. Já a combinação PC/FPGA pode ter um custo total (*total cost of ownership*, TCO) bem menor do que agregados ou grades com 32 ou 64 nós. Mesmo agregados de baixo custo, se forem comparados com sistemas multi-FPGA de baixo custo, acabam se mostrando mais caros [GU06].

Para definir a melhor opção entre tais sistemas é necessário um *benchmark* público, sem o qual se torna difícil a tarefa de comparação e avaliação. Para simulações de DM existe um *benchmark* disponibilizado pelos desenvolvedores do programa AMBER, para analisar o próprio AMBER, CHARMM e NAMD (disponível em http://amber.scripps.edu/amber8.bench2.html), mas este não foi citado nos artigos examinados aqui. As métricas mais comuns são em segundos/dia de simulação (na verdade, dada a capacidade computacional atual, em picossegundos/dia) ou o tempo que leva para terminar um *time-slice* da simulação. Mas, mesmo com tais métricas, o problema de quantificar a melhora obtida persiste, pois alguns autores fazem comparações tendenciosas [SCR06].

Para quem implementa soluções, o problema está em definir o melhor algoritmo para a resolução do problema, uma vez que estes não são igualmente efetivos em plataformas distintas. Diferentes métodos devem ser escolhidos quando se emprega FPGAs, PCs, MPPs ou ASICs. Gu, Vancourt e Herbordt [GU06] lembram que, em um PC, técnicas baseadas em transformadas (como transformadas de Fourier, Ewald, PME, *Particle-Particle Particle-Mesh* (PPPM)) são preferíveis; em um MPP elas causam uma sobrecarga na comunicação, então simulações com ponto de corte são mais apropriadas; já em um FPGA o tamanho do problema pode ser muito maior ou até mesmo inexistente. Tais idiossincrasias acabam por sobrecarregar o projetista, ao exigir conhecimento de cada método para escolha do que melhor se adapta a sua realidade.

As Seções finais deste Capítulo exploram algumas características dos FPGAs como forma de acelerar o processamento de aplicações nas áreas alvo deste trabalho.

4.3 Utilização de FPGAs na Construção de *Hardware* Dedicado

Um FPGA é uma plataforma de *hardware* reconfigurável, onde um algoritmo pode

ser diretamente mapeado em elementos básicos de lógica, como portas NAND ou tabelas verdade de tamanho fixo [HAS07]. Em outras palavras, trata-se de um circuito integrado que pode ser reconfigurado após sua fabricação. Ambos, FPGAs e processadores são dispositivos de propósito geral. A diferenciação entre eles se dá na granularidade com a qual uma computação é implementada. Enquanto processadores usam como base para implementar computações um conjunto fixo de instruções definidas na abstração denominada *arquitetura do processador*, em um FPGA a base são dispositivos de *hardware* e interconexões. Isto acrescenta enorme flexibilidade a FPGAs em relação a processadores, ao custo de um processo de projeto mais complexo. FPGAs têm como características a quantidade de blocos lógicos ou de transistores, sua arquitetura interna (importante para aplicar algoritmos de reconfiguração do *hardware* em tempo de execução), velocidade e consumo. Praticamente todos FPGAs possuem elementos lógicos, *lookup tables*, memória, fios e outros recursos de roteamento e entradas e saídas configuráveis – que provêem interface com o mundo externo [PEL05].

Como o FPGA é passível de configuração, torna-se simples para o projetista criar sistemas que possuam somente e exatamente os módulos necessários para sua aplicação, sendo possível até mesmo criar módulos específicos. Por exemplo, sistemas de alto desempenho que perdem muitos ciclos de CPU executando determinada seção de um código, podem ter esse código reescrito em *hardware*. Para aproveitar bem as características desse dispositivo, o novo código deve ser desenvolvido preferencialmente com o máximo de paralelismo. Essa é uma técnica cada vez mais comum para o aumento de desempenho, pois o processamento em *hardware* específico é mais rápido do que se realizado puramente em *software*. Além disso, livra-se o processador principal para efetuar outras operações enquanto o periférico executa suas instruções concomitantemente. Assim, FPGAs se mostram como uma boa solução para várias classes de problemas na bioinformática [ALT04] [HAS07].

Apesar de todos os benefícios, uma aplicação em *hardware* implementado em FPGA é mais lenta e consome mais energia que a mesma aplicação criada utilizando um ASIC[1]. Sua frequência de operação é tipicamente bem inferior a de CPUs, o que pode ser amplamente compensado pela personalização do *hardware*. Esta diferença de frequência de operação está diminuindo gradativamente, pois arquiteturas como o FPGA Xilinx Vir-

[1] Basicamente, ASIC é um circuito integrado projetado para uma aplicação específica, mais do que para aplicações gerais – assim como um FPGA. Entretanto, diferentemente do FPGA, depois de criada a pastilha de silício não é possível alterá-la. Normalmente, o FPGA é utilizado no início do desenvolvimento do *hardware*, e futuramente, pode ser transformado em um ASIC – desde que haja demanda de fabricação que justifique seu custo.

tex5 já alcançam 550 MHz. Dentre as limitações de FPGAs, constam [GU07b]:

- Área de *chip* – o tamanho e a complexidade do código afetam diretamente o tamanho de área requerida para programar a aplicação no FPGA, diminuindo a capacidade para paralelismo e, assim, o desempenho;
- Projetista – aplicações complexas necessitam de maior experiência do desenvolvedor e mais tempo para mapeá-la corretamente no FPGA, de modo a obter resultados precisos com um algoritmo que não implique perda significativa na frequência alcançada pelo projeto;
- Lei de Amdahl – se o problema a ser acelerado não ocupar a maior parte do tempo de execução, pode ser necessária uma reestruturação mais profunda na aplicação para se obter uma boa melhora no desempenho;
- Componentes – componentes físicos embarcados no próprio FPGA, como multiplicadores, memórias e suporte a ponto flutuante, são peças-chave para uma boa implementação mas, quando existem, são em número reduzido;
- Operações aritméticas – esse pode ser um ponto crítico, uma vez que a precisão, o modo aritmético (inteiro ou ponto flutuante, precisão simples ou dupla) e as operações exigidas têm um custo em espaço utilizado e eficiência que, se não (re)projetadas corretamente, podem inviabilizar o novo sistema.

Considerando o menor risco, menor custo de desenvolvimento e a possibilidade de reconfiguração, FPGAs são uma excelente opção, substituindo gradualmente *hardware* de propósito geral e sendo usados inclusive como plataforma para a prototipação de ASICs [PEL05] [YAN07]. Além disso, FPGAs estão ficando quase tão poderosos quanto ASICs e microprocessadores, pelo fato de estarem dirigindo os processos tecnológicos e por se tornarem híbridos ao embutir componentes físicos como os mencionados anteriormente [GU07a].

4.4 Planejamento de Projetos com FPGA

Do mesmo modo que portar aplicações existentes para sistemas maciçamente paralelos (*Massively Parallel Processing* – MPP) ou em grade pode ser complexo, o mesmo processo envolvendo FPGA não é uma tarefa trivial – com alguns agravantes. Por exemplo, técnicas empregadas na paralelização de um código uniprocessado, transformando-o em MPP, são mais difundidas, compreendidas e suportadas do que o mesmo processo dirigido a FPGAs. Assim, um bom planejamento se faz necessário. Herbordt *et al.* [HER07] endereçam bem as necessidades de planejamento para desenvolvimento com

FPGA, elencando-as da seguinte forma (o artigo ainda fornece exemplos e suas soluções):

- Reestruturação da aplicação:
  a) Usar um algoritmo ótimo para FPGA – deve-se escolher um método ou algoritmo que seja o melhor para FPGAs, o que não significa que será o mesmo para MPP ou PC;
  b) Usar modos apropriados – como linguagens HDL especificam *hardware*, e não *software*, bons modos para programação de *software* nem sempre são bons ou se aplicam para FPGAs. Por exemplo, o uso de ponteiros não é aconselhável;
  c) Usar estruturas apropriadas – estruturas análogas a pilhas, árvores e filas no FPGA diferem das encontradas em *software*;
  d) A já mencionada Lei de Amdahl;
- Projeto e Implementação:
  a) Esconder a latência de funções independentes – usar paralelismo, em especial sobrepondo computação e comunicação;
  b) Remoção de gargalos igualando taxas de uso – funções mais lentas podem ser replicadas para obter a vazão desejada;
  c) Aproveitar o *hardware* embarcado no FPGA – cada vez mais componentes físicos (ASIC ou módulos *hard-wired*), como multiplicadores, são integrados aos FPGAs, o que melhora o desempenho do sistema final;
- Operações Aritméticas:
  a) Usar a precisão aritmética apropriada – definir a quantidade de *bits* das representações, conforme discorrido ao longo desta Dissertação;
  b) Usar o modo aritmético apropriado – inteiro ou ponto flutuante, de precisão simples ou dupla, também visto ao longo deste trabalho;
  c) Minimizar o uso de operações de custo elevado – um FPGA trata operações aritméticas de modo diferente de um PC normal. Multiplicações são mais eficientes que adições, enquanto divisões são extremamente lentas ou consomem muita área para terem melhor desempenho. Assim, deve-se reestruturar a parte matemática pensando no custo das funções;
- Sistema e Integração:

a) Criar famílias de aplicações, não soluções pontuais – apesar de não haver em HDLs um nível de parametrização comparável à orientação a objetos, deve-se levar em conta o reuso para diminuir tempo, custo e habilidade necessários para um novo projeto;

b) Escalar a aplicação para uso máximo do FPGA – instanciar o máximo de elementos processadores possíveis no FPGA.

c) O desempenho da aplicação não é sensível à qualidade da implementação – se uma aplicação consome 90% de seu tempo executando uma parte de seu código, reestruturar o código poderá melhorar a performance sensivelmente, sem alterar o resultado.

Como se pode perceber existe um grande potencial para aumento de desempenho em aplicações com o emprego de FPGAs – mas alcançá-lo demanda uma boa seleção da aplicação e de métodos de projeto para garantir um resultado flexível, escalável e portável. Uma vez definido o *hardware*, ele é produzido com o uso de uma linguagem de descrição de *hardware* (HDL), como VHDL e Verilog, ou arranjando blocos de funções pré-existentes (os chamados blocos de propriedade intelectual ou IP). Mais recentemente surgiram ferramentas de desenvolvimento que suportam até mesmo o uso de linguagens de mais alto nível, como C e Java. Uma vez definido o *hardware*, é gerado um arquivo *netlist* que será enviado ao FPGA, deixando-o pronto para uso [PEL05].

4.5 *Hardware* de Uso Exclusivo

Uma das maneiras de utilizar o FPGA é como um co-processador. A idéia é aproveitar a base de um algoritmo puramente escrito em *software* e substituir a parte que tem maior demanda computacional por estruturas de hardware específicas, descritas para um FPGA. Essa é a idéia que move este trabalho. Hasan, Oostenbrink e van Gunsteren [HAS07] citam o artigo de J. Shaw *et al*. (*Hardware accelerator for genomic sequence alignment*) para demonstrar o quanto é interessante essa técnica. Os resultados nesse artigo mostram uma melhora média de 287% no tempo de processamento.

4.6 Reconfiguração de FPGA

Dada a natureza reconfigurável do FPGA, foram criadas técnicas para que seja possível alterá-lo também em tempo de execução. Tais técnicas, coletivamente conhecidas como *run-time reconfiguration* (RTR), consistem em dividir a aplicação em operações que podem ser implementadas em estruturas de *hardware* separadas. Essa abordagem per-

mite uma alta performance, comparável com *hardware* dedicado. Apesar de não mostrar exatamente como a reconfiguração permitiu um ganho de desempenho, Hasan, Oostenbrink e van Gunsteren [HAS07] citam artigos que conseguiram melhoras de 64 a 330 vezes com o uso de RTR. Como exemplo, pode-se modificar o FPGA para que, assim que termine um tipo de cálculo, a parte responsável pelo cálculo seja alterada para auxiliar nos processos que possuam maior demanda [GU07b]. Outro ponto interessante é a possibilidade de permitir que a precisão das variáveis se torne parametrizável [GU06].

5 TRABALHOS RELACIONADOS

Para simulações por dinâmica molecular, há desde sistemas acadêmicos que utilizam um único FPGA, como os desenvolvidos por Gu, Vancourt e Herbordt em [GU06], [GU07a], [GU07b], e [GU08], até supercomputadores como o Blue Protein do Centro de Pesquisa em Biologia Computacional do Japão, que está na lista do Top500 e pode ser visto na Figura 4. No meio desses extremos existem outros sistemas, como o MODEL (que usa 76 pastilhas ASICs para calcular o potencial LJ e forças de Coulomb), o MD-GRAPE (que acelera apenas o cálculo de força e potencial, deixando o resto para o hospedeiro) e suas diversas variações, o MD Engine [GU06] [YAN07] e o Molecular Dynamics Machine (MDM, que também usa ASIC). O MDM é um dos primeiros trabalhos com computadores de propósito especial, desenvolvido por Narumi, Kawai e Koishi [NAR01]. Por fim, são relacionados também os sistemas SRC MAPStation, Cray XD1 e SGI RASC, em que os aceleradores são FPGAs conectados a microprocessadores normais [ALA07] [SCR06]. Há ainda o ANTON, um MPP com 512 ASICs montados em uma conformação torus 3D, que afirma ser o primeiro a possibilitar simulações na ordem de milissegundos [SHA07].

Antes de iniciar o desenvolvimento de um sistema de auxílio para a simulação por DM que utilize *hardware* reconfigurável, existem diversos pontos importantes que devem ser observados além do que foi elencado até aqui. Um dos principais é a necessidade de considerar a precisão e a representação de variáveis. A precisão em DM é alterável, dependendo do estágio que se está computando, sendo que o mínimo requerido é aquele que mantêm a simulação estável. Há duas métricas para determinar sua qualidade: a flutuação das quantidades físicas que deveriam manter-se constantes e a razão de flutuação entre a energia total e a energia cinética [GU07b] [YAN07].

Esse problema é endereçado por Gu, Vancourt e Herbordt em [GU06], [GU07a], [GU07b] e [GU08], lembrando que FPGAs são caracterizados pelo uso de dados de baixa precisão – provável motivo pelo qual a maioria dos pesquisadores evita aplicações que requerem unidades de ponto flutuante de dupla precisão, como as simulações por DM. Entretanto, nesse mesmo artigo é constatado que tal precisão nem sempre é necessária, inclusive citando outras pesquisas na área – apesar dessa visão não ser um consenso. A questão é que as implementações de DM usualmente são executadas por computadores que dão pouco incentivo para não empregar dupla precisão – o que é um problema no FPGA, pois, se for possível reduzir a precisão sem perda significativa de qualidade, é preferível empregar os recursos que sobram para melhorar o desempenho do sistema.

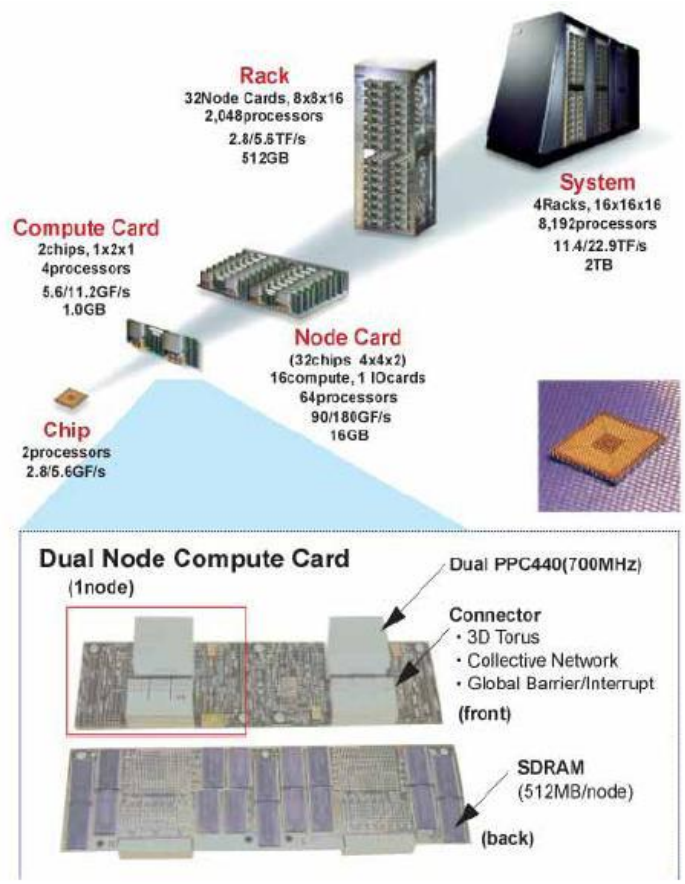


Figura 4 – Visão explodida do Blue Protein - supercomputador para simulações biofísicas [CBR09].

Além disso, eles lembram bem que “há uma grande diferença entre ‘bom o bastante’ e ‘o melhor possível’”, ou seja, uma grande precisão resulta em um melhor comportamento, que pode estar além do que é realmente necessário – por exemplo, uma razão da flutuação entre energia total e energia cinética abaixo de -0,5 é considerada suficiente e alcançada com o uso de 31 *bits* de precisão (Figura 5), o que significa que uma maior precisão é melhor, mas desnecessária. É possível observar na mesma figura que empregar 40 *bits* traz um resultado equivalente ao uso de um *datapath* completo de 53 *bits* para ponto flutuante, enquanto apenas 35 *bits* já podem ser suficientes. Segundo Gu e Herbordt em [GU07b], esses dados apenas corroboram a visão do artigo de Amisaki *et al* (*Error evaluation in the design of a special-purpose processor that calculates nonbonded forces in molecular dynamics simulation*) que, já em 1995, afirmavam que uma precisão de 35 *bits* é suficiente.

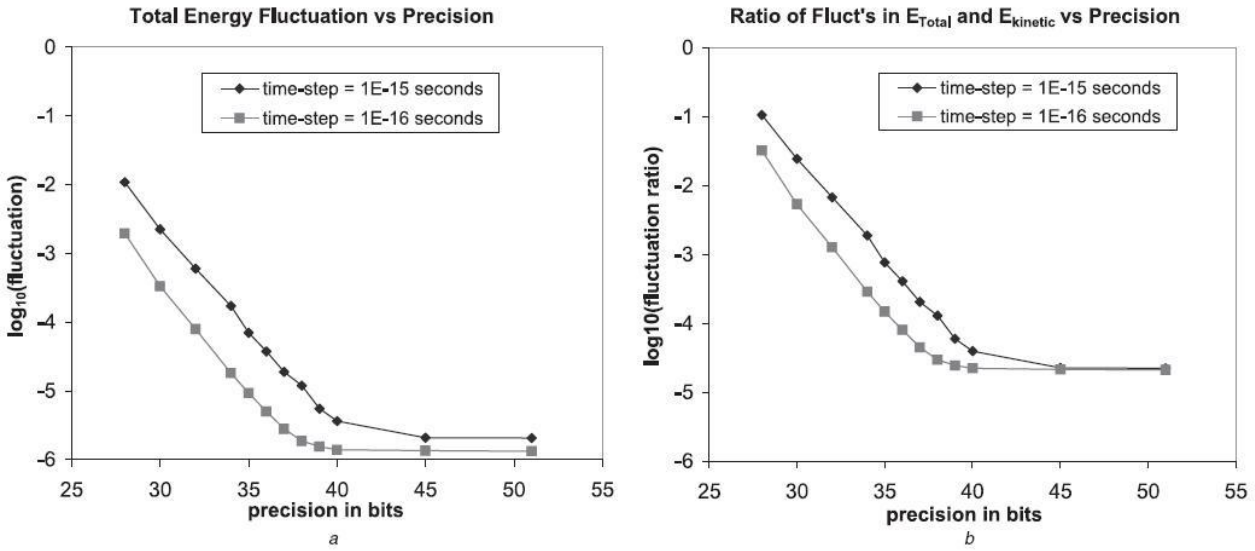


Figura 5 – Efeitos da precisão em duas métricas
para exatidão da simulação. a) Flutuação de energia total. b) Razão
da flutuação de energia total e cinética: -0,5 podem ser suficientes [GU06].

