Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

# Aceleração em Cell/B.E. da Animação de Superfícies Deformáveis

Sérgio Paulo Rodrigues de Oliveira

2009/2010
28 de Julho de 2010


Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

# Aceleração em Cell/B.E. da Animação de Superfícies Deformáveis

Sérgio Paulo Rodrigues de Oliveira (aluno nº 26416)

Orientador: Prof. Doutor Vítor Manuel Alves Duarte

*Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.*

2009/2010

28 de Julho de 2010

# Agradecimentos

Gostava de agradecer primeiramente ao meu orientador Vítor Duarte, que me acompanhou durante este longo ano. Um grande obrigado pela motivação que me transmitiu na elaboração deste trabalho, pela sua prestabilidade demonstrada nos vários obstáculos encontrados e pelo apoio que se revelou constante desde sempre.

Ao professor Fernando Birra, pelo apoio no desenvolvimento desta dissertação, por todo o tempo despendido no esclarecimento dos conceitos teóricos sobre a animação de tecidos, explicação do funcionamento do simulador e fornecimento de testes que serviram de base para as conclusões neste documento.

Ao melhor júri que me podia ser atribuído na apresentação da fase de preparação da dissertação, composto pelos professores Vítor Duarte, Fernando Birra e João Lourenço, pelas orientações oferecidas e até pela lucidez que fermentaram durante uma hora bastante esclarecedora.

Um especial obrigado ao pessoal da praceta por todas as vezes que não me deixavam acompanha-los numa partida de PES com o pretexto de eu ter muito trabalho para fazer. Também por todas as limpezas que deixei passar o prazo sem ninguém mostrar os dentes. Foram grandes.

Três obrigados para a Sophie, para a Mónica e para a Jacinta. À Sophie por todas as vezes que me desejou a fractura da tíbia ou da fíbula... ou mesmo das duas juntas. À Mónica por todas as vezes que disse que se orgulhava de mim. À Miss J pela pela constante insistência em definir o género sexual dos SPU's e por tudo o resto. Um beijo às três.

Um obrigado a todos os que não constam neste texto mas que sabem que, de alguma forma, me ajudaram a andar bem disposto e de atitude positiva.

Por ultimo, o maior obrigado de todos vai para os meus pais e para o meu irmão, por toda a paciência que demonstraram ao longo destes longos anos longe de casa. Um abraço especial para a minha mãe que não se cansava de insistir a cada fim de semana na motivação que me transmitiu. Ao meu pai pela confiança contagiante.

# Resumo

A animação de superfícies deformáveis, nomeadamente a modelação de tecidos, atravessa hoje uma época de grande relevância na indústria do cinema e no mundo dos jogos.

A grande dedicação a este tema, em termos de investigação e a evolução das capacidades das arquitecturas de computadores no que toca a poder de processamento, tornou hoje possível efectuar este tipo de simulações usando um vasto leque de técnicas com diferentes objectivos. Entre estas técnicas encontra-se a simulação através de modelos discretos. Geralmente, neste tipo de modelação, as características do tecido são discretizadas num sistema de partículas organizadas entre si segundo um esquema de forças ou energias internas. Assim, a simulação pode ser efectuada integrando o sistema de forma a calcular as novas posições das partículas ao longo do tempo. Este tipo de computação é normalmente caracterizado como sendo bastante intensivo.

A aceleração da animação de superfícies deformáveis recorrendo ao poder de processamento para além do CPU convencional foi realizada em vários trabalhos. No entanto, apenas uma pequena parte desses artigos está relacionada com a arquitectura Cell/B.E.

O Cell/B.E. foi desenvolvido por uma equipa de investigadores vindos da Toshiba, Sony e IBM. Esta equipa tinha como objectivo a criação de uma arquitectura que suportasse um elevado leque de aplicações, incluindo o suporte de uma consola de jogos, de forma eficaz e com baixo consumo de energia.

Assim, o processador Cell/B.E. convencional pode ser descrito por um chip *multicore* heterogéneo composto por um processador PowerPC e oito processadores vectoriais (SIMD) de 128 bits, permitindo assim ao programador uma maior flexibilidade na forma de paralelização de um determinado processamento.

O principal objectivo deste trabalho passou pelo estudo desta arquitectura e da forma de a explorar e avaliar as suas capacidades, aplicando-as na aceleração de um simulador de superfícies deformáveis com realismo acrescido, desenvolvido por Fernando Birra [10].