Essa é uma dificuldade que faz parte das pesquisas atuais, uma vez que apenas recentemente os FPGAs receberam recursos suficientes para abranger a aritmética de ponto flutuante. Um exemplo é o artigo de Dinechin, Detrey e Cret [DIN08], que levantam a questão sobre se é melhor ter unidades de ponto flutuante embutidas nos FPGAs (o que sacrificaria a flexibilidade, já que seria um *hardware* imutável) ou ter apenas um melhor suporte para esse tipo de cálculo. Existem também iniciativas como o projeto FPU e FPU Double, unidades de ponto flutuante de 32 e 64 *bits*, respectivamente, distribuídas livremente por Al-Eryani [ALE08] e Lundgren [LUN09] em linguagem VHDL.

Yang, Mou e Dou [YAN07] ainda comentam que é preciso cuidado com a latência criada por *pipelines*, que pode induzir problemas como *read-after-write*. Lembram também que o aumento de desempenho não é o único ponto a ser avaliado, mas o uso de recursos, de área e a precisão da simulação devem ser igualmente considerados. A Tabela 1 mostra bem esse problema: o número maior de *pipelines* resultou em um uso excessivo dos recursos do FPGA, o que levou à diminuição do *clock* e consequente aumento do atraso, sem um ganho significativo em desempenho [HAS07].

A escolha correta dos recursos disponíveis também é importante, especialmente quando tais recursos fazem parte do caminho crítico. Por exemplo, Gu, Vancourt e Herbordt em [GU06] conseguiram uma otimização com a troca de três multiplicadores de 25ns de latência por nove de 9ns de latência. Inclusive, os próprios recursos podem ser críticos, considerando-se a quantidade de multiplicadores, registradores e BRAM existentes na plataforma. Até mesmo as ferramentas de síntese alteram o desempenho final, de-

vido a diferenças no balanceamento entre a quantidade de *slices* e o desempenho resultante.

Tabela 1 – Resultados e várias implementações de DM
(adaptado de Gu, Vancourt e Herbordt [GU06]).

Precisão (*bits*)	*Pipelines*	Multiplicadores utilizados	BRAMs utilizadas	Atraso (ns)	Ganho
35	4	176 (53 %)	214 (65 %)	11,1	50,8 x
40	4	264 (80 %)	251 (77 %)	12,2	46,4 x
45	4	288 (88 %)	285 (87 %)	13,2	42,7 x
51	4	288 (88 %)	317 (97 %)	18,0	31,3 x
35	8	256 (78 %)	326 (99 %)	22,2	51,0 x

Gu, Vancourt e Herbordt levantam algumas questões interessantes, algumas já apontadas anteriormente, que tem dirigido a maioria dos estudos existentes:

- Precisão – qual é a precisão necessária e qual a motivação da escolha;
- Modo aritmético – ponto flutuante, logarítmico, híbrido, etc;
- Código de DM – qual o sistema de DM será modelado (AMBER, NAMD, experimental, etc.);
- *Hardware* alvo – qual FPGA disponível e como ele é interligado com o *host*;
- Escopo – qual a variação da implementação de DM, se o método é mais apurado e computacionalmente complexo ou se será empregado um simples raio de corte, e se o manterá em *software* ou no FPGA;
- Design – configuração do FPGA.

Uma vez tomados os cuidados necessários, é possível partir para as decisões de implementação. A idéia ao usar um FPGA é aproveitar sua alta capacidade de paralelismo. O paralelismo pode ser dividido em “grão pequeno” (operações independentes envolvidas em produzir o mesmo resultado) e “grão grande” (unidades que operam independentes para produzir resultados também independentes) [SCR06]. Outra possibilidade é o emprego de *pipelines*, criando, por exemplo, vetores para cálculo das forças em cada partícula e para atualização dos movimentos (Verlet), como feito por Gu, Vancourt e Herbordt em [GU06]. Inclusive, se os *pipelines* forem bem planejados, pode haver um compartilhamento de recursos entre eles.

Especificamente para DM, o artigo de Gu e Herbordt [GU07a] traz diversos métodos para aumento de desempenho em FPGA. Em seu trabalho estão contempladas as forças

de Coulomb e LJ e múltiplos tipos de átomos – ou seja, alguns átomos, sem apontar quais ou quantos. Para cálculo de forças de curto alcance foi utilizada interpolação, comparando os métodos de Taylor, Hermite e polinômios ortogonais. Para o problema de ponto flutuante, considerando as características da DM já elencadas, eles criam uma alternativa: uma unidade de ponto semi-flutuante combinada com unidades de inteiros nos locais em que não se pode perder precisão.

Por fim, o artigo mostra que algoritmos baseados em grade são interessantes para mapear a função de Green, necessária para o cálculo das forças de Coulomb. Esse método, chamado *Multigrid*, é visto em diversos artigos como os de Gu e Herbort [GU07a], Azizi *et al.* [AZI04] e Cho, Bourgeois e Fernández-Zepeda [CHO08]. O *multigrid* se mostra excelente para FPGAs, pois tira vantagem do alto número de memórias RAM endereçadas independentemente e da eficiência para implementação de estruturas sistólicas complexas, e para dinâmica molecular, pois é rápido e preciso para cálculos eletrostáticos. Outro benefício é dispensar o uso de FFT 3D. O modo de operação do *multigrid* se dá em três passos principais: aplicar as cargas em grades 3D, efetuar convoluções nas grades e aplicá-las novamente nas partículas. O paralelismo resultante acelera a computação e diminui a quantidade de *pipelines*.

Alam *et al.* [ALA07] têm uma abordagem diferente. Em seu artigo eles entendem que o uso de HDL é impraticável, considerando a complexidade da aplicação a ser acelerada. Os algoritmos revisados em seu artigo são os mesmos vistos até agora (método de Ewald para o problema da Lei de Coulomb), com a diferença sendo apenas a aplicação – o programa utilizado foi o AMBER [CAS05], enquanto Gu e Herbordt usaram NAMD e GROMACS. Na pesquisa de Alam *et al.*, a programação do FPGA foi feita com Fortran (a mesma linguagem usada no AMBER), não demonstrando queixas quanto à necessidade de usar FFT. Uma boa técnica apresentada foi o desenrolamento e achatamento de laços e sua paralelização (também comentado em [GU06]).

Na mesma linha de pensamento, Kindratenko e Pointer [KIN06] tentaram se evadir do HDL. É interessante notar que, em ambos os casos, a ferramenta escolhida foi o SRC Carte, um ambiente de desenvolvimento específico para o processador SRC MAPstation, altamente integrado com esse *hardware* e seu FPGA. Tal ambiente disponibiliza uma versão própria de C e Fortan. Quanto ao problema da aritmética, nesse artigo os autores dividem os dados de 64 *bits* em duas palavras de 32 *bits*, que é mais apropriado para o FPGA. Nos sistemas simulados, o desempenho foi melhorado em até três vezes. Os autores mencionam que esse resultado depende muito da simulação, pois para um raio de

corte maior o ganho pode ser de até 260 vezes.

Assim como Alam *et al.*, Scrofano e Prasanna [SCR06] também utilizaram o AMBER como base para aceleração. Entretanto, a técnica estudada foi a malha de partículas suave de Ewald (SPME), que facilita o uso de FFT 3D – para o qual foram empregadas bibliotecas prontas da Intel (da *Intel Math Kernel Library* ®). Do mesmo modo que os outros autores, eles reclamam da falta de ponto flutuante nos FPGAs e que é necessário um correto particionamento para saber o que acelerar em *hardware* e o que manter em *software* – sem perder o foco, que deve ser uma melhora no sistema como um todo. Em seu texto, criticam a trabalho de Gu e Herbordt por estar limitado a simulações pequenas que caibam na memória do FPGA (os artigos mais atuais de Gu e Herbordt afirmam que simulações maiores são possíveis, usando acesso à memória externa) e por utilizar técnicas $O(n^2)$ que não escalam bem para encontrar pares de átomos interativos. Ainda, os ganhos reportados são comparados a programas de DM lentos.

Bowers *et al.* [BOW06] e Scrofano *et al.* [SCR08] aumentam a visão das investigações ao ampliar o horizonte de computadores únicos acelerados com FPGA, ao criar agregados de nós acelerados, que é a junção de diversos computadores providos de aceleradores em uma rede. Bowers *et al.* apresentam um extenso trabalho, com muitos dados sobre paralelização, comunicação, técnicas e testes. Comparando os resultados de *clusters* acelerados e normais, Scrofano *et al.* chegaram à Tabela 2.

Tabela 2 – Aumento de performance com *cluster* de nós acelerados, em [SCR08].

Nº de nós	Latência (segundos / passo) – *software*	Latência (segundos / passo) – acelerado	Melhora
1	0,73	0,35	2,08 ×
2	0,44	0,24	1,81 ×
4	0,27	0,17	1,60 ×
8	0,17	0,11	1,51 ×

Apesar da busca por resultados que melhorem o desempenho dos sistemas discutidos aqui, tal progresso nem sempre é alcançado em um primeiro momento. No trabalho de Azizi *et al.* [AZI04], sua solução causou uma perda de desempenho: o tempo de execução de certa simulação em *software* foi de 10 seg., enquanto seu *hardware* precisou de 37 seg. – um “ganho” de 0.27 vezes. Entretanto, Azizi *et al.* encontram na memória e na velocidade do FPGA os gargalos de sua implementação. A partir dessa informação eles propõem uma melhoria no sistema memória, sem o emprego de SRAM e paralelizando seu acesso. Além disso, projetam resultados baseados em uma freqüência de trabalho

maior no FPGA (100 MHz). Ambas as alterações podem significar um ganho de 21 vezes no tempo de simulação, na mesma comparação feita anteriormente.

O trabalho de Sartin [SAR09] tem objetivos paralelos aos desta Dissertação. Seu enfoque é o *software*, caracterizando uma API de comunicação do ambiente AMBER com a plataforma de *hardware*. Estão presentes diversas comparações, como diferenças entre diversas plataformas de DM (como AMBER, GROMACS, PROTOMOL, etc.), diferenças de desempenho dos programas SANDER e PMEMD, modos de uso do PMEMD (mono e multiprocessado), traçado de perfil da ferramenta e necessidades de comunicação. Sua pesquisa foi muito importante para o desenvolvimento desta Dissertação – muitos dos dados apresentados e das escolhas tomadas aqui foram baseados no trabalho conjunto com Sartin, como se pode observar ao longo do texto. A integração de sua API com a arquitetura de *hardware* desenvolvida neste trabalho é desejada para pesquisas futuras, conforme a Seção 8.1.

6 MATERIAIS

Este Capítulo apresenta os recursos empregados para a execução das atividades propostas para atingir os objetivos específicos desta Dissertação.

6.1 Alvo: Pacote AMBER e PMEMD

Como pôde ser observado na revisão realizada neste volume, existem pesquisadores trabalhando com o ambiente AMBER, que é uma coleção de programas utilizados para efetuar e analisar simulações por dinâmica molecular [CAS05]. Dentre tais programas, especificamente para a simulação existem os programas SANDER e PMEMD. O foco desta Dissertação é o último, por se tratar de uma versão retrabalhada do SANDER e que, apesar de dar suporte apenas às principais opções do primeiro, possui desempenho superior. A simplicidade do código devido ao menor número de opções e o melhor desempenho fazem parte da motivação para uso do PMEMD.

Como o nome sugere, o PMEMD é responsável pelo cálculo do *Particle Mesh Ewald* na simulação. Para a descrição de seu funcionamento, a seguir, cita-se principalmente Crowley *et al.* [CRO97]. O método de Ewald expande as somas simples da Lei de Coulomb em três partes principais, conforme a Equação (8). O somatório direto possui um raio de corte e um fator ERFC – que força os valores de seus termos, em um valor finito de distância entre os átomos, a serem extremamente pequenos. A soma recíproca é a maior parte da energia eletrostática perdida pelo fator ERFC, tendo a forma de uma transformada discreta de Fourier. A soma de correção remove potenciais contados erroneamente na tarefa anterior.

$$E_{el} = E_{direta} + E_{reciproca} + E_{correção} \qquad (8)$$

As simulações executadas pelo AMBER apontam que o PMEMD chega a consumir mais de 80% do tempo de processamento em simulação, como se esperava – resultado compatível com um alvo de otimização, segundo a lei de Amdahl e com os trabalhos relacionados no Capítulo 5. O esforço de computação, detalhado na Tabela 3 como resultado de uso da ferramenta de traçado de perfil gprof, está concentrado no módulo pme_direct_mod (a linha *Direct Ewald time* na Tabela 3), mais especificamente nos dois laços internos à função short_ene_vec (a linha *Short ene time*, na mesma tabela) responsáveis pela lei de Coulomb na soma direta de (8) – laços “Laço 1” e “Laço 2” da Figura 6.

Tabela 3 – Processamento consumido em uma simulação
de aproximadamente 35.000 átomos no PMEMD, calculada a partir de 9900ps.
Dados obtidos com a ferramenta de traçado de perfil gprof.

```
           Build the list           421.95 (99.48% of List)
           Other                      2.19 ( 0.52% of List)
        List time                 424.14 (12.64% of Nonbo)
           Short ene time          2420.19 (99.01% of Direc)
           Other                     24.24 ( 0.99% of Direc)
        Direct Ewald time        2444.42 (83.42% of Ewald)
        Adjust Ewald time          27.54 ( 0.94% of Ewald)
           Fill Bspline coeffs        23.19 ( 5.14% of Recip)
           Fill charge grid           69.63 (15.42% of Recip)
           Scalar sum                 87.80 (19.44% of Recip)
           Grad sum                  111.59 (24.71% of Recip)
           FFT time                  159.40 (35.29% of Recip)
        Recip Ewald time          451.63 (15.41% of Ewald)
        Force Adjust                4.49 ( 0.15% of Ewald)
        Virial junk                 2.04 ( 0.07% of Ewald)
      Ewald time               2930.17 (87.36% of Nonbo)
     Nonbond force            3354.32 (98.98% of Force)
     Bond/Angle/Dihedral        33.59 ( 0.99% of Force)
     Other                       1.10 ( 0.03% of Force)
   Force time               3389.01 (98.76% of Runmd)
   Shake time                 23.98 ( 0.70% of Runmd)
   Verlet update time         14.77 ( 0.43% of Runmd)
   Ekcmr time                  2.33 ( 0.07% of Runmd)
   Other                       1.36 ( 0.04% of Runmd)
 Runmd Time                3431.44 (100.0% of Total)
 Other                        0.54 ( 0.02% of Total)
Total time                3432.14 (100.0% of ALL)
```

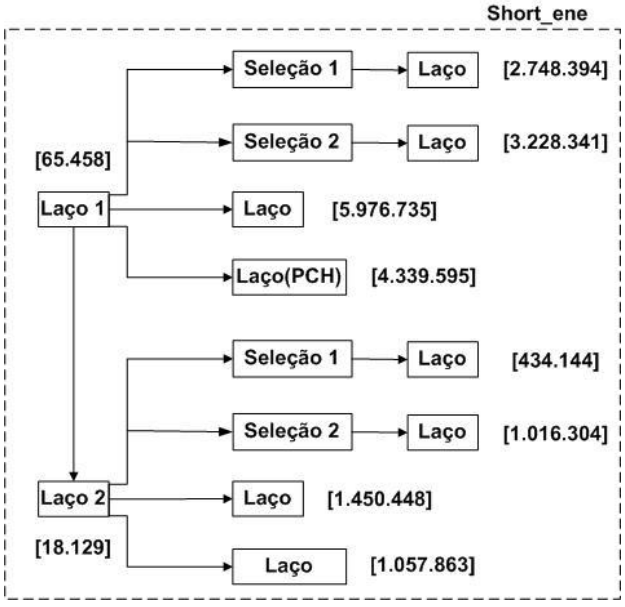


Figura 6 – Descrição da função short_ene_vec
com todos os seus laços e respectivas quantidades de repetições
(mostradas entre colchetes) na simulação de 35.000 átomos [SAR09].

A investigação do perfil da ferramenta PMEMD, realizada por Sartin [SAR09], aponta o Laço 1 como o de maior quantidade de repetições. O que não fica claro na Figura 6 é que a estrutura de ambos os laços é praticamente idêntica, existindo poucas diferenças no código. Assim, seria possível duplicar o *hardware* existente para trabalhar em paralelo

(com as devidas modificações e considerando que haja espaço suficiente no FPGA) ou alterar o controlador e os módulos para que se possam executar ambos os laços com o mesmo *hardware* – sendo que, na prática, esta é a única escolha possível uma vez que os laços estão dispostos de modo sequencial. Considerando tais ressalvas, para simplificar o desenvolvimento, apenas o primeiro laço será abordado ao longo deste trabalho.

Observando-se a Figura 6 pode parecer que o Laço do Laço 1 seria o candidato natural para ser acelerado, dada a quantidade elevada de repetições necessárias para sua execução. Entretanto, seu código é extremamente simples e não é adequado implementá-lo isoladamente no FPGA. Já o código do Laço PCH é maior e mais complexo – seu grande número de operações de ponto-flutuante propicia um melhor desempenho se executado em uma plataforma de *hardware* específica. Portanto, esse é o foco desta Dissertação.

O Laço PCH pode ser desmembrado em quatro partes principais. Cada um desses elementos foi transcrito com sucesso em *hardware*, como será visto no Capítulo 7. As partes do Laço PCH são:

- Inicialização e teste de continuidade no laço;
- Cálculo da soma direta de Coulomb utilizando EFS;
- O mesmo cálculo, com SPLINE;
- Atualização das variáveis.

Além dos apontamentos sobre o PMEMD, Sartin [SAR09] desenvolveu uma API para integração de tal *software* com um *hardware* de aceleração, mas aquela não foi aplicada nesta Dissertação por ter sido avaliada e finalmente revelar-se ineficiente. Como nem todas as tabelas são alteradas em cada iteração, atualizar seus valores no FPGA como proposto consome tempo de processamento desnecessário. Não há teste para verificar se houveram modificações nas variáveis – e mesmo tal avaliação reduz o desempenho. Cada escrita ou leitura inicia e descarrega o *driver* de comunicação, ações que poderiam ser executadas apenas no momento de começar e encerrar o PMEMD – tempo dispensável que acaba por se somar à transferência de dados, através da API.

Ainda, a API não foi preparada para trabalhar com múltiplos escravos. Seu desenho teve como alvo original especificamente um projeto denominado X10GIGA[2], que possui diversas particularidades como o envio de variáveis de ponto flutuante de precisão dupla,

[2] O Grupo de Apoio ao Hardware é um grupo de pesquisa e desenvolvimento da Universidade PUCRS, responsável pelo projeto X10GIGA. Maiores informações sobre o grupo e seus projetos estão em http://www.inf.pucrs.br/~gaph/.

divididas em duas palavras de 32 *bits* o que, portanto, exige duas escritas no FPGA. Nota-se a necessidade de adaptações melhores para lidar com o problema desta Dissertação.