**Palavras-chave:** simulação de tecidos, métodos de integração numérica, método dos gradientes conjugados, equações diferenciais, aceleração, Cell/B.E.

# Abstract

The animation of deformable surfaces, including the modeling of cloth is going through a time of great relevance in the film industry and the gaming world.

Great dedication to this topic in terms of research, made it possible to do this type of simulations using a wide range of techniques with different objectives, simulation through discrete models being one amongst them. Usually, in this type of modeling, the cloth's properties are discretized in a system of particles related to each other through a scheme of internal forces or energies. Thus, the simulation can be done by the system's integration in order to calculate the new particles' positions over the time. This type of computing is often characterized as being very intensive.

The acceleration of the deformable surfaces' animation using processing power beyond the power of the conventional CPU was performed in several studies. However, only a small portion of those articles is related with the Cell/B.E. architecture.

The Cell/B.E. architecture was developed by a team of researchers from Toshiba, Sony and IBM. The team aimed to create an architecture that should support a large range of applications. This was to be achieved in an efficient manner, with low power consumption.

Therefore, the trivial Cell/B.E. processor can be described as an heterogeneous multicore chip containing a PowerPC processor and eight vector processors (SIMD) with 128-bit each, allowing the programmer great flexibility in the parallelization of a given process.

The main objective of this thesis was to study the afore mentioned architecture, in particular, assessing its ability to accelerate the deformable surfaces simulator with increased realism, developed by Fernando Birra [10].

**Keywords:** cloth simulation, numerical integration methods, conjugated gradient method, differential equations, acceleration, Cell/B.E.

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# Lista de Algoritmos

# 1. Introdução

A simulação do comportamento dinâmico de tecidos é um tema de investigação em voga nos dias que passam. O interesse nesta área vindo de múltiplas comunidades, tais como a indústria têxtil, o cinema ou mesmo a insdústria dos jogos de vídeo, dão a este assunto, uma grande atenção no que toca ao desenvolvimento de técnicas que tornem essa modelação possível, apesar de apresentarem objectivos bastante distintos. De salientar que a complexidade inerente ao comportamento macroscópico de um tecido apenas representa uma pequena fracção na descrição da animação a um nível mais detalhado.

Assim, o comum interesse neste assunto por parte das várias comunidades contribuiu activamente para o desenvolvimento das mais variadas técnicas no que toca à modelação de superfícies deformáveis. Os modelos que definem a estrutura da superfície, a definição das forças internas do tecido ou as técnicas envolvidas na previsão da posição e forma do tecido ao longo do tempo são hoje muito diversos e apresentam-se ainda em evolução.

Com o crescimento computacional que presenciamos, como podemos observar na arquitectura heterogénea do processador Cell e no poder de processamento vectorial deste, torna-se possível efectuar simulações de tecidos integralmente baseadas em modelos físicos, fugindo assim dos modelos geométricos uma vez que estes apenas se baseiam num modelo matemático de equações geométricas, ignorando praticamente na totalidade as características físicas da superfície.

Dentro das modelações exclusivamente baseadas em modelos físicos, destacam-se as simulações baseadas em sistema de partículas. Neste sistema, as características dos tecidos, tais como as forças, velocidades e posições são discretizadas ao longo da superfície. Os movimentos destas partículas são então integrados no tempo através de métodos de integração numérica. Neste modelo, a abordagem baseada em técnicas explícitas é caracterizada pela pouca necessidade de processamento, obtendo assim um maior desempenho. No entanto a utilização desta técnica conduz a uma maior instabilidade do sistema com o aumento do intervalo de tempo para o qual se integra os movimentos das partículas. Em alternativa, os métodos de integração implícitos vêem trazer, apesar do seu menor desempenho, uma maior estabilidade aquando do aumento do intervalo de tempo da integração.

De realçar que o presente trabalho [10] se enquadra nos objectivos da comunidade gráfica dando especial atenção ao aspecto gráfico do tecido no que toca ao realismo visual e consequentemente às propriedades macroscópicas, subvalorizando o valor das propriedades microscópicas do tecido.

## 1.1 Motivação

A principal motivação para a elaboração deste trabalho passa pela aceleração de um simulador de superfícies deformáveis com realismo acrescido desenvolvido em [10], tirando partido do modelo de programação oferecido pela arquitectura Cell/B.E. Como foi já referido, a modelação de tecidos, baseada nas suas propriedades físicas é uma tarefa extremamente complexa a nível computacional que, devido aos elevados tempos de computação requeridos, se pode mesmo tornar impraticável.