Do trabalho desenvolvido por Sartin em [SAR09], além dos levantamentos de *profiling*, foi parcialmente utilizada a integração do PMEMD com os *drivers* da placa de prototipação DINI. Diversas chamadas ao *driver* foram reescritas, de modo a simplificar o processamento. Acrescentou-se também um sinal de *reset*, inexistente na API até então.

6.2 Plataformas de *hardware*

Uma vez feito o levantamento de dados, é possível responder às indagações de Gu, apresentadas anteriormente:

- Precisão – para esse projeto foi utilizada uma precisão de 32 *bits*, por se estar em um intervalo considerado suficiente [GU07b], além de corresponder ao tamanho do barramento para comunicação entre o hospedeiro e o FPGA , o que aumenta a eficiência de comunicação entre *hardware* e *software*;
- Modo aritmético – ponto flutuante, IEEE-754;
- Código de DM – AMBER/PMEMD, utilizado no Laboratório de Bioinformática desta universidade;
- *Hardware* alvo – está em uso a placa DN8000K10PCI da Dini, instalada em um microcomputador com processador Intel Pentium 4 de 2.40 GHz e 1 GB de memória RAM;
- Escopo – foi recriado o código do Laço PCH, conforme indicado por Sartin [SAR09], no FPGA alvo;
- Projeto – em hardware utilizou-se VHDL e Verilog, pela facilidade de integração com as ferramentas empregadas (Modelsim [MEN07], Xilinx ISE [XIL08] e Core Generator) e pela otimização do *hardware* final – estágio ainda não alcançado por compiladores mais genéricos, como SystemC.

A plataforma de prototipação da DINI é composta por um FPGA Xilinx Virtex4 LX100, interligada ao PC hospedeiro via barramento PCI. Para facilitar a transferência de dados entre o hospedeiro e o FPGA, a DINI criou uma estrutura de comunicação denominada MainBus [TDG05]. Desta fazem parte o programa AETest, escrito em C++ e que traz funções para a escrita no barramento, e a descrição de módulos de *hardware*, responsáveis pela comunicação com a placa. O barramento possui 32 *bits* de dados e 4 *bits*

de controle, totalizando 36 *bits*. Os módulos com a descrição de *hardware* são disponibilizados pelo fabricante na linguagem Verilog.

Visando acelerar o projeto e a implementação de sistemas é necessário capitalizar na construção modular de componentes complexos reutilizáveis, tanto de *hardware* como de *software*. No caso do *hardware*, tais componentes são denominados núcleos de propriedade intelectual (mais simplesmente, núcleos, ou, em inglês, *intellectual property cores*, *IP cores* ou apenas *cores*). Em síntese, um *IP core* é um módulo complexo de *hardware* (correspondendo a dezenas de milhares ou centenas de milhares de portas lógicas equivalentes), que desempenha alguma tarefa específica e que é criado visando o reuso em múltiplos projetos.

Nesse intuito, assim como Scrofano e Prasanna [SCR06], este trabalho também emprega bibliotecas de ponto flutuante e memórias prontas, em detrimento a criação de um *core* inteiramente novo. A diferença está na escolha da ferramenta do próprio fabricante (o Xilinx CORE Generator ®), que origina um *hardware* otimizado para o FPGA específico – a exemplo do SRC Carte [ALA07] [KIN06]. Essa é uma das motivações, conforme apresentado nas Seções 6.3 e 6.4.

6.3 Memória

Nesse quesito, as alternativas são o uso de memória interna ao FPGA (sobretudo os blocos dedicados BRAMs) ou o uso de memória externa na própria plataforma do FPGA (memórias RAM de alto desempenho), com troca de dados via acesso direto à memória (DMA) ou através do processador. A escolha mais natural para o acesso aos dados seria o DMA, pois não utiliza o processador principal do PC – não onerando o desempenho da aplicação com o *overhead* de comunicação e liberando a CPU do que seria uma sobrecarga. Entretanto, a falta de documentação acessível sobre o uso deste protocolo na plataforma adotada, somada ao nível de complexidade de seu uso, levou ao descarte desta opção.

O uso de memória RAM externa também apresenta empecilhos. A quantidade de pinos e sinais de controle, a dificuldade para implementação de uma interface de comunicação, a necessidade de mapeamento de dados nas memórias disponíveis (a organização das memórias da plataforma não possui uma estrutura linear similar a uma tabela. De fato, a posição de qualquer elemento é dada por um endereço dividido em banco, coluna e linha) e a latência maior do que a encontrada nas BRAMs (o que causa perda de desempenho) [COS07] desestimulam seu uso. Assim, elegeu-se o uso de BRAMs neste pro-

jeto, apesar de, neste caso, o trabalho deparar-se com a limitação de tamanho físico destas no FPGA utilizado, o que restringe a quantidade de dados manipuláveis e, consequentemente, o tamanho da simulação. A memória externa, ao contrário, possui tamanho que permite o uso de todo o espaço de endereçamento do protocolo MainBus.

6.4 Unidades de Ponto Flutuante

Para as unidades de ponto flutuante, pesquisaram-se duas opções. A primeira foi o módulo FPU de Al-Eryani [ALE08]. Disponível publicamente na Internet e compatível com o padrão IEEE-754, alcança uma frequência de 100MHz, opera apenas com ponto flutuante de precisão simples (32 *bits*) e possui as principais operações no mesmo pacote. Essa última característica não é interessante para o projeto desenvolvido aqui, pois todas as operações executadas são imutáveis, e requer alterações no código para que cada cálculo esteja disponível separadamente, sob pena de tomar um considerável espaço no FPGA com as áreas não utilizadas da FPU. Durante o desenvolvimento desta Dissertação, Lundgren [LUN09] disponibilizou seu projeto “FPU Double VHDL” nos mesmos moldes de Al-Eryani, mas provendo precisão dupla – 64 *bits*. Entretanto, este sequer foi testado, devido ao estágio avançado da Dissertação no momento em que se encontrou o projeto e considerando as vantagens expostas na alternativa apontada a seguir.

A opção de maiores benefícios revelou-se ser o emprego da ferramenta do próprio fabricante, o Xilinx CORE Generator. Essa aplicação gera automaticamente um *hardware* otimizado para o FPGA específico, a exemplo da abordagem empregada no SRC Carte [ALA07] [KIN06]. A vantagem é visível em espaço ocupado e em frequência máxima alcançada pelos módulos da Xilinx. Outro ponto relevante é que os módulos gerados já possuem *pipelines* – cada *core* permite o envio de um novo dado a cada ciclo de relógio. Por fim, a migração para ponto flutuante de dupla precisão – que necessita de 64 *bits* – é facilitada, pois o CORE Generator também cria com este grau de precisão. Tantas conveniências se tornaram razões para preferir esta opção em relação às FPUs de Al-Eryani e de Lundgren.

7 ARQUITETURA DE HARDWARE DESENVOLVIDA

A arquitetura, apresentada de modo geral na Figura 7, foi construída de maneira modular, dividindo o Laço PCH conforme o levantamento efetuado na Seção 6.1. Os módulos correspondentes a execução proposta naquela seção são a máquina de estados, o Coulomb-EFS, o Coulomb-SPLINE e o de atualização, descritos nas próximas seções – à exceção do primeiro.

A máquina de estados é do tipo Moore, com 27 estados codificados no estilo "*one-hot*" (cada estado possui um *flip-flop* próprio, representando o estado atual) [CUM02], sendo cinco desenhados para depuração dos módulos Coulomb-EFS e Coulomb-SPLINE. É responsável por comandar o início e fim da execução de cada *hardware* e o término do laço, passando os valores das variáveis entre diversos módulos e ao escravo de controle. É através do escravo de controle que se efetiva a comunicação entre *software* alvo e *hardware* – as sinalizações para começar o processamento e verificação de sua conclusão, assim como a inicialização e recuperação das variáveis principais. Detalhamentos estão presentes na Seção 7.3.

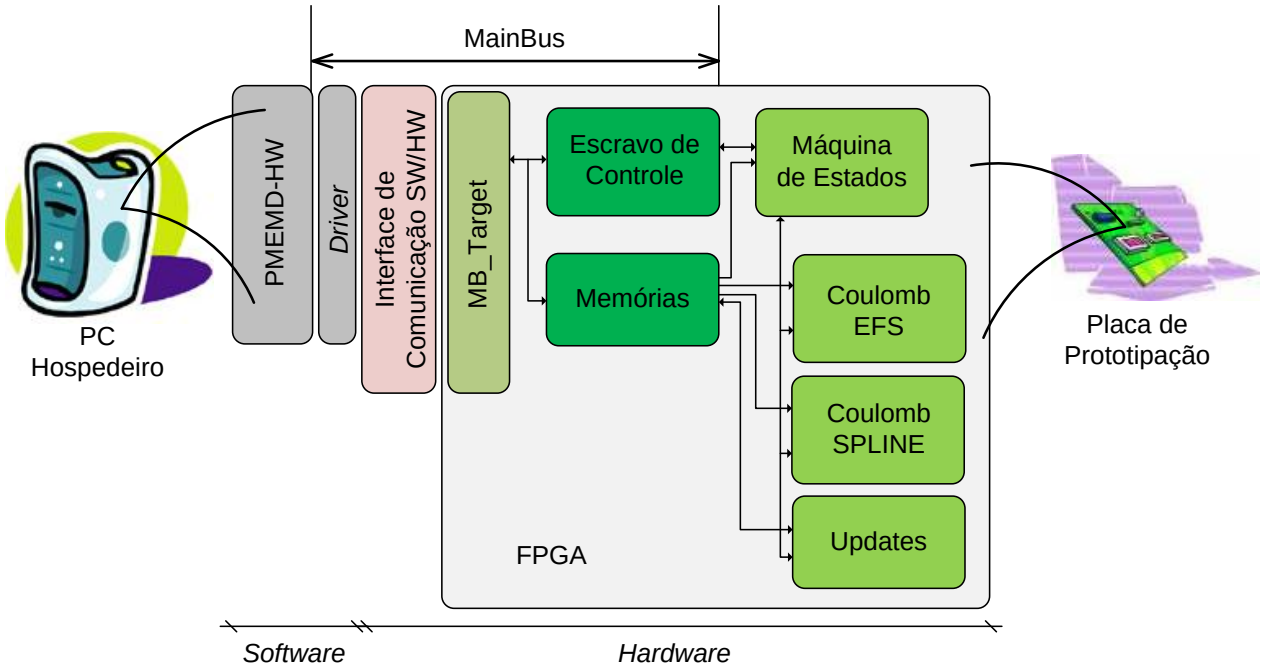


Figura 7 – Representação da arquitetura desenvolvida.

Durante a construção da arquitetura foram executados dois processos para a descrição do *hardware*: escrita diretamente em VHDL [MOR98] [PER05] [ASH90] ou Verilog [CUM00] e desenho a partir de esquemático. O VHDL foi empregado inicialmente, com o qual se compôs o módulo Coulomb-EFS. Por ser uma linguagem de descrição, parece-se com programação de *software*, o que facilita a migração dos profissionais entre as áreas.

Uma vez conhecida a complexidade de um, elegeu-se o esquemático para a criação do outro, o Coulomb-SPLINE. O desenho é um método mais rápido quando o módulo a ser desenvolvido é pequeno, com poucas ligações. A partir de um determinado tamanho, a quantidade de fios a gerenciar (de modo a manter o desenho com fácil visualização e entendimento) e a falta de espaço disponível na folha de desenho (limitação da ferramenta – por assumir que o esquemático será impresso, aceita no máximo o tamanho das maiores folhas disponíveis para impressão – no caso, A0) se tornam fatores que dificultam o desenvolvimento.

Como a visualização é mais simples com esquemático e a depuração é mais rápida em VHDL/Verilog, o projeto foi continuado de modo híbrido, alternando entre ambos os métodos. Após terminar o desenho principal do módulo Coulomb-SPLINE, os ajustes finos foram feitos em VHDL – a suíte da Xilinx permite criar o arquivo VHDL a partir do esquemático, e vice-versa. O mesmo processo foi aplicado para o módulo Updates.

A Figura 8 oferece o esquemático da arquitetura desenvolvida neste trabalho, com simplificações para facilitar sua visualização. Por exemplo, as variáveis de inicialização e os sinais contendo os resultados no escravo de controle foram agrupados; os escravos idênticos estão reunidos; o sinal MB_sel (em pontilhado largo), que habilita a um determinado escravo o acesso ao MainBus, é individual para cada escravo mas é identificado por apenas um fio, pois tem a mesma funcionalidade. Além disso, dada a reunião dos escravos, os pinos de endereçamento das tabelas foram omitidos e seu conteúdo foi representado por fios pontilhados, indicando que ou a memória serve a mais de um módulo e/ou o módulo recebe de mais de uma memória.

7.1 Memória

Tanto as memórias quanto as unidades de ponto flutuante foram geradas pelo Xilinx CORE Generator [XIL05], parte do pacote Xilinx ISE Design Suite 10.1 [XIL08]. É uma ferramenta que cria blocos de propriedade intelectual (IP) parametrizáveis, otimizados para FPGAs da Xilinx e, por isso, a escolha deste projeto. Após a geração do bloco, a aplicação entrega, entre outros arquivos, um modelo para simulação (o *wrapper*, em VHDL, Verilog ou esquemático) e o arquivo de implementação, o *netlist* (NGC ou EDIF).

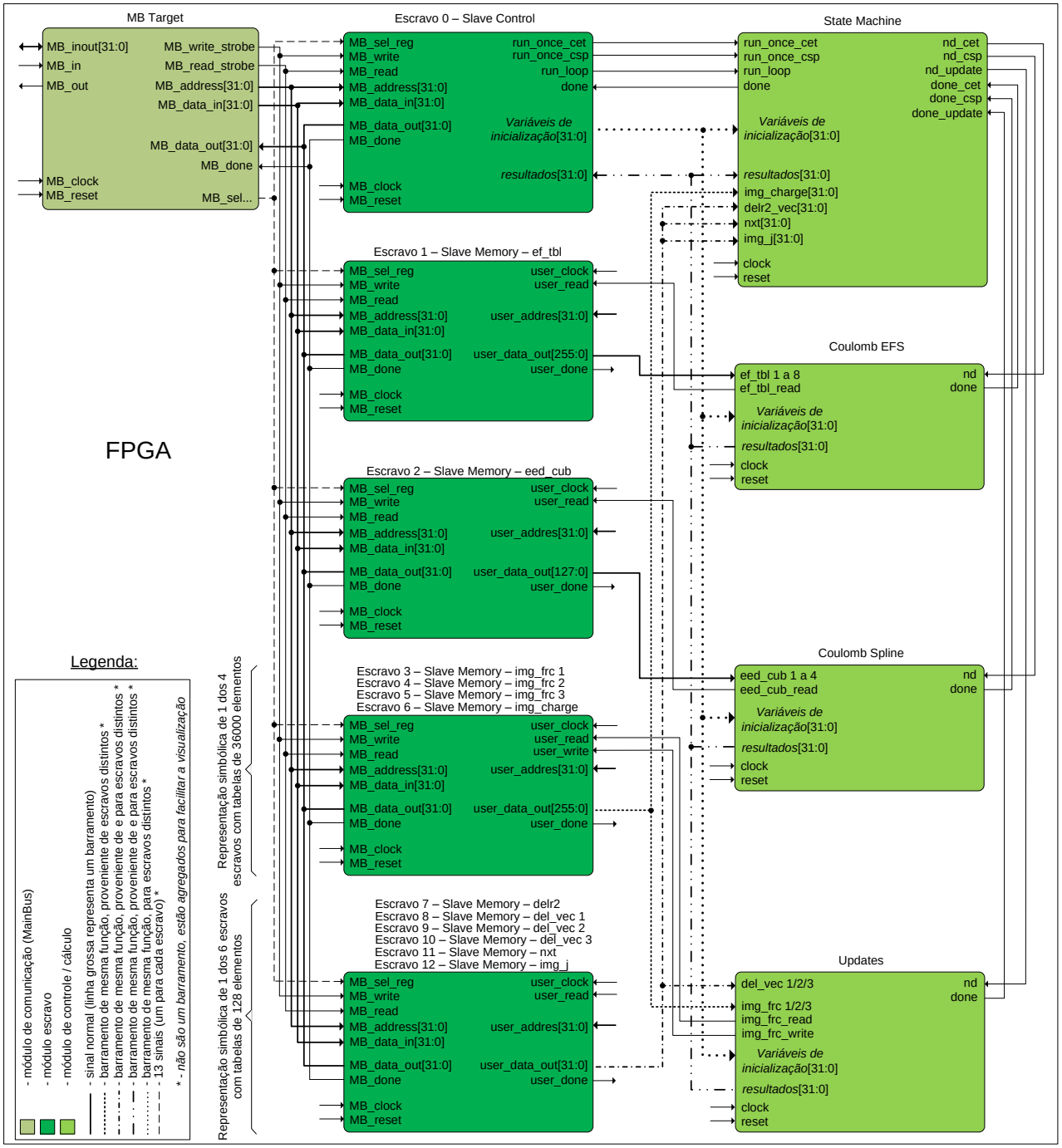


Figura 8 – Esquemático simplificado da arquitetura desenvolvida.

Todos os blocos de memória foram descritos em *hardware* para funcionar de maneira idêntica, alterando-se apenas o espaço disponível para dados e a largura equivalente das portas. Tal ação facilita o projeto da arquitetura, pois ao dominar o funcionamento de um módulo é possível replicar o conhecimento para o uso dos outros – parte do princípio do reuso de componentes. Os blocos de memória apresentam duas portas, sendo entrada e saída separadas. Todas as entradas possuem 32 *bits* de largura, condizente com o tamanho de uma variável de ponto flutuante de precisão simples. Já as saídas são de largura variável, conforme a necessidade de acesso aos dados – indo de 32 a 256 *bits* (ou oito

palavras de 32 *bits*). São duas as principais razões para tal escolha. Para os módulos de cálculo, o acesso simultâneo aos dados é importante, pois garante certo grau de paralelismo. Já para a escrita nas tabelas, o *clock* da porta de entrada precisa estar sincronizado com o MainBus, enquanto o *clock* da porta de saída deve se sincronizar com os módulos – que podem empregar uma frequência diferente.

As tabelas *eed_cub* e *ef_tbl* têm uma peculiaridade na porta de saída. Como explicado anteriormente, para que haja paralelismo na obtenção de dados, ambas as tabelas têm saídas com largura suficiente para entregar quatro e oito elementos ao mesmo tempo – 128 e 256 *bits* – respectivamente. As tabelas *del_vec* e *img_frc* poderiam apresentar o mesmo comportamento, oferecendo três elementos simultaneamente. Entretanto, há diversas memórias com o mesmo tamanho de *del_vec* e *img_frc* (128 e 12.461 elementos, respectivamente), assim estimulando o reuso. Além disso, o CORE Generator disponibiliza, na porta de saída, profundidades 1, 2, 4, 8 e 16 vezes a da entrada, impossibilitando o acesso a apenas três endereços – o que deixaria um endereço sem uso, um *overhead* desnecessário. Desse modo, empregaram-se três tabelas distintas em cada caso.

Simulações com um número maior de átomos ou um raio de corte diferente demandarão a geração de novas tabelas, para conter as novas quantidades de elementos. Lembrando que se deve alterar apenas o tamanho das tabelas, nada mais. Um ponto a ser considerado para simulações maiores é que o protocolo MainBus reserva apenas 24 *bits* para o espaço de endereçamento, alcançando então um máximo de 16.777.215 posições em cada “escravo” do protocolo [TDG05] – ou seja, pode-se empregar tabelas com tamanho de até 16 milhões de elementos, suficientes para a maioria das simulações atuais. Esse valor pode ser alterado, à custa da diminuição no número de escravos.

A Figura 9 apresenta o símbolo esquemático com os sinais de dados e *handshaking* disponíveis nos blocos de memória de FPGAs Xilinx. Os sinais são os mesmos para as duas portas, “a” e “b”, respectivamente: addr[a|b] (endereço de leitura/escrita); din[a|b] (entrada de dados/não utilizado); we[a|b] (habilitação escrita/não utilizado); en[a|b] e sinit[a|b] (não utilizados); nd[a|b] (novo dado a escrever/ler); clk[a|b] (relógio de cada porta); dout[a|b] (não utilizado/saída de dados); rfd[a|b] (indica que o módulo está pronto – na prática não é empregado); rdy[a|b] (não utilizado/dado disponível).

Cada bloco de memória foi embalado em um módulo escravo do protocolo MainBus, demonstrado no símbolo da Figura 10. Novamente, o intuito é o reuso de componentes, além de facilitar a programação do aplicativo-alvo (PMEMD), pois assim mantêm-se os índices originais, alterando-se apenas a identificação do escravo a ler ou escrever. Caso

utilizasse mais de um bloco em um mesmo escravo, seria necessário utilizar um *offset* para escolher qual a tabela destino.

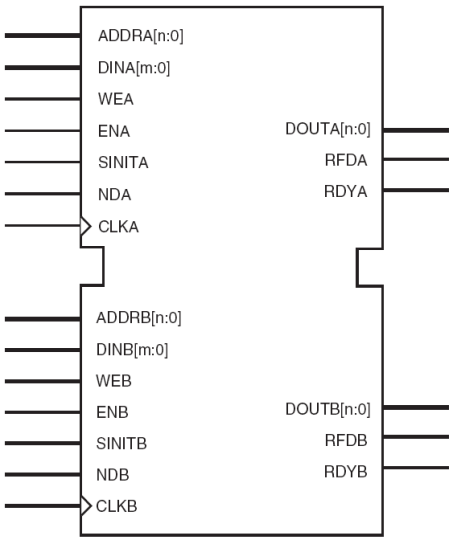


Figura 9 – Símbolo esquemático de um bloco de memória interno em FPGA da Xilinx [XIL05].

Com o uso de uma API como a de Sartin [SAR09], isso ficaria transparente para o programador do aplicativo-alvo – mas, nesse caso, a API mencionada requer adaptação, pois não prevê tal uso. Entretanto, para quem descreve o *hardware*, se faz necessária uma lógica de controle adicional, além de diminuir o espaço de endereçamento (já que esse seria dividido por *n* blocos).

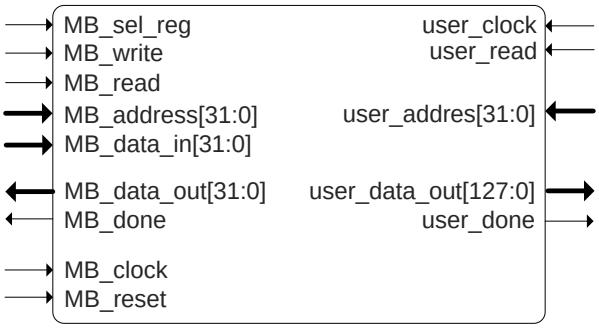


Figura 10 – Exemplo de um símbolo esquemático de escravo manipulador de memória.

7.2 Unidades de Ponto Flutuante

Conforme os apontamentos da Seção 6.4 e a exemplo dos blocos de memória, a aritmética de ponto flutuante foi desenvolvida com o auxílio da ferramenta Xilinx CORE Generator [XIL06]. Todas as operações estão em conformidade com o padrão IEEE 754, com desvios apenas em comprimentos de palavra não-padrão (*non-standard wordlength* – gerencia tamanhos além dos que estão definidos na norma), números não-normalizados (não há suporte, todos são tratados como zero), modo de arredondamento (possui apenas o “arredondar para o mais próximo” – *round-to-nearest*) e indicação de “não é um

número" sinalizado ou silencioso (*signaling NaN* e *quiet NaN* são tratados como *quiet NaN – not-a-number*).

Dada a natureza das operações de soma e subtração, o CORE Generator permite mantê-las em uma mesma estrutura, sem alteração significativa no uso de recursos – como se pode observar da Tabela 4, ambas as operações têm o mesmo consumo. Entretanto, é necessária uma lógica de controle que indique qual operação será executada naquele momento. Isso pode ser útil no reuso de recursos, mas não será empregado neste trabalho considerando que as operações são fixas e causaria um excesso de fios, que permaneceriam em desuso.

Todas as FPUs foram criadas com precisão simples (32 *bits*), exclusivamente para o FPGA Virtex4 FX100 que se encontra na plataforma. A Tabela 4 mostra o tamanho de cada *core* gerado. Todos foram parametrizados de modo a obter a maior latência (número de ciclos de relógio entre a entrada de um novo operador e a obtenção do resultado, o que garante uma maior frequência de operação do *hardware*) e o maior paralelismo disponível (pode-se aplicar um novo dado a cada *clock*, tendo, assim, maior desempenho).

Tabela 4 – Utilização de recursos por operação, com latência máxima
(para Virtex4 FX100 1152, dados do Xilinx CORE Generator). *Módulo com recurso Xilinx DSP48E.

Recurso	Adição*	Adição	Sub.*	Sub.	Mult.*	Mult.	Divisão	Raiz	Comparação	Fixo para Flut.	Flut. Para Fixo
Slice Flip-Flops	486	589	486	589	278	698	1365	819	29	234	236
Slices ocupados	347	447	348	443	216	432	880	540	59	180	177
LUT de 4 entradas (total)	354	581	354	581	163	651	820	566	97	280	229
Xilinx DSP48E	4	-	4	-	4	-	-	-	-	-	-

Entretanto, essa última característica implica em um consumo elevado de recursos e obriga a criar registradores para guardar os resultados entre operações com tempo de execução diferentes (maiores detalhes na Seção 7.7). Uma opção seria aplicar uma taxa de envio de dados menor (o que habilitaria o reuso de *hardware* na própria FPU), até obter uma implementação completamente sequencial. Nesse caso, alcança-se o paralelismo ao replicar todo o módulo de cálculo – mas, como a memória é acessada uma vez a cada *clock*, não há ganhos com essa concepção. Poder-se-ia, então, replicar também as me-

mórias, mas não há espaço físico para tal no FPGA disponível no momento de escrita desta Dissertação. Acrescente-se a isso as necessidades de realimentações – o que obrigaria aos módulos adicionais esperar o resultado uns dos outros – e se perde qualquer vantagem dessa implementação.

Uma peculiaridade comum a todas as unidades de ponto flutuante é o sinal *operation_rfd*, que indica que a FPU está pronta para receber dados (***r**eady **f**or **d**ata*). Como todos os *cores* foram preparados de modo a receber novos dados a cada ciclo do relógio, tal *flag* é assertada assim que a FPU está inicialmente pronta e depois permanece imutável, a menos que seja enviado um sinal de *reset* – configurações diferentes da escolhida podem ter outro comportamento, conforme o manual da ferramenta [XIL06]. Essa propriedade desobriga a vigiar a *flag* de cada *core*, pois se pode assumir que, quando as primeiras FPUs estão prontas, todas as posteriores também estarão. Assim, a respectiva porta permanece aberta nas unidades não vigiadas, diminuindo a lógica no FPGA. Os outros sinais, visíveis na Figura 11, são: a e b (operandos), operation (não utilizado), operation_nd (*new data* – há novos dados a serem computados), operation_rfd (explicado anteriormente), sclr (*synchronous reset* – reiniciar), ce (não utilizado), clk (relógio), result (resultado da operação), underflow, overflow, invalid_operation, divide_by_zero (sinais de erro, não utilizados), rdy (*ready* – cálculo terminado).

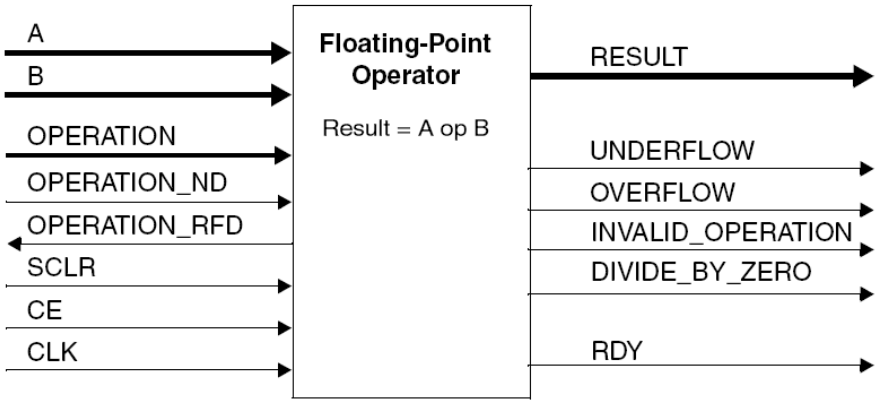


Figura 11 – Diagrama genérico de um operador binário de ponto flutuante [XIL06].

De todas as operações disponíveis, as unidades de transformação de ponto flutuante para fixo (*float-to-fixed*) foram descartadas, uma vez que o único método de arredondamento disponível (*round-to-nearest*) insere erros no momento de obter o endereço do índice nas tabelas. Tal endereço é obtido utilizando-se apenas a parte inteira de um número calculado em ponto flutuante, sendo que valores maiores do que *n,5* se transformam, assim, em *n+1*, enquanto o esperado é apenas *n*. Para corrigir esse problema, cri-

ou-se um módulo específico para truncar os operandos (Figura 12). Seu funcionamento é simples: conforme o expoente do número (segundo as normas da IEEE-754), de 0 a 23, aplica-se uma máscara que mantém apenas a parte inteira do valor, descartando a parte fracionária. Há dois benefícios nessa abordagem. A aplicação da máscara é imediata – a latência é de apenas um ciclo do relógio. Além disso, não é preciso retornar o valor de fixo para flutuante – como ambas as formas são utilizadas nos cálculos, mas sempre truncadas, obrigaria a conversão flutuante-fixo (para obter o fixo) e depois fixo-flutuante (para obter o valor truncado), operações de grande latência.

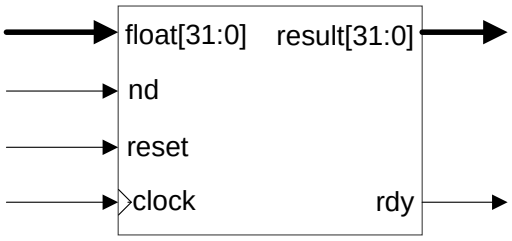


Figura 12 – Símbolo esquemático do módulo de truncamento.

Outro obstáculo é a parca quantidade de unidades Xilinx DSP48E no FPGA, impedindo sua colocação em todas as unidades aritméticas. Além de diminuir o uso de recursos, os DSP48E melhoram o desempenho. Assim, se faz necessário o levantamento do caminho crítico para substituir os *cores* que o compõem por outros com DSP48E.

7.3 Comunicação

Conforme mencionado na Seção 6.2, a DINI Group facilita a transmissão de dados entre o PC hospedeiro e a plataforma FPGA ao disponibilizar seu protocolo de comunicação denominado MainBus – um conjunto composto de *driver* (no lado do PC) e descritores de *hardware* (no lado do FPGA) [TDG05]. A interligação entre PC e FPGA pode ser feita pelos barramentos USB ou PCI, sendo que a escolha desta Dissertação recaiu no último.

Na especificação do MainBus existe um mestre e até 16 escravos. O conceito de escravo, neste caso, é um módulo no FPGA que pode se comunicar diretamente com o barramento MainBus. O barramento se compõe de 32 *bits* compartilhados para endereço e dados (bi-direcional para, utilizado tanto para a leitura quanto para gravação), mais 4 *bits* de controle, totalizando 36 *bits*. O controle é formado pelos sinais ALE, indicando o envio do endereço, o sinal RD, para realizar leitura, o sinal WR, para realizar gravação, e o sinal DONE, que indica que o escravo terminou o processamento (seja o consumo do valor a ser registrado ou a disponibilização do dado a ser lido) [TDG05].

A transferência de dados entre o MainBus e o hospedeiro é simples. Após iniciar o

driver da placa, o *software* pode requisitar uma escrita ou uma leitura no barramento. As duas operações são divididas, no *driver*, em quatro partes: habilitação do acesso à PCI, habilitação do acesso ao MainBus, envio do conjunto escravo/endereço e, finalmente, efetivação da leitura ou gravação. A forma de onda para os procedimentos de leitura e gravação no FPGA pode ser vista na Figura 13 e na Figura 14, respectivamente [TDG05]. Ao término do programa, o *driver* deve ser descarregado.

As operações de inicialização e descarga do *driver* de comunicação com a placa foram inseridas no arquivo pmemd.fpp, que é o responsável por começar e terminar a execução do PMEMD. Aproveitando a integração entre o *driver* e o PMEMD desenvolvida por Sartin em [SAR09], foram acrescentadas duas primitivas para tais operações, chamadas:

- accel_init;
- accel_deinit.

Para o envio e recebimento de dados, foram adicionadas primitivas que manipulam variáveis inteiras e de ponto flutuante. A operação de leitura recebe como parâmetro o valor do local de onde o dado deve ser lido e a variável para onde guardar o resultado. Já a de escrita é formada pelo endereço e o dado a escrever:

- accel_write_int (endereço, valor inteiro)
- accel_write_double (endereço, valor ponto flutuante)
- accel_read_int (endereço, variável inteira)
- accel_read_double (endereço, variável ponto flutuante)

Para indicar o endereço onde manipular os dados, o protocolo necessita de um valor composto pelo número que identifica o FPGA na placa (a plataforma DN8000K10PCI da DINI comporta até três FPGAs), o número do escravo (até 16) e o endereço ou índice do dado propriamente dito. O MainBus reserva 4 *bits* para identificação do FPGA, 4 para indicar o escravo e os outros 24 para decodificação dentro do escravo. Tais valores podem ser alterados para, por exemplo, aumentar a quantidade de escravos – consequentemente diminuindo o espaço de endereçamento interno aos escravos [TDG05] [TDG09].

A arquitetura desenvolvida neste trabalho ocupa 12 dos 16 escravos disponíveis. O primeiro escravo, o escravo de controle – “slave_control”, é responsável pela transferência de diversas variáveis, além de comandar o início do processamento e verificar seu término. As variáveis que ele conduz de e para o FPGA estão na Tabela **7**, linhas da coluna “Escravo” com valor “0”. Para controle da carga de dados na memória, cada tabela a ser manipulada conecta-se a um dos escravos restantes, conforme a Seção 7.1.

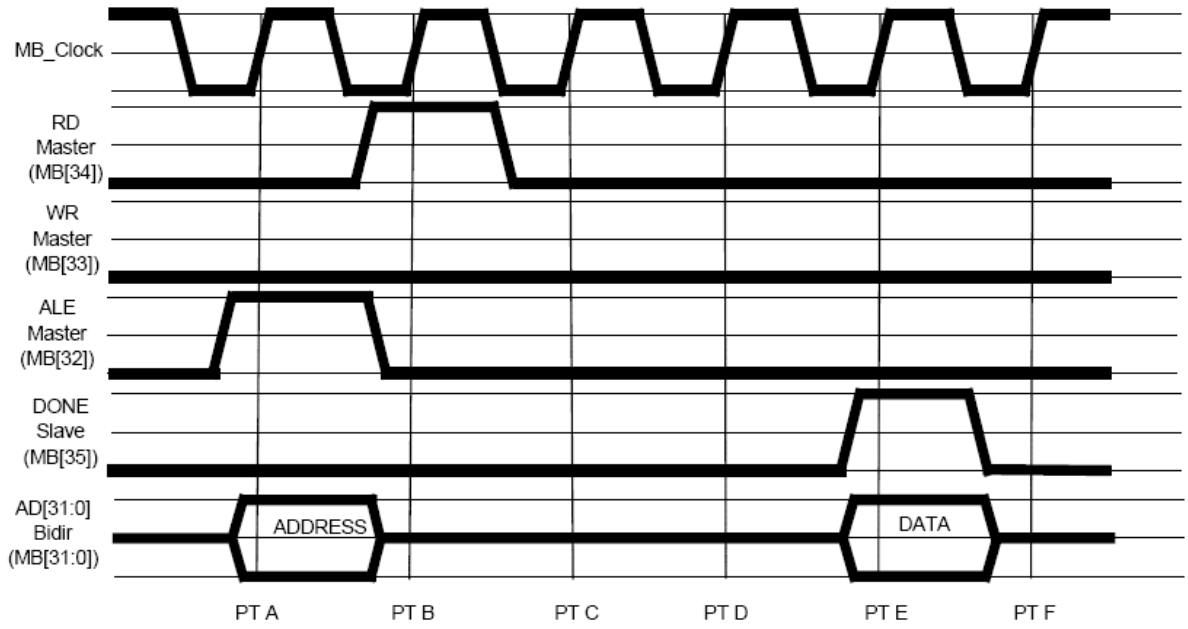


Figura 13 – Operação de leitura no protocolo MainBus.
O sinal ALE indica o envio do endereço. O sinal DONE indica que o dado está disponível [TDG05].

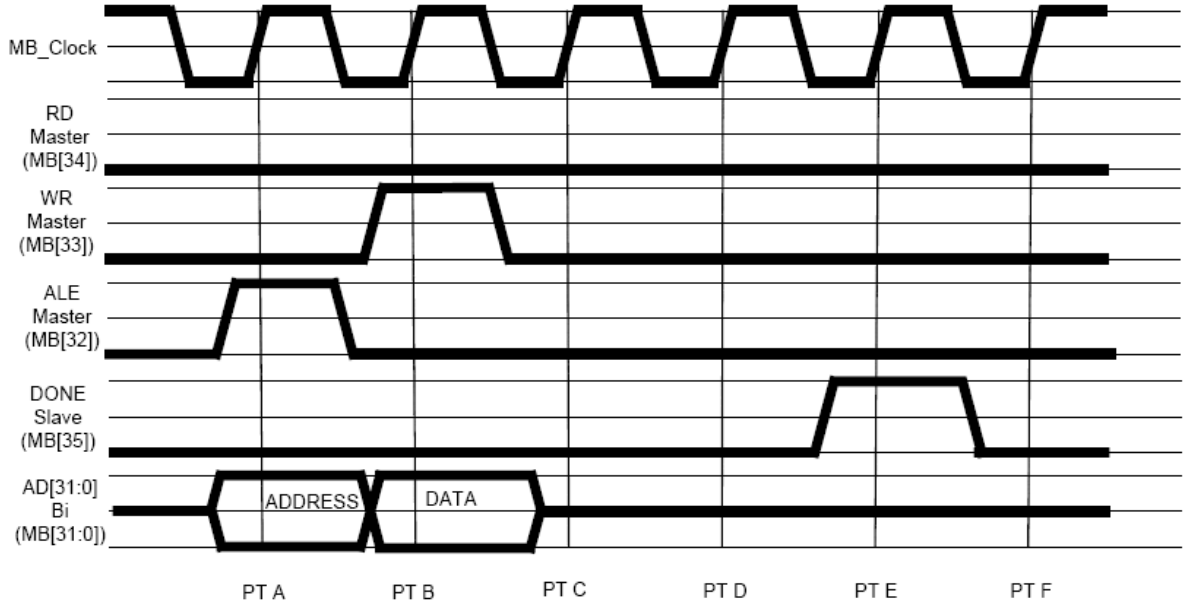


Figura 14 – Operação de escrita no protocolo MainBus.
O sinal ALE indica o envio do endereço.O sinal DONE indica que o dado foi consumido [TDG05].