O desenvolvimento, com início no ano de 2000, por parte de uma equipa composta por perto de quatrocentos investigadores vindos da Toshiba, Sony e IBM, do Cell/B.E. foi efectuado tendo em mente o objectivo da criação de uma arquitectura de alto desempenho com capacidades para suportar um vasto conjunto de classes de aplicações. O produto desenvolvido tinha também como objectivo suportar uma consola de videojogos.

Para o desenvolvimento de tal arquitectura, foram analisados um elevado número de *workloads* na área da criptografia, física, matemática e álgebra. A equipa de investigadores teve também em consideração factores como a área da *board* utilizada, o custo de fabrico e o consumo energético. Para materializar tal ideia, a equipa atacou o problema com uma arquitectura heterogénea composta por um processador IBM 64-bit Power Architecture e vários cores orientados para o processamento vectorial segundo o modelo SIMD. Desta forma é dado ao programador o conforto e flexibilidade oferecidos por um processador convencional e o desempenho disponibilizado pelos processadores vectoriais. A IBM [1] descreve a arquitectura Cell como sendo uma estrutura de processadores orientados para o processamento distribuído.

Como incremento à motivação para a elaboração deste trabalho podemos ainda considerar alguns documentos presentes na bibliografia. Encontram-se na bibliografia documentos que comprovam o maior desempenho da Arquitectura Cell quando comparada com o CPU convencional. Em [19] são descritos os testes da execução de algumas computações relacionadas com aplicações cientificas. Segundo os autores do documento, neste tipo de computações, o Cell/B.E. obtém geralmente melhores tempos de execução que arquitecturas como o AMD Opteron, Intel Itanlium2 ou mesmo Cray X1E.

A heterogenidade alcançada no desenvolvimento desta arquitectura permite ao programador a utilização de um elevado número de modelos de programação distintos, oferecendo assim maleabilidade na criação de aplicações destinadas a serem executadas em Cell/B.E.

Entre estes modelos destacam-se a utilização dos processadores vectoriais para a execução de tarefas computacionalmente intensivas ou a organização desses mesmo processadores em *pipelines* de forma forçar o fluxo de dados a ser processado, evitando assim transferências de memória redundantes.

Os autores de [19] apontam ainda para as vantagens de uma arquitectura onde a gestão de memória é controlada pelas aplicações. Desta forma será dado um melhor uso à largura de banda disponível para transacções de dados. É assim permitido às aplicações a gestão de memória com base na informação de nível aplicacional. Segundo os autores do artigo, apesar de potenciar melhores resultados que um sistema convencional, este factor aumenta consideravelmente a complexidade do modelo de programação, tornando difícil retirar o máximo de proveito da arquitectura.

Deste modo, para motivação na realização deste trabalho podemos referir os resultados que a arquitectura Cell/B.E. tem vindo a demonstrar [19]. Assim, o objectivo desta dissertação passa pela exploração e avaliação desta arquitectura de forma a analisar as suas capacidades na aceleração do simulador de tecidos com realismo acrescido apresentado em [10]. Será também realizado um esforço na implementação de uma solução que avance nesse sentido, assim como uma posterior análise dos resultados obtidos, comparando-os com outras implementações disponibilizadas anteriormente.

## 1.2 Descrição e Contexto

O simulador em causa foi desenvolvido por Birra e tem como base o modelo de tecidos proposto por David Baraff e Andrew Witkin em 1998 [9]. Este modelo foi considerado uma referência na simulação de tecidos pela comunidade gráfica. A simulação conforme este modelo é baseada na integração implícita de Euler, encontrando-se as partículas discretizadas organizadas segundo uma malha de polígonos triangulares.

A aplicação desenvolvida por Birra pode dividir-se num ciclo de três fases distintas [18]:

**Evaluation** Na fase de avaliação do modelo de tecidos são determinadas as forças internas e externas exercidas sobre cada partícula no sistema a modelar.

**Solve** Nesta etapa são incluídas operações como a construção das matrizes das derivadas parciais e a resolução do sistema de equações gerado pelo modelo implícito.

**DLOD** A variação ao nível do detalhe consiste na criação ou destruição de partículas tendo como objectivo o maior realismo oferecido pela simulação. Nesta fase é aplicada a técnica mencionada assim como gestão das estruturas de dados associadas aos objectos que vão sendo criados ou destruídos ao longo da modelação.