O escravo de controle foi construído de modo a permitir a execução singular do módulo Coulomb-EFS e Coulomb-SPLINE (sinais run_once_cet e run_once_csp), ou realizar o laço completo do PMEMD (sinal run_loop). A motivação para executar cada módulo em separado é facilitar a depuração tanto do *hardware* como da comunicação entre o hospedeiro e o FPGA. Cada variável, assim como os sinais de controle, possuem um endereço próprio que pode ser acessado para atualizar ou ler seu conteúdo.

Para saber se o processamento foi concluído, deve-se apenas ler o conteúdo do sinal “done”. Apesar de ser uma solução simples, não é a melhor, uma vez que o tempo

para o cálculo terminar é indeterminado, forçando a verificação contínua desse sinal. Assim, é necessário manter o processador em um laço efetuando *pooling* naquele endereço, o que significa que o processador receberá uma carga desnecessária de trabalho além de não estar ciente do momento exato do término do processamento. O ideal seria que o programa permanecesse em espera até ser avisado pelo FPGA. Entretanto, o protocolo MainBus não prevê tal forma de comunicação.

O levantamento dos tempos de comunicação entre o PC e o FPGA está listado na Tabela 5 e na Tabela 6. A quantidade de testes na segunda tabela é limitada devido à falta de precisão do Fortran na função utilizada para aquisição do tempo de execução das operações. Os tempos podem variar levemente a cada teste, provavelmente dependendo da disputa pelo barramento PCI. Como se pode perceber, uma troca intensa de dados eleva consideravelmente o tempo total de execução do programa-alvo. Essa é uma área que necessita cuidados especiais, sob pena de degradar o desempenho.

Na Tabela **7** estão as variáveis transferidas entre a plataforma de *hardware* e o PMEMD. A coluna “Quantidade de variáveis / Sartin” possui a quantidade de dados que foi classificada por Sartin em [SAR09], que difere do que foi adquirido pelo autor desta Dissertação. Em parte, isso acontece porque o modelo de simulação utilizado neste trabalho é diferente, com número de partículas e raio de corte menores – imprescindível para caber no FPGA em uso.

Tabela 5 – Tempo para comunicação PC - FPGA (apenas *driver*).

Apenas driver	
Operação	Tempo (µs)
Inicialização	477345
Descarga	2973
Envio de 1 variável	1
Envio de 10 variáveis	4
Envio de 20 variáveis	12
Envio de 20000 variáveis	24180
Leitura de 1 variável	3
Leitura de 10 variáveis	25
Leitura de 20 variáveis	49
Leitura de 20000 variáveis	48396

Tabela 6 – Tempo para comunicação PC - FPGA (no PMEMD).

No PMEMD	
Operação	Tempo (s)
Inicialização	0,001
Descarga	0,002
Envio de 20000 variáveis	0,024
Leitura de 20000 variáveis	0,049

Tabela 7 – Características das variáveis transferidas entre o FPGA e o PMEMD. "–" indica dado não utilizado (adaptado de Sartin [SAR09]). Os nomes das variáveis espelham os encontrados no programa PMEMD.

Variável	Escravo	Tipo	Quantidade de variáveis	
			Sartin	Autor
dens_efs	0	Ponto flutuante	1	1
dxdr	0	Ponto flutuante	1	1
eedtbdns_stk	0	Ponto flutuante	1	1
eed_stk	0	Ponto flutuante	1	1
eedvir_stk	0	Ponto flutuante	1	1
cgi	0	Ponto flutuante	1	1
nxt_cnt	0	Inteiro	-	1
del	0	Ponto flutuante	1	-
vxx	0	Ponto flutuante	1	1
vxy	0	Ponto flutuante	1	1
vxz	0	Ponto flutuante	1	1
vyy	0	Ponto flutuante	1	1
vyz	0	Ponto flutuante	1	1
vzz	0	Ponto flutuante	1	1
dumx	0	Ponto flutuante	1	1
dumy	0	Ponto flutuante	1	1
dumz	0	Ponto flutuante	1	1
ef_tbl	1	Ponto flutuante	25.600	26.000
eed_cub	2	Ponto flutuante	26.396	83.672
img_frc	3, 4 e 5	Ponto flutuante	3 x 35.681	3 x 12.461
img.charge	6	Ponto flutuante	35.681	12.461
delr2_vec	7	Ponto flutuante	128	128
del_vec	8, 9 e 10	Ponto flutuante	3 x 128	3 x 128
img_j_vec	11	Inteiro	128	128
nxt	12	Inteiro	128	128

7.4 Módulo Coulomb – EFS

O módulo Coulomb-EFS efetua o cálculo da soma direta de Coulomb com dados provenientes da tabela EFS (que é composta de duas tabelas spline com quatro coeficientes spline cada – justificando os oito elementos que devem ser carregados da memória, na Figura 15), batizando, assim, o nome do módulo. Seu funcionamento replica o código disponível no programa PMEMD e pode ser visto na Figura 15 – os símbolos “,0” e “,-” representam o truncamento e conversão de ponto flutuante para inteiro, respectivamente. Deve-se perceber que esse não é o esquemático, mas sim uma visão simplificada do cálculo. Devem se somar à figura, ainda, diversos sinais:

- clock – entrada, relógio;
- reset – entrada, voltar aos parâmetros de inicialização;
- nd – entrada, indica que há dados novos e comanda o início da computação;
- rfd – saída, o hardware está pronto para receber dados;
- done – saída, determina o término de seu cálculo;
- ef_tbl_read – saída, ordena que a tabela ef_tbl leia a memória contida no endereço ind.

Na Tabela 10 estão dispostos os valores de todas as variáveis que compõem essa área de cálculo. Os dados demonstram que há uma perda de precisão entre o que é registrado no PMEMD e o que é calculado no módulo correspondente em *hardware*, na ordem de -0,000005978082207036 55% a 0,000000032771199753867 1%. Esse comportamento é esperado, uma vez que o *software* trabalha com precisão dupla (64 bits) e o *hardware* com precisão simples (32 bits).

Por fim, são apresentadas estatísticas fornecidas pela ferramenta Xilinx ISE/XST. Informações sobre a frequência máxima que o *hardware* pode alcançar estão em destaque na Tabela 8 (no caso, até 290 MHz) e os recursos que seriam consumidos no FPGA estão na Tabela 9.

Tabela 8 – Sumário de temporização do módulo Coulomb-EFS
para Virtex4 FX100 1152, dados do Xilinx ISE/XST.

```
Timing Summary:
---------------
   Minimum period: 3.426ns (Maximum Frequency: 291.885MHz)
   Minimum input arrival time before clock: 3.022ns
   Maximum output required time after clock: 5.003ns
   Maximum combinational path delay: No path found
```

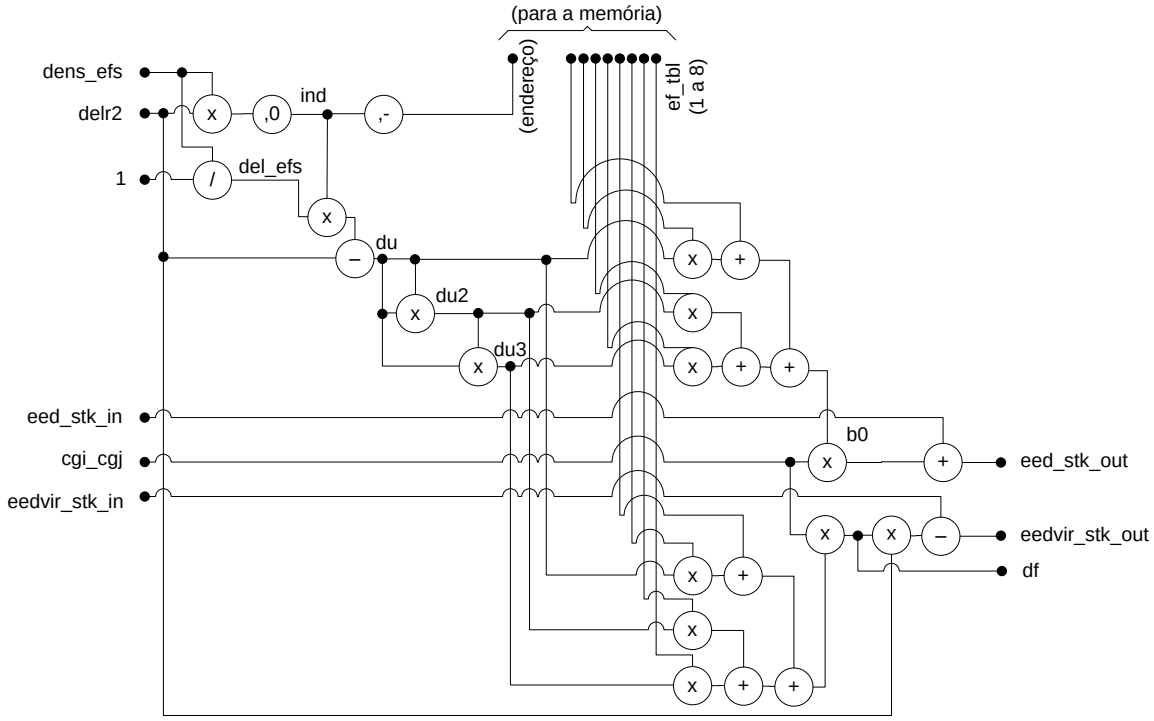


Figura 15 – Bloco para o cálculo de Coulomb com a tabela EFS.
Onde “,0” é o truncamento e “,-” é a conversão flutuante/inteiro.

Tabela 9 – Sumário de design do módulo Coulomb-EFS
para Virtex4 FX100 1152, dados do Xilinx ISE/XST.

```
Design Summary
--------------
Logic Utilization:
  Total Number Slice Registers:      16,210 out of  84,352   19%
    Number used as Flip Flops:       16,204
    Number used as Latches:               6
  Number of 4 input LUTs:            14,494 out of  84,352   17%
Logic Distribution:
  Number of occupied Slices:         10,818 out of  42,176   25%
    Number of Slices containing only related logic:  10,818 out of  10,818 100%
    Number of Slices containing unrelated logic:          0 out of  10,818   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:      15,062 out of  84,352   17%
    Number used as logic:            13,448
    Number used as a route-thru:        568
    Number used as Shift registers:   1,046
  Number of bonded IOBs:                550 out of     576   95%
    IOB Latches:                         32
  Number of BUFG/BUFGCTRLs:               2 out of      32    6%
    Number used as BUFGs:                 2
```

Tabela 10 – Comparação entre resultados do PMEMD e do módulo Coulomb-EFS.
Variáveis em itálico indicam entradas do PMEMD para o módulo.

Variável	Valor registrado no PMEMD	Valor calculado no módulo	
		Hexadec.	Ponto Flutuante
delr2	52.4095273016550	4251A35B	52.40952682495117
cgi_cgj	6.32788590028586	40CA7E0B	6.32788610458374
dens_efs	50	42480000	50
ef_tbl_i1	2.257498023728525E-004	396CB740	2.2574979811906815E-004
ef_tbl_i2	-2.537607427479283E-005	B7D4DEB4	-2.537607360864058E-005
ef_tbl_i3	1.462357689213157E-006	35C4463A	1.4623576589656295E-006
ef_tbl_i4	-5.786844007791887E-008	B3788B06	5.786844070598818E-008
ef_tbl_i5	5.075214854959984E-005	3854DEB4	5.075214721728116E-005
ef_tbl_i6	-5.849433620177149E-006	B6C44641	-5.849433819093974E-006
ef_tbl_i7	3.476396026906283E-007	34BAA339	3.4763959888550744E-007
ef_tbl_i8	-1.428821506012740E-008	B275783C	-1.4288215055557885E-008
df	3.208013560454216E-004	39A83138	3.2080127857625484E-004
eed_stk	1.426989966468243E-003	3ABB09D4	1.4269896782934666E-003
eedvir_stk	-1.681304742807046E-002	BC89BB82	-1.6813043504953384E-002

7.5 Módulo Coulomb – SPLINE

Da mesma forma que na Seção 7.4, a Figura 16 apresenta o desenho simplificado que espelha o funcionamento de PMEMD, sem os sinais de controle – idênticos aos do Coulomb-EFS, à exceção do comando de leitura da memória que, nesse caso, atinge a tabela de interpolação spline cúbica com função de erro complementar (erfc spline), chamada de eed_cub, que denomina este módulo. Os dados de *timing* e uso de recursos para esse módulo, conforme levantamento oferecido pela ferramenta Xilinx ISE/XST, estão referenciados na Tabela 11 e na Tabela 12. Já a Tabela 13 faz a comparação entre os valores capturados do PMEMD e do módulo.

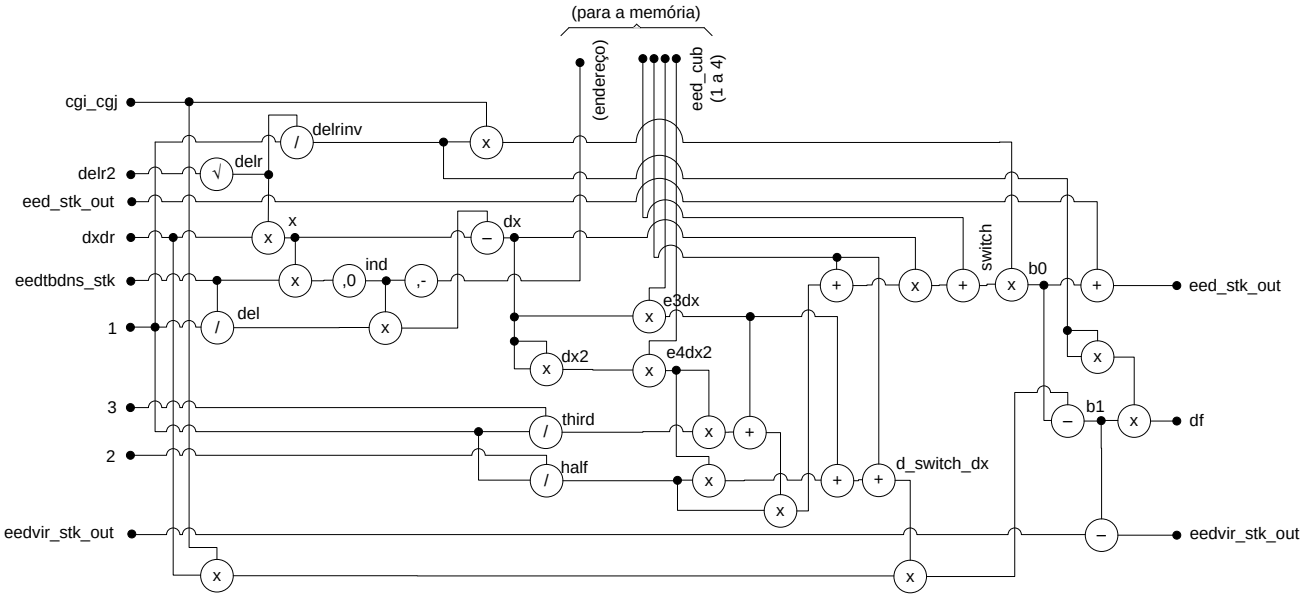


Figura 16 – Bloco para o cálculo de Coulomb com SPLINE.
Aqui ",0" representa truncamento e ",-" conversão flutuante/inteiro.

Tabela 11 – Sumário de temporização do módulo Coulomb-SPLINE
para Virtex4 FX100 1152, dados do Xilinx ISE/XST.

```
Timing Summary:
---------------
   Minimum period: 3.426ns (Maximum Frequency: 291.885MHz)
   Minimum input arrival time before clock: 3.445ns
   Maximum output required time after clock: 4.360ns
   Maximum combinational path delay: No path found
```

Tabela 12 – Sumário de design do módulo Coulomb-SPLINE
para Virtex4 FX100 1152, dados do Xilinx ISE/XST.

```
Design Summary
--------------
Logic Utilization:
  Number of Slice Flip Flops:        19,031 out of  84,352   22%
  Number of 4 input LUTs:            16,476 out of  84,352   19%
Logic Distribution:
  Number of occupied Slices:         12,263 out of  42,176   29%
    Number of Slices containing only related logic:  12,263 out of  12,263 100%
    Number of Slices containing unrelated logic:          0 out of  12,263   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:      17,331 out of  84,352   20%
    Number used as logic:            15,236
    Number used as a route-thru:        855
    Number used as Shift registers:   1,240
  Number of bonded IOBs:                292 out of     576   50%
  Number of BUFG/BUFGCTRLs:               1 out of      32    3%
```

Tabela 13 – Comparação entre resultados do PMEMD e do módulo Coulomb-SPLINE.
Variáveis em itálico indicam entradas do PMEMD para o módulo.

Variável	Valor registrado no PMEMD	Valor calculado no módulo	
		Hexadecimal	Ponto Flutuante
delr2	7.59276273717744	40F2F7EA	7.5927629470825195
cgi_cgj	62.0049738714706	42780518	62.004974365234375
dxdr	0.307676385582667	3E9D87C2	0.30767637491226196
eedtbdns_stk	5000	459C4000	5000
eed_cub_i1	0.230539510568335	3E6C128D	0.23053951561450958
eed_cub_i2	-0.549919924944569	BF0CC78D	-0.5499199032783508
eed_cub_i3	0.932444237719683	3F6EB4AA	0.9324442148208618
eed_cub_i4	-0.481538342533410	BEF68C32	-0.4815383553504944
eedvir_stk	-58.5484654463509	C26A31A0	-58.5484619140625
eed_stk	12.5738647113279	41492E8C	12.573863983154297
df	2.06495392360059	40042831	2.064953088760376

7.6 Módulo Updates

Uma vez terminado o cálculo de Coulomb (seja através do módulo Coulomb-EFS ou do Coulomb-SPLINE), diversos acumuladores devem ser atualizados. Esse processamento é feito pelo módulo Updates – representado simbolicamente na Figura 17. Esse é o único módulo onde um escravo armazena dados provenientes do FPGA nos blocos de memória – a tabela img_frc – e que, portanto, possui um sinal adicional que sinaliza à memória para arquivar o valor correspondente. O controle foi incorporado ao módulo, pois, assim, o tempo perdido na escrita fica escondido, uma vez que é executado em paralelo com outras operações. Além disso, desonera a máquina de estados e diminui a quantidade de fios ligando-a aos escravos de memória – a conexão já existe, com o módulo Updates.

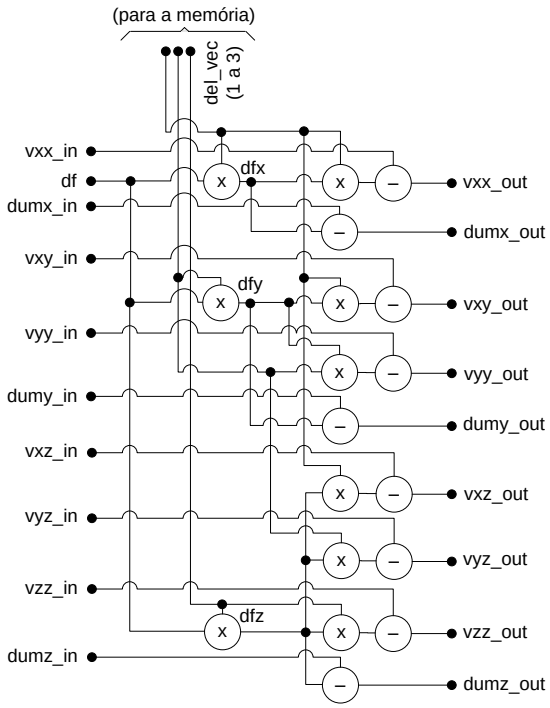


Figura 17 – Bloco para o cálculo dos acumuladores e atualizações.
O endereço é dado na máquina de estados e, portanto, não aparece na figura.

É interessante notar que não é possível sintetizar o módulo isoladamente da maneira que está disposto o diagrama, pois a quantidade de E/S necessária é maior do que a disponível na placa. Esse fato é apontado no sumário do design, apresentado na Tabela 15, em negrito. Entretanto, isso não é impeditivo para realizar simulações. Novamente, a Tabela 13 e mostra os valores capturados da arquitetura e do PMEMD.

Tabela 14 – Sumário de temporização do módulo Updates
para Virtex4 FX100 1152, dados do Xilinx ISE/XST.

```
Timing Summary:
---------------
   Minimum period: 3.365ns (Maximum Frequency: 297.194MHz)
   Minimum input arrival time before clock: 2.930ns
   Maximum output required time after clock: 5.829ns
   Maximum combinational path delay: No path found
```

Tabela 15 – Sumário de design do módulo Updates
para Virtex4 FX100 1152, dados do Xilinx ISE/XST.

```
ERROR:Pack:2309 - Too many bonded comps of type "IOB" found to fit this device.

Design Summary:
---------------
Number of errors:      1
Number of warnings:   27
Logic Utilization:
  Number of Slice Flip Flops:        11,581 out of  84,352   13%
  Number of 4 input LUTs:            10,686 out of  84,352   12%
```

```
Logic Distribution:
  Number of occupied Slices:          7,885 out of  42,176   18%
    Number of Slices containing only related logic:   7,885 out of   7,885 100%
    Number of Slices containing unrelated logic:          0 out of   7,885   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:      11,091 out of  84,352   13%
    Number used as logic:             9,885
    Number used as a route-thru:        405
    Number used as Shift registers:     801
  Number of bonded IOBs:                709 out of     576  123% (OVERMAPPED)
  Number of BUFG/BUFGCTRLs:               1 out of      32    3%
    Number used as BUFGs:                 1
```

Tabela 16 – Comparação entre resultados do PMEMD e do módulo Updates.
Variáveis em itálico indicam entradas do PMEMD para o módulo.

Variável	Valor registrado no PMEMD	Valor calculado no projeto	
		Hexadec.	Ponto Flutuante
del_vec 1	-3,24009749999999	c04f5dc2	-3,2400975227355957
del_vec 2	-2,60344490000001	c0269ed7	-2,603444814682007
del_vec 3	-5,92734089999999	c0bdacc7	-5,927340984344482
df	3,208013560454216E-004	39a8313b	3,208013658877462e-004
dfx	-1,039427671719377E-003	ba883d68	-1,0394277051091194e-003
dfy	-8,351886543095416E-004	ba5af090	-8,35188664495945e-004
dfz	-1,901498998463486E-003	baf93bb9	-1,901499112136662e-003
vxx	-3,367847000568765E-003	bb5cb719	-3,367847064509988e-003
vxy	-2,706092670856702E-003	bb3158b4	-2,706092782318592e-003
vxz	-6,161042151174028E-003	bbc9e292	-6,1610424891114235e-003
vyy	-2,174367642600051E-003	bb0e7fd6	-2,1743676625192165e-003
vyz	-4,950447869904899E-003	bba2375e	-4,9504479393363e-003
vzz	-1,127083278490164E-002	bc38a94d	-1,127083320170641e-002
dumx	-1,039427671719377E-003	ba883d68	-1,0394277051091194e-003
dumy	-8,351886543095416E-004	ba5af090	-8,35188664495945e-004
dumz	-1,901498998463486E-003	baf93bb9	-1,901499112136662e-003

7.7 Pipelining

Todas as unidades de ponto flutuante podem receber um novo dado a cada ciclo do

relógio, conforme explicado na Seção 6.4 – e essa característica não foi explorada aqui, o que poderia melhorar o desempenho. Para tanto, é necessário um estudo do caminho crítico e da árvore de dependências, pois cada unidade tem um tempo de processamento diferente (conforme a operação a ser executada) e os operadores são necessários em tempos diferentes – por exemplo, a variável *delrinv* do módulo Coulomb-SPLINE é obtida após o segundo cálculo terminar e seu valor é requisitado tanto para o terceiro quanto para o penúltimo cálculo. Caso os operadores mudem, todos os resultados intermediários devem ser guardados até chegarem ao penúltimo *core*.

Outro exemplo está no módulo Coulomb-EFS, onde as variáveis *ind* e *del_efs* estão disponíveis em tempos alternados, decorrentes de uma multiplicação e uma divisão com latências próprias – e ambos são operadores de uma multiplicação subsequente, o que obriga a manter um até o outro estar disponível. Nesse caso, é possível gerar unidades de ponto flutuante com latências diversas (o que é parametrizável pela ferramenta CORE Generator), tendo um enorme cuidado para inserir cada unidade em seu devido lugar de modo a ter os resultados no mesmo instante. Entretanto, dependendo da configuração utilizada, a inserção de novos dados pode ter uma constância diferente de uma unidade para outra, inviabilizando essa constituição.

O tamanho das memórias intermediárias (nos diferentes estágios do *pipeline*) é extenso, podendo alcançar, por exemplo, 8 * 32 * atraso de cada *core* (número de índices a serem guardados da tabela *ef_tbl*; profundidade de *bits*; atraso de três multiplicadores e um subtrator), ou pode ser ainda mais longo, no caso das realimentações. Esses são dois pontos que requerem cuidados especiais.

O acesso à memória pode ter seu desenho alterado, pois o design atual obtém os oito elementos ao mesmo tempo, o que não necessariamente é o melhor para uso com *pipeline*. Já as realimentações demandam que pelo menos um ciclo completo do módulo seja terminado, para preencher o *pipeline*. Havendo atenção aos tempos de latência para cada *core*, se podem descartar os mecanismos de *handshaking*.

7.8 Resultados

Uma vez terminado o processo de *design*, os módulos foram sintetizados para envio ao FPGA. Segundo os apontamentos da ferramenta de síntese Xilinx XST, a arquitetura como um todo pode ser executada até uma frequência de 166MHz (Tabela 17) – na prática, os testes foram efetuados com 50 e 160MHz. Entretanto, os módulos individualmente alcançam cifras bem maiores, como descrito nas Seções 7.4 a 7.6. Tal alteração de valo-

res decorre, provavelmente, do uso excessivo de *slices* no FPGA (observável na Tabela 18) – que, infelizmente, não pode ser diminuído dada a quantidade de operadores matemáticos e de memória necessários ao processamento do PMEMD. Isso acarreta problemas de roteamento e tamanhos de fios, causando atrasos para a chegada dos sinais (especialmente os dados em 32 *bits*) consequentemente, diminuindo a frequência atingível.

Tabela 17 – Sumário da temporização da arquitetura desenvolvida para Virtex4 FX100 1152, dados do Xilinx ISE/XST.

```
Timing Summary:
---------------
   Minimum period: 6.020ns (Maximum Frequency: 166.118MHz)
   Minimum input arrival time before clock: 4.903ns
   Maximum output required time after clock: 5.333ns
   Maximum combinational path delay: No path found
```

Tabela 18 – Sumário do *design* da arquitetura desenvolvida para Virtex4 FX100 1152, dados do Xilinx ISE/XST.

```
Design Summary:
---------------
Number of errors:      0
Number of warnings:  129
Logic Utilization:
  Number of Slice Flip Flops:        55,037 out of  84,352   65%
    Number of Slice FFs used for
    DCM autocalibration logic:           14 out of  55,037    1%
  Number of 4 input LUTs:            54,568 out of  84,352   64%
    Number of LUTs used for
    DCM autocalibration logic:            8 out of  54,568    1%
      *See INFO below for an explanation of the DCM autocalibration logic
       added by Map
Logic Distribution:
  Number of occupied Slices:         39,215 out of  42,176   92%
    Number of Slices containing only related logic:  39,215 out of  39,215 100%
    Number of Slices containing unrelated logic:          0 out of  39,215   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:      56,416 out of  84,352   66%
    Number used as logic:            51,157
    Number used as a route-thru:      1,848
    Number used as Shift registers:   3,411
  Number of bonded IOBs:                 40 out of     576    6%
  Number of BUFG/BUFGCTRLs:               2 out of      32    6%
    Number used as BUFGs:                 2
  Number of FIFO16/RAMB16s:             346 out of     376   92%
    Number used as RAMB16s:             346
  Number of DCM_ADVs:                     2 out of      12   16%
```

Também, diversos pontos do *hardware* poderiam receber amostragens, que não foram incluídas no projeto, talvez melhorando esse resultado. Além disso, nenhuma técnica especial de desenvolvimento (como, por exemplo, *floorplaning* – impraticável, considerando a avançada utilização de recursos no FPGA e a quantidade de interligações entre os módulos, trazendo resultados negativos) foi empregada no projeto, o que pode contribuir para a baixa frequência alcançada – em comparação com as possíveis em cada módulo individualmente.

Tabela 19 – Sumário da utilização de recursos da arquitetura desenvolvida para Virtex4 FX100 1152, dados do Xilinx ISE/XST.

```
Device Utilization Summary:
---------------------------
   Number of BUFGs                           2 out of 32      6%
   Number of DCM_ADVs                        2 out of 12     16%
   Number of External IOBs                  40 out of 576     6%
      Number of LOCed IOBs                  40 out of 40    100%
   Number of OLOGICs                        33 out of 768     4%
   Number of RAMB16s                       346 out of 376    92%
   Number of Slices                      39215 out of 42176  92%
      Number of SLICEMs                   3334 out of 21088  15%
```

Tabela 20 – Comparação entre execuções do PMEMD e do módulo Coulomb-EFS.

Variável	Valor registrado no PMEMD	Valor retornado pelo módulo
df	2.408883592579514E-004	2.408884691399771E-004
eed_stk	8.784497622400522E-004	8.784501271671030E-004
eedvir_stk	-1.467766705900431E-002	-1.467767327950283E-002
df	1.209064503200352E-003	1.209064621228592E-003
eed_stk	5.180276930332184E-003	5.180277313609063E-003
eedvir_stk	-7.449990510940552E-002	-7.449990758563596E-002
df	1.689998316578567E-004	1.689998322340714E-004
eed_stk	5.799283739179373E-003	5.799283809569067E-003
eedvir_stk	-8.522972464561462E-002	-8.522972402913223E-002

Para comprovar a eficácia da arquitetura desenvolvida, foram realizados testes com o PMEMD adaptado para comunicar com o co-processar carregado no FPGA. Os resultados demonstram que a construção está correta, havendo uma perda de precisão dado o emprego de unidades de ponto flutuante de 32 *bits*. Os valores apresentados na Tabela 20, Tabela 21 e Tabela 22 foram obtidos em diversas execuções reais do PMEMD e do

hardware no FPGA, demonstrando a validade da implementação. Os valores da Tabela 22 são resultados da execução do laço por inteiro, com 85, 34 e 55 iterações, respectivamente.

Tabela 21 – Comparação entre execuções do PMEMD e do módulo Coulomb-SPLINE.

Variável	Valor registrado no PMEMD	Valor calculado pelo módulo
DF	-3.17306995391846	-3.17307084118923
eed_stk	-6.76151323318481	-6.76151437523846
eedvir_stk	24.1611423492432	24.1611452242008
DF	1.68840694427490	1.68840723356386
eed_stk	-3.07688403129578	-3.07688353076840
eedvir_stk	11.4074249267578	11.4074223309580
DF	1.46229350566864	1.46229315524202
eed_stk	-1.90786409378052	-1.90786450430607
eedvir_stk	10.2832517623901	10.2832530988264

Além dos resultados apresentados acima, avaliou-se também o desempenho na interação hospedeiro / protótipo. Para tentar diminuir ao máximo o impacto da comunicação no desempenho, a carga dos dados foi espalhada no programa PMEMD. Isso é possível identificando os locais onde as variáveis são alteradas e adicionando primitivas para envio ou leitura das informações.

Entretanto, dois problemas atingiram o estudo. A tabela img_charge é alterada em diversos locais no código fonte do PMEMD e nem todos esses pontos foram encontrados. Além disso, a tabela img_frc não possui identificação nos elementos modificados pelo FPGA. Em ambos os casos, isso obriga enviar todo o conteúdo da tabela img_charge antes da execução do laço e a ler toda a tabela img_frc após seu término, aumentando de forma significativa o tempo de execução do programa.

Para a tabela img_frc o problema é mais grave, pois é preciso ler apenas de 1 a 128 elementos – ao forçar o acesso a toda a tabela, se lê um número de posições equivalente à quantidade de partículas no sistema simulado (no caso, são 12.461 átomos, conforme Tabela 7). Lembrando que a tabela é uma matriz $3 \times n$, somando então 37.383 posições, em contraponto ao máximo efetivamente necessário de 384.

Tabela 22 – Comparação entre execuções do PMEMD e do módulo Updates.

Variável	Valor registrado no PMEMD	Valor calculado pelo módulo
dfx	-1.652655803218322E-003	-1.652654958888888E-003
dfy	-1.050006309544946E-003	-1.050005783326924E-003
dfz	4.114606958749521E-004	4.114604962524027E-004
vxx	1.21906696826790	1.21906673908234
vxy	1.70133076468267	1.70133066177368
vxz	5.33270978988333	5.33271121978760
vyy	2.72752637297715	2.72752618789673
vyz	3.14798453667041	3.14798378944397
vzz	7.01268949723764	7.01269197463989
dumx	1.93965560645497	1.93965566158295
dumy	1.69023821018163	1.69023835659027
dumz	2.31906038527355	2.31906080245972
dfx	1.74193224942241	1.74193191528320
dfy	0.677864685469161	0.677864551544189
dfz	3.50049973942656	3.50049901008606
vxx	20.0410785053208	20.0410785675049
vxy	1.59759217586946	1.59759211540222
vxz	-1.63501582287834	-1.63501548767090
vyy	-6.07553164747079	-6.07553243637085
vyz	-1.56215377403672	-1.56215357780457
vzz	-2.12797629616537	-2.12797403335571
dumx	5.09877007995504	5.09877014160156
dumy	-2.00211882904384	-2.00211882591248
dumz	2.52983917117761	2.52983832359314
dfx	0.108945966037722	0.108945950865746
dfy	-1.172268837019160E-002	-1.172268670052290E-002
dfz	-0.391752983349912	-0.391752928495407
vxx	13.9716107512889	13.9715986251831
vxy	2.13772855448865	2.13772869110107
vxz	3.10618063989684	3.10617756843567
vyy	-8.83410840518032	-8.83410930633545
vyz	-0.760918091745936	-0.760919094085693

vzz	-2.36733381097751	-2.36733365058899
dumx	-0.474865115517444	-0.474866986274719
dumy	1.38403442376576	1.38403415679932
dumz	-1.56994467427298	-1.56994485855103

A solução encontrada foi manter o início do Laço-PCH no PMEMD, mais precisamente a linha que encontra os índices a serem manipulados nas tabelas. Assim, apenas os elementos importantes são enviados ao FPGA. Entretanto, mesmo utilizando tal técnica, o desempenho foi impactado, assim como aconteceu a Azizi *et al.* em [AZI04].

O PMEMD executando apenas em *software* precisa de cinco segundos para terminar uma simulação de apenas 10 passos ou 0,02 ps. A mesma simulação empregando o protótipo gerado leva 4 min 6 s. Ao remover a interação com as tabelas img_charge e img_frc, o tempo total cai para 2 min 23 s. Esse último dado serve apenas para comprovar que o excesso de comunicação degrada de modo sensível o desempenho do sistema, pois sem o conteúdo de tais tabelas o resultado não é válido. Dois testes foram realizados, com o protótipo sendo executado em frequências de 50 e 100Mhz, sem alteração no tempo total de execução. Esses resultados comprovam que a melhoria na comunicação entre hospedeiro e protótipo é um dos principais tópicos para estudos futuros.

8 CONCLUSÃO

Durante o percurso desta Dissertação fica claro que a demanda por aceleradores em bioinformática é alta. Essa é uma linha de pesquisa promissora e que pode trazer resultados importantes tanto para a área da tecnologia da informação quanto da biologia e química. Mesmo com todas as investigações e resultados de trabalhos paralelos, mostrados ao longo deste texto, ainda existem vários pontos que podem ser trabalhados ou aprofundados.

O uso de *hardware* para melhorar a execução do programa AMBER é um ponto de interesse. O estudo feito por Alam *et al.* [ALA07] usa HLL, que atualmente gera estruturas superdimensionadas e lentas – linguagens HDL, como VHDL e Verilog, geram estruturas menores, pontuais e mais rápidas e, portanto, são mais indicadas na construção de aplicações com FPGA. O emprego de um FPGA com tecnologia mais atual e interfaces mais modernas (e, portanto, mais eficientes) de comunicação entre *hardware* e *software* pode trazer novos resultados, passíveis de avaliação na forma como afetam os ganhos revisados ao longo da Dissertação.

Assim, a união entre a necessidade de maior desempenho para os programas de simulação por dinâmica molecular utilizados no LABIO e a ciência e equipamentos disponíveis no GAPH gerou essa Dissertação. Diversos objetivos foram traçados na Seção 1.1 e, de maneira geral, todos foram alcançados. O processo de projetar *hardware* para comunicação entre *software* e *hardware* está compreendido, o que é comprovado pela arquitetura desenvolvida aqui. A infra-estrutura implementada se mostra funcional e eficaz, como visto nos resultados da Seção 7.8. As metas elencadas na Seção 1.1 também foram atingidas, como descrito extensivamente no Capítulo 7.

Portanto, essa Dissertação documenta a criação bem sucedida de um protótipo que reproduz a atuação do PMEMD. O projeto é capaz de simular um sistema de aproximadamente 13.000 partículas, com resultados próximos aos de uma execução puramente com *software*. A variação percebida é causada pela aplicação de variáveis com precisão diferenciada (dupla, 64 *bits,* no PMEMD e simples, 32 *bits*, no FPGA). O desempenho da aplicação teve uma forte degradação, causada principalmente pela comunicação entre hospedeiro e protótipo. Além disso, a linha de pesquisa iniciada com esse trabalho abre um vasto campo de estudos, o que pode ser observado na lista de atividades futuras apresentada na seção abaixo.