Na figura 1.1 estão representados os tempos de execução tomados nas diversas fases distinguidas no simulador desenvolvido por Fernando Birra. Como se pode observar na figura, a simulação é dominada temporalmente, quase na totalidade, pela fase *Solve*. Como foi

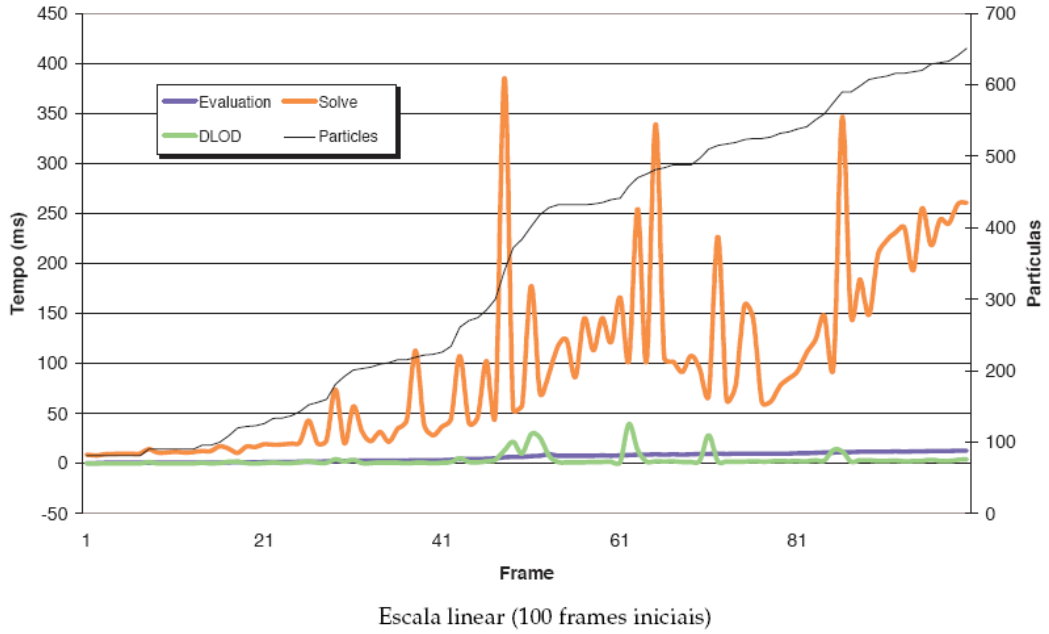


**Figura 1.1** Perfil do tempo de execução tomado pelas diversas etapas da simulação

referido anteriormente esta fase é composta pela construção das matrizes das derivadas parciais e pela resolução implícita do sistema de equações lineares.

Tendo em conta a linha representativa do número de partículas ao longo da simulação, podemos constatar que, para além de dominar o tempo de execução, a curva relativa à fase *Solve* é caracterizada com um crescimento bastante mais acentuados que as curvas representativas das fases *Evaluation* ou *DLOD*. Este fenómeno ocorre devido ao peso computacional envolvido na resolução do método de integração inserido na fase *Solve* ser consideravelmente superior ao envolvido em qualquer uma das outras fases. De salientar que, como veremos mais tarde, nesta fase predominam as operações vector-vector e matriz-vector.

## 1.3 Objectivos

Tendo em conta os aspectos referidos na secção anterior, a abordagem conveniente passa pela aceleração da fase *Solve*. O cumprimento deste objectivo pode ser alcançado através da paralelização do método dos gradientes conjugados usado para obter a solução do sistema de equações. Este objectivo será alcançado através da computação distribuída fornecida pela arquitectura Cell/B.E. e pela utilização das capacidades dos *cores* vectoriais que potencialmente permitem optimizar as operações vector-vector e matriz-vector.

Fazem assim parte dos objectivos deste trabalho os seguintes tópicos:

- Estudo e avaliação dos modelos de programação em Cell/B.E. por forma a usufruir da sua capacidade de processamento paralelo e distribuído.
- Alteração do funcionamento do simulador alvo de aceleração nesta dissertação de forma a efectuar rotinas computacionalmente intensivas como operações sobre vectores e matrizes, tirando maior proveito da arquitectura Cell/B.E. efectuando um estudo da usabilidade da biblioteca BLAS.
- Desenvolvimento e substituição de rotinas com vista a acelerar a fase *Solve* do simulador. Entre estas rotinas encontram-se uma variedade de operações sobre vectores que podem ser executadas tirando partido dos processadores vectoriais presentes na arquitectura.
- Análise de uma estrutura de dados com o objectivo de suportar representações de matrizes esparsas, tendo em conta factores como a quantidade de memória utilizada, o grau da complexidade dos cálculos associados a matrizes e a sua performance.
- Discussão dos resultados das várias abordagens possíveis seguidas na elaboração da resolução deste problema assim como a análise das vantagens e desvantagens a eles associados. Pretende-se também proceder à comparação da abordagem seguida com anteriores soluções apresentadas para outras arquitecturas.