8.1 Trabalhos Futuros

Enquanto esta Dissertação desenvolveu o *hardware* funcional equivalente à área de maior custo computacional no programa PMEMD (segundo [SAR09]), inclusive integrando-o a esse programa, permanecem em aberto várias opções de estudo para melhorar seus resultados. Alguns estão elencados abaixo:

- Aproveitamento dos *pipelines* internos aos *cores*. Detalhes na Seção 7.7;
- Alteração do controlador para executar o laço “gêmeo” ao laco_pch. Detalhes na Seção 6.1;
- Ampliação da abrangência do *hardware* no PMEMD. O tempo e a plataforma disponíveis à época da criação dos módulos aqui apresentados não permitiram englobar todo o módulo short_ene_vec do PMEMD. Avaliar os benefícios e programar o código restante do shot_ene_vec pode aumentar o desempenho ao manter uma maior computação no FPGA e diminuir a comunicação hospedeiro × plataforma;
- Uso da memória RAM. Implica na perda do paralelismo na aquisição dos dados das tabelas. Entretanto, pode-se buscá-los durante a execução dos cálculos, guardando-os em registradores até seu uso efetivo. Um ganho secundário é aumentar o tamanho das simulações, pois há mais espaço para manipular uma maior quantidade de partículas;
- Aumento da precisão da arquitetura para 64 *bits*. Todos os *cores*, memórias e barramentos devem ser refeitos para atingir tal precisão – facilmente alcançável com o uso da ferramenta Xilinx CoreGen. Entretanto, também provoca perda de paralelismo durante a obtenção de dados das tabelas, caso sejam geradas pelo CoreGen, por limitações da ferramenta no tamanho da porta de leitura (máximo de 256 *bits* de largura [XIL05]). A comunicação do escravo de controle com o PMEMD também precisa alteração, pois o protocolo MainBus e a interface PCI permitem o envio de apenas 32 *bits* a cada interação, obrigando a executar duas operações de escrita para obter um dado. Além disso, o espaço do FPGA na plataforma atual pode não conter todos os módulos em 64 *bits;*
- Aumento da largura da interface de comunicação para 64 *bits*, em conjunto com o barramento PCI-e – para plataformas que o possuam. Há duas opções para isso: a troca completa do protocolo de comunicação, removendo toda re-

ferência ao MainBus, ou alterá-lo de modo a utilizar todos os 64 *bits* do PCI-e – nesse caso escolhendo também alterar a arquitetura para 64 *bits* ou aproveitar para enviar duas palavras de 32 *bits* (se for mantida tal precisão na arquitetura) juntamente com seus dois endereços;

- Uso de *Direct Memory Access* – DMA. Outro ponto que pode levar a um ganho de desempenho, ao melhorar a interface de comunicação;
- Revisão do caminho crítico, substituindo os *cores* por outros que possuam Xilinx DSP48E, o que diminui o tempo total de execução dos módulos;
- Implementação de pedidos de interrupção (IRQ) no protocolo MainBus, evitando o uso de *pooling* mencionado no Capítulo 7.3.

Além dos estudos pertinentes a plataforma, a área de *software* possui uma alteração importante. A API disponibilizada por Sartin [SAR09] não é empregada nesta Dissertação, conforme apontamentos da Seção 6.1. Essa atividade requer a adaptação da API para que trabalhe com diversos escravos, segundo explicações na seção mencionada.

Ainda, os diversos autores pesquisados também indicam alguns temas de trabalhos na área. Por exemplo, Yang *et al.* [YAN07] sugerem o estudo sobre a precisão aritmética imprescindível em *hardware* para determinado tipo de simulação – ponto bastante discutido nos artigos revisados aqui. Outro tópico é o problema de *read-after-write* provocado pelo uso de *pipeline* e a latência incluída. Mais um campo de pesquisa seria implementações paralelas com múltiplos FPGAs, o que já está sendo endereçado por alguns autores como Scrofano [SCR08].

Scheraga [SCH07] afirma que os campos de força ainda não estão perfeitos, já que é possível ter resultados diferentes com campos de força diferentes, sendo uma prioridade para estudos futuros. Também diz que é necessário um método para converter trajetorias de grão grosso para trajetórias calculadas por funções de potencial *all-atom*. Gu *et al.* [GU06] indicam o uso de outros algoritmos de transformadas, como somas de Ewald ou transformadas rápidas de Fourier. Segundo os autores, o último já está em andamento, apesar de eles mesmos afirmarem em [GU07b] que é difícil implementá-lo com eficiência. Do mesmo modo, o uso de SMPE já foi revisado, mas por Scrofano [SCR06], que, por sua vez, indica outras técnicas como o método multipólo rápido (*fast multipole method*).

Luethy e Hoover [LUE04] aponta a falta de *benchmarks* como um problema para facilitar a comparação entre os diversos tipos de algoritmos e aceleradores disponíveis. Isso é facilmente percebido nos artigos que compõem a bibliografia revisada – apesar de existir um *benchmark* para AMBER, não utilizado em nenhum de tais artigos.

REFERÊNCIAS BIBLIOGRÁFICAS

[ALA07] Alam, S. R. *et al*. “Using FPGA Devices to Accelerate Biomolecular Simulations. Computer”, vol 40-3, 2007, pp. 66-73.

[ALE08] Al-Eryani, J. “FPU100”. Capturado em http://www.opencores.org/projects.cgi/web/fpu100, Novembro 2008.

[ALT04] Altera Inc. “NIOS Development Board Reference Manual, Cyclone Edition”. San Jose, CA, Altera Corporation, 2004.

[AND05] Andrade, J. A. “Bioinformática: um estudo sobre a sua importância na aprendizagem em Ciências no Ensino Médio”. Dissertação de Mestrado. Faculdade de Química - PUCRS. 2005.

[ASH90] Ashenden, P. J. “The VHDL Cookbook”. University of Adelaide, Department of Computer Science, 1990, 110p.

[AZI04] Azizi, N.; Kuon, I.; Egier, A.; Darabiha, A.; Chow, P. “Reconfigurable Molecular Dynamics Simulator”. In: *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’04)*. 2004, pp.197-206.

[BOW06] Bowers, K. J. *et al*. "Scalable algorithms for molecular dynamics simulations on commodity clusters". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'06)*, 2006.

[BIS09] Bishop, D. W. “VHDL-2008 Support Library”. Capturado em http://www.eda-stds.org/fphdl/, Dezembro, 2009.

[CAS05] Case, D. A. *et al*. “The AMBER Biomolecular Simulation Programs”**.** *Journal of Computational Chemistry*, vol. 26-16, 2005, pp. 1668-1688.

[CBR09] Computational Biology Research Center [CBRC]. “Computer System <Blue Protein>”. Capturado em http://www.cbrc.jp/eng/intro/system_blue.eng.html. Agosto 2009

[CHO08] Cho, E.; Bourgeois, A. G.; Fernandez-Zepeda, J. A. “Efficient and accurate FPGA-based simulator for Molecular Dynamics”. In: *IEEE International Symposium on Parallel and Distributed Processing (IPDPS’08)*, 2008, pp. 1-7.

[CLC08] CLC Bio. Capturado em http://www.clcbio.com/, Junho 2008.

[COM08] COMPUGEN. Capturado em http://www.cgen.com/, Junho 2008.

[COS07] Cosoroba, A. “Memory interfaces made easy with Xilinx FPGAs and the memory interface generator”. Xilinx White Paper WP260 v. 1.0, February 2007, 16p.

[CRO97] Crowley, M.; Darden, T.; Cheatham, T.; Deerfield, D. “Adventures In improving the scaling and accuracy of a parallel molecular dynamics program”. Journal of Supercomputing, vol. 11-3, 1997, pp. 255-278.

[CUM00] Cummings, C. E. “Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!” In: Synopsys Users Group (SNUG’00), 2000, 23p.

[CUM02] Cummings, C. E. “The Fundamentals of Efficient Synthesizable Finite State Machine Design using NC-Verilog and BuildGates”. In: *International Cadence Usergroup Conference (ICUC’02)*, 2002, 27p.

[DIN08] Dinechin, F. *et al*. “When FPGAs are better at floating-point than microprocessors”. In: *Proceedings of the 16th international ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA’08)*, 2008, pp. 260-260.

[GIB01] Gibas, C.; Jambeck, P. “Desenvolvendo bioinformática: ferramentas de software para aplicações em biologia”. Rio de Janeiro, RJ: Campus, 2001, 440p.

[GU06] Gu, Y.; Vancourt, T.; Herbordt, M. C. “Accelerating molecular dynamics simulations with configurable circuits”. *IEE Proceedings – Computers and Digital Techniques*. vol. 153, 2006, pp. 189-195.

[GU07a] Gu, Y.; Herbordt, M. C. “FPGA-Based multigrid computation for molecular dynamics simulations”. In: *15th Annual IEEE Symposium Field-Programmable Custom Computing Machines (FCCM’07),* 2007, pp.117-126.

[GU07b] Gu, Y.; Herbordt, M. C. “High Performance Molecular Dynamics Simulations with FPGA Coprocessors”. In: *Proceedings of Reconfigurable Systems Summer Institute (RSSI’07)*, 2007.

[GU08] Gu, Y.; Vancourt, T. V.; Herbordt, M. C. “Explicit design of FPGA-based coprocessors for short-range force computations in molecular dynamics simulations”. *Parallel Computing,* vol. 34-4 e 5, 2008, pp. 261-277.

[HAN02] Hansson, T.; Oostenbrink, C.; van Gunsteren, W. F. “Molecular dynamics simulations”. *Current Opinion in Structural Biology*, vol. 12-2, 2002, pp. 190-196.

[HAS07] Hassan, L.; Al-Ars, Z.; Vassiliadis, S. “Hardware Acceleration of Sequence Alignment Algorithms – An Overview”. In: *Proceedings of International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS’07)*, 2007, pp. 96–101.

[HER07] Herbordt, M. C. *et al*. “Achieving High Performance with FPGA-Based Computing”. IEEE Computer, vol. 40-3, 2007, pp. 50-57.

[KAR02] Karplus, M.; Mccammon, A. “Molecular dynamics simulations of biomolecules”. *Nature Structural Biology*, vol 90-9, 2002, pp. 646-652.

[KIN06] Kindratenko, V.; Pointer, D. “A case study in porting a production scientific supercomputing application to a reconfigurable computer”. In: *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’06)*, 2006, pp. 13-22.

[LAR08] Larson, R. H. *et al*. “High-Throughput Pairwise Point Interactions in Anton, a Specialized Machine for Molecular Dynamics Simulation”. In: *Proceedings of the*

14th Annual International Symposium on High-Performance Computer Architecture (HPCA'08), 2008, pp. 331-341.

[LAW00] Law, A. M.; Kelton, W. D. "Simulation Modeling and Analysis". McGraw Hill, Boston-MA, 3a. ed., 2000.

[LES08] Lesk, A. M., "Introdução à Bioinformática". Artmed, Porto Alegre-RS, 2a. ed. 2008.

[LUE04] Luethy, R.; Hoover, C. "Hardware and software systems for accelerating common bioinformatics sequence analysis algorithms". *Drug Discovery Today: BIOSILICO*, vol. 2-1, 2004, pp.12-17.

[LUN09] Lundgren, D. "FPU Double VHDL". Capturado em http://www.opencores.org/project,fpu_double, Dezembro 2009.

[LUS01] Luscombe, N. M.; Greenbaum, D.; Gerstein, M. "What is Bioinformatics? A Proposed Definition and Overview of the field". *Methods of Information on Medicine*, vol. 40-4, 2001, pp. 346-358.

[MEN07] Mentor Graphics Corp. "ModelSim SE User's Guide – Software Version 6.3d". User Guide, Mentor, 2007, 902p.

[MOR98] Moraes, F. "Linguagem de Descrição de Hardware – VHDL". Apostila de aula, PUCRS, 1998, 36p.

[NAR01] Narumi, T.; Kawai, A.; Koishi, T. "An 8.61 T_op/s molecular dynamics simulation for NACL with a special-purpose computer: MDM". *In: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (SC'01)*, 2001.

[NCB10] National Center for Biotechnology Information. ''GenBank Overview". Capturado em http://www.ncbi.nlm.nih.gov/Genbank/index.html, Janeiro 2010.

[PAR08] PARACEL. "Striking Development". Capturado em http://www.paracel.com/, Junho 2008.

[PAS02] Pascutti, P.G.: "Introdução à Modelagem e Dinâmica Molecular". In: Pedro G. Pascutti. (Org). Introdução à Modelagem e Dinâmica Molecular, 2002, pp. 1-38.

[PDB10] "RCSB Protein Data Bank". Capturado em http://www.rcsb.org/pdb/ home/home.do, Janeiro 2010.

[PEL05] Pellerin, D.; Thibault, S. "Practical FPGA Programming in C". Prentice Hall, Westford-MA, 2005.

[PER05] Perry, D. L. "VHDL Programming by Example". McGraw-Hill, Boston-MA, 4a. ed., 2002.

[PRO02] Prosdocimi, F.; Cerqueira, G. C.; Binneck, E.; *et al*. "Bioinformática: Manual do Usuário". Biotecnologia Ciência & Desenvolvimento, vol. 29, 2002, pp. 12-25.

[SAR09] Sartin, M. “Uma API de comunicação para aceleração por hardware de simuladores moleculares”. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2009, 100p.

[SCH07] Scheraga, H.; Khalili, M.; Liwo, A. “Protein-Folding Dynamics: Overview of Molecular Simulation Techniques”. *Annual Review of Physical Chemistry*, vol. 58, 2007, pp. 57-83.

[SCR06] Scrofano, R.; Prasanna, V. K. “Preliminary Investigation of Advanced Electrostatics in Molecular Dynamics on Reconfigurable Computers”. In: In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'06)*, 2006.

[SCR08] Scrofano, R.; Gokhale, M.; Trouw, F.; Prasana, V. K. “Accelerating Molecular Dynamics Simulations with Reconfigurable Computers”. *IEEE Transactions on Parallel and Distributed Systems*, vol. 19-6, 2008, pp. 764-778.

[SHA07] Shaw, D. E. *et al*. “Anton, a Special-Purpose Machine for Molecular Dynamics Simulation”. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA’07)*, 2007, pp. 1-12.

[TDG05] The DINI Group. “DN8000K10PCI User Guide”. User Manual, Revision 3, 2007.

[TDG09] The DINI Group. “MainBus Specification”. Product Specification. Capturado em http://www.dinigroup.com/product/common/mainbus_spec.pdf, Dezembro 2009.

[TIM08] Timelogic – Biocomputing Solutions. Capturado em http://www.timelogic.com/, Junho 2008.

[XIL05] Xilinx Inc. “Dual-Port Block Memory Core v6.3”. Product Specification DS235, Xilinx, 2005, 20p.

[XIL06] Xilinx Inc. “Floating Point Operator v3.0”. Especificação de Produto DS335, Xilinx, 2006, 28p.

[XIL08] Xilinx, Inc. “Development System Reference Guide 10.1”. Reference Guide, Xilinx, 2008, 500p.

[YAN07] Yang, X.; Mou, S.; Dou, Y. “FPGA-Accelerated Molecular Dynamics Simulations: An Overview”. *Lecture Notes in Computer Science 4419*. Springer-Verlag. 2007, pp. 293-301.

APÊNDICE A – PMEMD-HW

Para que os cálculos do PMEMD possam ser realizados em um *hardware* específico – no caso, a arquitetura desenvolvida no Capítulo 9 – o controle da execução deve, em algum momento, ser desviado para tal *hardware*. Analogamente, os dados que serão computados precisam ser transferidos para o dispositivo. Considere-se que a comunicação entre o programa-alvo e o FPGA causa um custo adicional em processamento que atinge o tempo total de execução e, consequentemente, o desempenho. Assim, deve-se manter a quantidade de chamadas ao *hardware* no mínimo possível.

Para alcançar esse objetivo, a transmissão de dados para o FPGA está espalhada em diversos pontos do PMEMD, mas apenas onde as variáveis ou tabelas são alteradas. Desse modo, minimiza-se a comunicação, pois, se a informação não for modificada, não é necessário transferi-la. Exemplos das transformações do código estão indicados na Figura 18, abaixo.

As primitivas para inicialização (accel_init) e descarga (accel_deinit) do *driver* foram inseridas, respectivamente, no início e ao término do programa PMEMD, no arquivo pmemd.fpp. As tabelas eed_cub e ef_tbl são completamente populadas nos arquivos pme_setup.fpp e ene_frc_splines.fpp, respectivamente. Já os pontos de preenchimento da tabela img_charge não foram todos encontrados (há referências em pelo menos seis arquivos: cit.fpp, find_img_pairs_comtran_1cut.i, find_img_pairs_comtran_2cut.i, find_img_pairs_nocomtran_2cut.i, find_img_pairs_nocomtran_1cut.i e pme_recip.fpp) e, portanto, tal tabela precisa ser transferida logo antes do laço iniciar. Isso causa uma perda de desempenho, devido ao excesso de comunicação ocasionado.

Similarmente, a tabela img_frc é totalmente transferida do FPGA para o PMEMD ao fim da execução do laço, pois não se conhecem os elementos que foram alterados. Novamente há perda de desempenho, uma vez que se transfere o total de coordenadas do sistema (no modelo utilizado são 3 x 12.461, segundo a Tabela 7) enquanto o número real varia de 1 a 128 (máximo de iterações do laço). As demais variáveis são transportadas no arquivo pme_direct.fpp e no short_ene_vec.i – o último sendo responsável pelos cálculos da soma direta de Coulomb. Considerando que agora a computação se sucede no FPGA, o código-fonte referente a tais operações pode ser removido do arquivo. A remodelação necessária pode ser vista na Figura 19.