# 2 . Modelos de Simulação de Tecidos

A complexidade por detrás da simulação dinâmica de tecidos é bastante elevada. As superfícies deformáveis possuem um grau de liberdade imensamente superior ao que encontramos em superfícies rígidas. Assim, este tema é alvo de investigação activamente abordado em comunidades relacionadas com a industria têxtil e gráfica.

Apesar da diferença nos interesses destas duas comunidades, ambos os objectivos passam pelo desenvolvimento de modelos matemáticos que permitam proceder à simulação de tecidos e determinação de vários aspectos como a aparência ou as propriedades mecânicas.

Nesta secção serão descritas sucintamente algumas técnicas de modelação de tecidos, sendo que para uma melhor compreensão é aconselhada a leitura de [10] onde é realizada uma descrição bastante mais vasta.

O assunto desta tese está relacionado com os objectivos da comunidade gráfica. Para este grupo, uma simulação de uma superfície está correcta quando assim o parecer, dando especial atenção ao realismo visual e subvalorizando o rigor da modelação física. Desta forma, podemos dividir as modelações de tecidos por três grupos diferentes: os modelos geométricos, os modelos físicos e os híbridos.

No grupo das técnicas de simulação geométricas inserem-se todas as modelações que usam um modelo matemático baseado em equações geométricas. Desta forma podemos observar neste tipo de técnicas uma grande eficiência dada a simplicidade dos cálculos usados para este fim. Esta abordagem caracteriza-se pela sua grande preocupação com o aspecto do tecido, resolvendo assim o problema da visualização, e pelo total desprezo das suas propriedades reais, impossibilitando assim a simulação de uma animação.

A modelação híbrida surgiu numa altura em que o peso das simulações baseadas no modelo físico era demasiado alto. O objectivo deste tipo de modelação passa por conjugar a modelação física e a modelação geométrica. Assim, o desempenho deste género de técnicas era melhorado ao ser efectuada uma pequena parte da simulação física apenas no inicio ou no fim, deixando o restante ser calculado por aproximação geometricamente.

Actualmente, com a crescente capacidade computacional disponível, torna-se possível efectuar simulações de tecidos integralmente baseadas em modelos físicos. Os modelos físicos, por sua vez podem dividir-se em categorias como os modelos discretos ou modelos contínuos.

Os modelos contínuos recorrem à utilização de equações diferenciais para definir a energia de deformação de toda a superfície. Assim, ao contrário dos modelos discretos, onde as partículas distribuídas ao longo do tecido representam os valores daquela zona durante a simulação, neste modelo as energias são deduzidas directamente de um sistema de equações

contínuo em todo o domínio da superfície a ser simulada. Este modelo apresenta no entanto algumas desvantagens no que toca à detecção de colisões, o que o torna impraticável uma vez que as colisões são um fenómeno bastante frequente na modelação de tecidos.

Nos modelos discretos, geralmente é usado uma malha de partículas onde são discretizados os valores daquela zona do tecido. Assim, as partículas são normalmente organizadas em polígonos triangulares ou quadriláteros e interagem entre si através de um esquema de massas e molas representantes das forças internas entre as partículas. Os cálculos usados para calcular a força ou a energia associada a cada partícula do sistema são efectuados baseando-se nas relações envolvendo as partículas vizinhas.

Assim, cada vértice corresponde a uma partícula do tecido que representa aproximadamente as grandezas de uma determinada zona do tecido como a massa, a velocidade e as forças ou energias. De realçar que a teoria relacionada com este cálculo varia substancialmente de técnica para técnica. Este tipo de simulação pode ser baseado em sistemas de equações diferenciais que nos permitem determinar os novos valores, nomeadamente a posição, associados a cada partícula integrando a sua trajectória para um determinado intervalo de tempo.