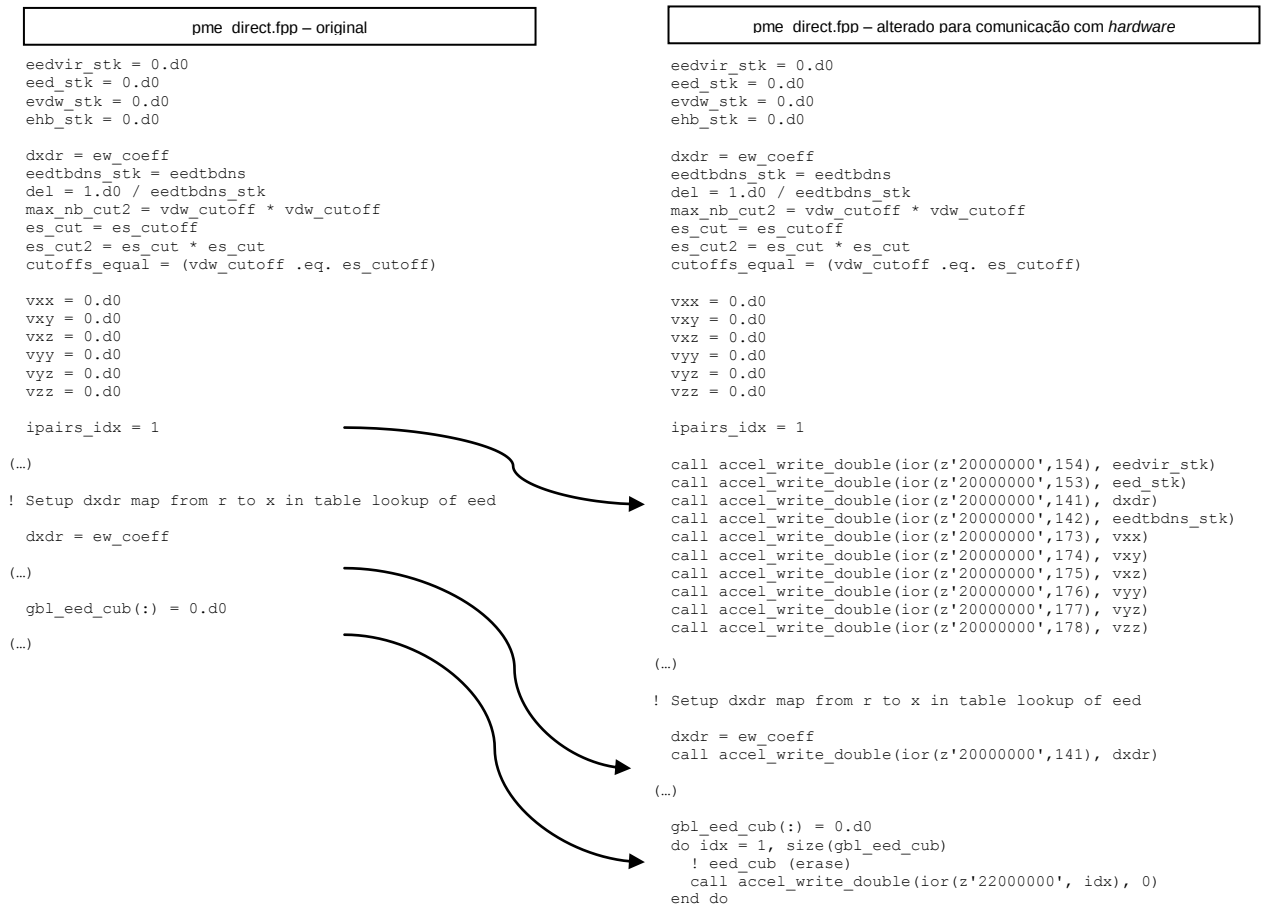
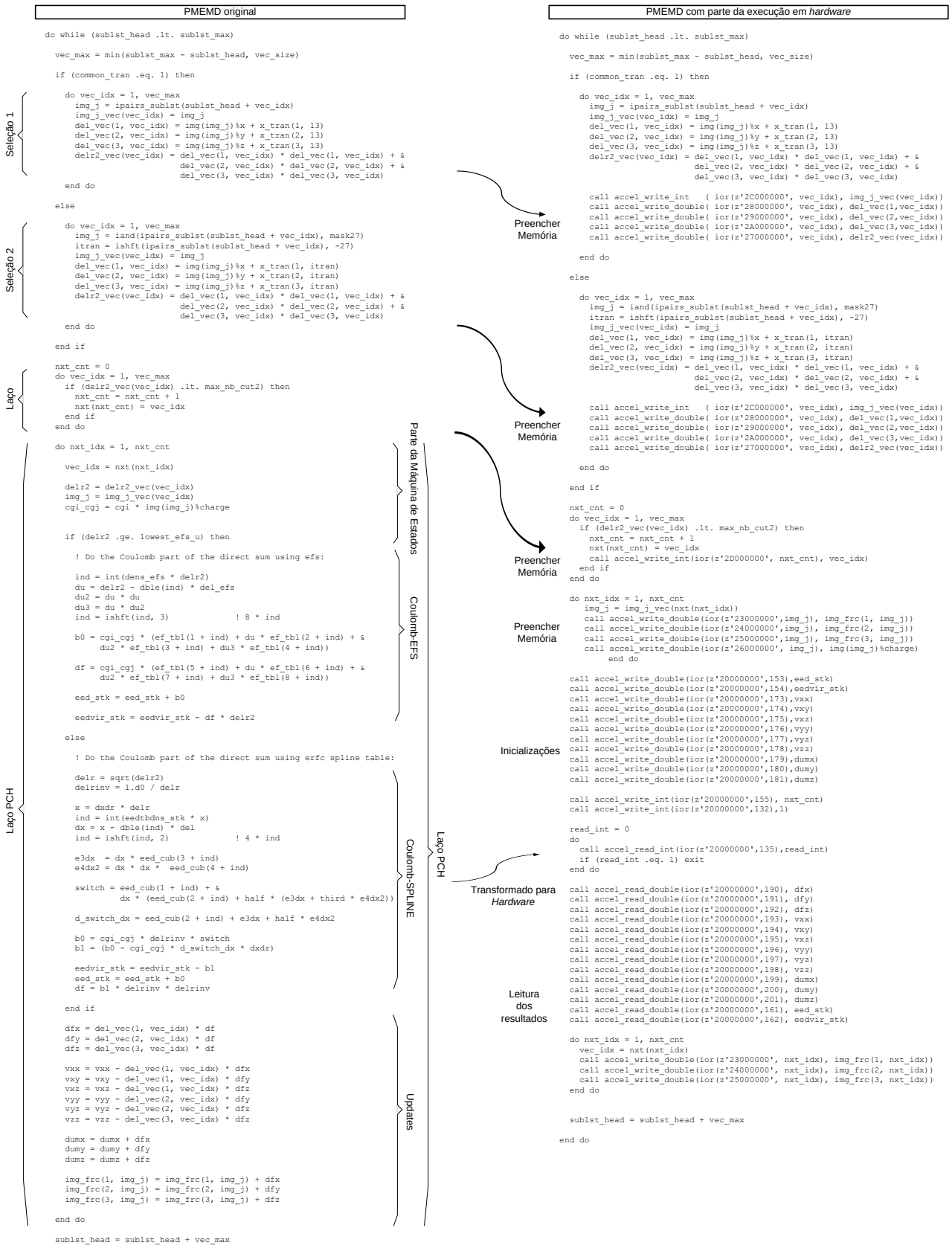


Figura 18 – Adicionando comunicação com a plataforma – arquivo pme_direct.fpp. As setas indicam onde estão as mudanças. As reticências são trechos do código que não foram alterados e, assim, estão suprimidos.




```
do while (sublst_head .lt. sublst_max)

  vec_max = min(sublst_max - sublst_head, vec_size)

  if (common_tran .eq. 1) then

    do vec_idx = 1, vec_max
      img_j = ipairs_sublst(sublst_head + vec_idx)
      img_j_vec(vec_idx) = img_j
      del_vec(1, vec_idx) = img(img_j)%x + x_tran(1, 13)
      del_vec(2, vec_idx) = img(img_j)%y + x_tran(2, 13)
      del_vec(3, vec_idx) = img(img_j)%z + x_tran(3, 13)
      delr2_vec(vec_idx) = del_vec(1, vec_idx) * del_vec(1, vec_idx) + &
                           del_vec(2, vec_idx) * del_vec(2, vec_idx) + &
                           del_vec(3, vec_idx) * del_vec(3, vec_idx)
    end do

  else

    do vec_idx = 1, vec_max
      img_j = iand(ipairs_sublst(sublst_head + vec_idx), mask27)
      itran = ishft(ipairs_sublst(sublst_head + vec_idx), -27)
      img_j_vec(vec_idx) = img_j
      del_vec(1, vec_idx) = img(img_j)%x + x_tran(1, itran)
      del_vec(2, vec_idx) = img(img_j)%y + x_tran(2, itran)
      del_vec(3, vec_idx) = img(img_j)%z + x_tran(3, itran)
      delr2_vec(vec_idx) = del_vec(1, vec_idx) * del_vec(1, vec_idx) + &
                           del_vec(2, vec_idx) * del_vec(2, vec_idx) + &
                           del_vec(3, vec_idx) * del_vec(3, vec_idx)
    end do

  end if

  nxt_cnt = 0
  do vec_idx = 1, vec_max
    if (delr2_vec(vec_idx) .lt. max_nb_cut2) then
      nxt_cnt = nxt_cnt + 1
      nxt(nxt_cnt) = vec_idx
    end if
  end do

  do nxt_idx = 1, nxt_cnt

    vec_idx = nxt(nxt_idx)

    delr2 = delr2_vec(vec_idx)
    img_j = img_j_vec(vec_idx)
    cgi_cgj = cgi * img(img_j)%charge


    if (delr2 .ge. lowest_efs_u) then

      ! Do the Coulomb part of the direct sum using efs:

      ind = int(dens_efs * delr2)
      du = delr2 - dble(ind) * del_efs
      du2 = du * du
      du3 = du * du2
      ind = ishft(ind, 3)              ! 8 * ind

      b0 = cgi_cgj * (ef_tbl(1 + ind) + du * ef_tbl(2 + ind) + &
           du2 * ef_tbl(3 + ind) + du3 * ef_tbl(4 + ind))

      df = cgi_cgj * (ef_tbl(5 + ind) + du * ef_tbl(6 + ind) + &
           du2 * ef_tbl(7 + ind) + du3 * ef_tbl(8 + ind))

      eed_stk = eed_stk + b0

      eedvir_stk = eedvir_stk - df * delr2

    else

      ! Do the Coulomb part of the direct sum using erfc spline table:

      delr = sqrt(delr2)
      delrinv = 1.d0 / delr

      x = dxdr * delr
      ind = int(eedtbdns_stk * x)
      dx = x - dble(ind) * del
      ind = ishft(ind, 2)              ! 4 * ind

      e3dx  = dx * eed_cub(3 + ind)
      e4dx2 = dx * dx *  eed_cub(4 + ind)

      switch = eed_cub(1 + ind) + &
               dx * (eed_cub(2 + ind) + half * (e3dx + third * e4dx2))

      d_switch_dx = eed_cub(2 + ind) + e3dx + half * e4dx2

      b0 = cgi_cgj * delrinv * switch
      b1 = (b0 - cgi_cgj * d_switch_dx * dxdr)

      eedvir_stk = eedvir_stk - b1
      eed_stk = eed_stk + b0
      df = b1 * delrinv * delrinv

    end if

    dfx = del_vec(1, vec_idx) * df
    dfy = del_vec(2, vec_idx) * df
    dfz = del_vec(3, vec_idx) * df

    vxx = vxx - del_vec(1, vec_idx) * dfx
    vxy = vxy - del_vec(1, vec_idx) * dfy
    vxz = vxz - del_vec(1, vec_idx) * dfz
    vyy = vyy - del_vec(2, vec_idx) * dfy
    vyz = vyz - del_vec(2, vec_idx) * dfz
    vzz = vzz - del_vec(3, vec_idx) * dfz

    dumx = dumx + dfx
    dumy = dumy + dfy
    dumz = dumz + dfz

    img_frc(1, img_j) = img_frc(1, img_j) + dfx
    img_frc(2, img_j) = img_frc(2, img_j) + dfy
    img_frc(3, img_j) = img_frc(3, img_j) + dfz

  end do

  sublst_head = sublst_head + vec_max

end do
```

```
do while (sublst_head .lt. sublst_max)

  vec_max = min(sublst_max - sublst_head, vec_size)

  if (common_tran .eq. 1) then

    do vec_idx = 1, vec_max
      img_j = ipairs_sublst(sublst_head + vec_idx)
      img_j_vec(vec_idx) = img_j
      del_vec(1, vec_idx) = img(img_j)%x + x_tran(1, 13)
      del_vec(2, vec_idx) = img(img_j)%y + x_tran(2, 13)
      del_vec(3, vec_idx) = img(img_j)%z + x_tran(3, 13)
      delr2_vec(vec_idx) = del_vec(1, vec_idx) * del_vec(1, vec_idx) + &
                           del_vec(2, vec_idx) * del_vec(2, vec_idx) + &
                           del_vec(3, vec_idx) * del_vec(3, vec_idx)

      call accel_write_int   ( ior(z'2C000000', vec_idx), img_j_vec(vec_idx))
      call accel_write_double( ior(z'28000000', vec_idx), del_vec(1,vec_idx))
      call accel_write_double( ior(z'29000000', vec_idx), del_vec(2,vec_idx))
      call accel_write_double( ior(z'2A000000', vec_idx), del_vec(3,vec_idx))
      call accel_write_double( ior(z'27000000', vec_idx), delr2_vec(vec_idx))

    end do

  else

    do vec_idx = 1, vec_max
      img_j = iand(ipairs_sublst(sublst_head + vec_idx), mask27)
      itran = ishft(ipairs_sublst(sublst_head + vec_idx), -27)
      img_j_vec(vec_idx) = img_j
      del_vec(1, vec_idx) = img(img_j)%x + x_tran(1, itran)
      del_vec(2, vec_idx) = img(img_j)%y + x_tran(2, itran)
      del_vec(3, vec_idx) = img(img_j)%z + x_tran(3, itran)
      delr2_vec(vec_idx) = del_vec(1, vec_idx) * del_vec(1, vec_idx) + &
                           del_vec(2, vec_idx) * del_vec(2, vec_idx) + &
                           del_vec(3, vec_idx) * del_vec(3, vec_idx)

      call accel_write_int   ( ior(z'2C000000', vec_idx), img_j_vec(vec_idx))
      call accel_write_double( ior(z'28000000', vec_idx), del_vec(1,vec_idx))
      call accel_write_double( ior(z'29000000', vec_idx), del_vec(2,vec_idx))
      call accel_write_double( ior(z'2A000000', vec_idx), del_vec(3,vec_idx))
      call accel_write_double( ior(z'27000000', vec_idx), delr2_vec(vec_idx))

    end do

  end if

  nxt_cnt = 0
  do vec_idx = 1, vec_max
    if (delr2_vec(vec_idx) .lt. max_nb_cut2) then
      nxt_cnt = nxt_cnt + 1
      nxt(nxt_cnt) = vec_idx
      call accel_write_int(ior(z'2D000000', nxt_cnt), vec_idx)
    end if
  end do

  do nxt_idx = 1, nxt_cnt
     img_j = img_j_vec(nxt(nxt_idx))
     call accel_write_double(ior(z'23000000',img_j), img_frc(1, img_j))
     call accel_write_double(ior(z'24000000',img_j), img_frc(2, img_j))
     call accel_write_double(ior(z'25000000',img_j), img_frc(3, img_j))
     call accel_write_double(ior(z'26000000', img_j), img(img_j)%charge)
        end do

call accel_write_double(ior(z'20000000',153),eed_stk)
call accel_write_double(ior(z'20000000',154),eedvir_stk)
call accel_write_double(ior(z'20000000',173),vxx)
call accel_write_double(ior(z'20000000',174),vxy)
call accel_write_double(ior(z'20000000',175),vxz)
call accel_write_double(ior(z'20000000',176),vyy)
call accel_write_double(ior(z'20000000',177),vyz)
call accel_write_double(ior(z'20000000',178),vzz)
call accel_write_double(ior(z'20000000',179),dumx)
call accel_write_double(ior(z'20000000',180),dumy)
call accel_write_double(ior(z'20000000',181),dumz)

call accel_write_int(ior(z'20000000',155), nxt_cnt)
call accel_write_int(ior(z'20000000',132),1)

read_int = 0
do
  call accel_read_int(ior(z'20000000',135),read_int)
  if (read_int .eq. 1) exit
end do

call accel_read_double(ior(z'20000000',190), dfx)
call accel_read_double(ior(z'20000000',191), dfy)
call accel_read_double(ior(z'20000000',192), dfz)
call accel_read_double(ior(z'20000000',193), vxx)
call accel_read_double(ior(z'20000000',194), vxy)
call accel_read_double(ior(z'20000000',195), vxz)
call accel_read_double(ior(z'20000000',196), vyy)
call accel_read_double(ior(z'20000000',197), vyz)
call accel_read_double(ior(z'20000000',198), vzz)
call accel_read_double(ior(z'20000000',199), dumx)
call accel_read_double(ior(z'20000000',200), dumy)
call accel_read_double(ior(z'20000000',201), dumz)
call accel_read_double(ior(z'20000000',161), eed_stk)
call accel_read_double(ior(z'20000000',162), eedvir_stk)

do nxt_idx = 1, nxt_cnt
  vec_idx = nxt(nxt_idx)
  call accel_write_double(ior(z'23000000', nxt_idx), img_frc(1, nxt_idx))
  call accel_write_double(ior(z'24000000', nxt_idx), img_frc(2, nxt_idx))
  call accel_write_double(ior(z'25000000', nxt_idx), img_frc(3, nxt_idx))
end do


  sublst_head = sublst_head + vec_max

end do
```

Figura 19 – Adicionando comunicação com a plataforma – arquivo short_ene_vec.i. As setas indicam onde estão as mudanças. As chaves mostram os laços designados por Sartin [SAR09] (à esquerda) e a equivalência aos módulos de *hardware* (no meio).

ANEXO A – CÓDIGO FORTRAN DO “LAÇO_PCH”

O código abaixo é parte do arquivo pme_direct.f90, gerado a partir da compilação do arquivo short_ene_vec.i que compõe o programa PMEMD. Esse laço foi transcrito para *hardware*, seguindo o levantamento efetuado por Sartin em [SAR09]. O laço mais externo foi chamado por Sartin de Laço-PCH e é controlado pelo escravo de controle e pela máquina de estados. Há um condicional, que determina qual tipo de cálculo de Coulomb será utilizado (módulos Coulomb-EFS e Coulomb-SPLINE, respectivamente), terminando com a atualização de acumuladores (módulo Updates).

```
do nxt_idx = 1, nxt_cnt

  vec_idx = nxt(nxt_idx)

  delr2 = delr2_vec(vec_idx)
  img_j = img_j_vec(vec_idx)
  cgi_cgj = cgi * img(img_j)%charge

  if (delr2 .ge. lowest_efs_u) then

       ! Do the Coulomb part of the direct sum using efs:

       ind = int(dens_efs * delr2)
       du = delr2 - dble(ind) * del_efs
       du2 = du * du
       du3 = du * du2

       ind = ishft(ind, 3)             ! 8 * ind

       b0 = cgi_cgj * (ef_tbl(1 + ind) + du * ef_tbl(2 + ind) + &
              du2 * ef_tbl(3 + ind) + du3 * ef_tbl(4 + ind))
       df = cgi_cgj * (ef_tbl(5 + ind) + du * ef_tbl(6 + ind) + &
              du2 * ef_tbl(7 + ind) + du3 * ef_tbl(8 + ind))
       eed_stk = eed_stk + b0
       eedvir_stk = eedvir_stk - df * delr2

  else

       ! Do the Coulomb part of the direct sum using erfc spline table:

       delr = sqrt(delr2)
       delrinv = 1.d0 / delr
       x = dxdr * delr
       ind = int(eedtbdns_stk * x)

       dx = x - dble(ind) * del
       ind = ishft(ind, 2)             ! 4 * ind

       e3dx  = dx * eed_cub(3 + ind)
       e4dx2 = dx * dx *  eed_cub(4 + ind)

       switch = eed_cub(1 + ind) + &
                    dx * (eed_cub(2 + ind) + half * (e3dx + third * e4dx2))
       d_switch_dx = eed_cub(2 + ind) + e3dx + half * e4dx2

       b0 = cgi_cgj * delrinv * switch
       b1 = (b0 - cgi_cgj * d_switch_dx * dxdr)

       eedvir_stk = eedvir_stk - b1
       eed_stk = eed_stk + b0
       df = b1 * delrinv * delrinv
```

```
  end if



  dfx = del_vec(1, vec_idx) * df
  dfy = del_vec(2, vec_idx) * df
  dfz = del_vec(3, vec_idx) * df

  vxx = vxx - del_vec(1, vec_idx) * dfx
  vxy = vxy - del_vec(1, vec_idx) * dfy
  vxz = vxz - del_vec(1, vec_idx) * dfz
  vyy = vyy - del_vec(2, vec_idx) * dfy
  vyz = vyz - del_vec(2, vec_idx) * dfz
  vzz = vzz - del_vec(3, vec_idx) * dfz

  dumx = dumx + dfx
  dumy = dumy + dfy
  dumz = dumz + dfz

  img_frc(1, img_j) = img_frc(1, img_j) + dfx
  img_frc(2, img_j) = img_frc(2, img_j) + dfy
  img_frc(3, img_j) = img_frc(3, img_j) + dfz


end do
```