De uma forma geral, em todas as simulações que usam este tipo de modelos discretos, um denominador comum é o cálculo da resultante das forças ou energias aplicadas a cada partícula do sistema. Para determinar este valor, o algoritmo usado tem que determinar as forças provenientes de dois grupos: as forças internas que resultam das características do tecido tais como a capacidade de deformação e forças externas tais como a gravidade e o vento.

Depois de determinadas as novas posições e velocidades de cada partícula no sistema, a modelação pode avançar no tempo avaliando novamente as forças e derivadas parciais do novo estado e formulando o novo sistema de equações a ser resolvido pelo método de integração numérica. Este ciclo é então repetido até ao fim da animação, integrando sucessivamente o sistema de equações gerado em pequenos intervalos de tempo.

## 2.1 Simulação Através de Integração Numérica

Como foi descrito na secção anterior, as simulações baseadas em sistemas de partículas consistem num sistema de equações diferenciais que, ao ser integrado numericamente, determinará os valores das partículas na iteração seguinte. Esta integração do sistema de partículas pode ser realizado de forma implícita ou explícita. A diferença destes dois métodos de integração reside no facto de o método explicito apenas se basear nos valores existentes no início de cada iteração enquanto que nos métodos implícitos o cálculo do próximo estado é efectuado procurando um estado a partir do qual se possa retornar ao estado actual retirando-lhe o passo do tempo.

Os métodos explícitos, apesar da sua eficácia, apresentam no entanto alguns pontos de desvantagem. Um dos problemas descritos [8] reside no facto da instabilidade do estado do sistema aumentar consideravelmente com o aumento do intervalo de tempo entre iterações. Esta desvantagem pode ser ilustrada com um exemplo relacionado com o movimento circular uniforme infinito. Podemos observar na figura 2.1 que, ao simularmos este movimento utilizando um método explicito, a aproximação obtida será uma espiral. Podemos ainda notar que quanto maior o passo de tempo, maior será o erro de aproximação de cada iteração.

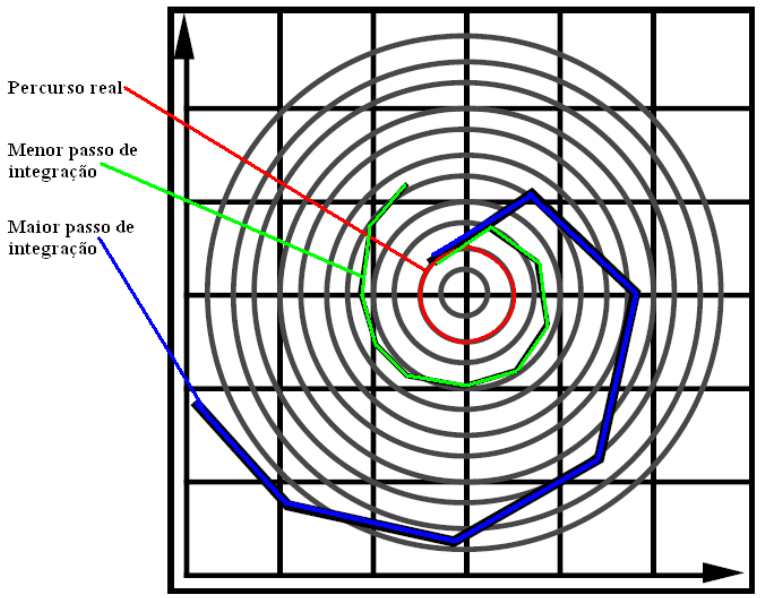


**Figura 2.1** Erro Associado ao Método Explicito de Euler no Movimento Circular Uniforme

Segundo [18] podem ser usadas várias técnicas para atenuar este tipo de comportamento tais como o pós-processamento por restrições, a aplicação por forças de amortecimento ou mesmo, como sugerido em [20] usando o método *Midpoint* ou *Runge-Kutta*. Dos métodos enumerados, o que produz melhores resultados é aquele em que se reduz o passo de tempo de cada iteração da simulação. Assim, ao reduzir o tempo, o sistema de equações terá de ser resolvido mais vezes por cada unidade de tempo simulada aumentando assim o peso computacional de cada *frame*. A utilização de métodos implícitos pode resolver todos estes problemas e permitir um maior passo de tempo, sem necessitar da aplicação de forças de amortecimento ou pós-processamento.

## 2.2 Modelo de Simulação Proposto por Baraff e Witkin

David Baraff e Andrew Witkin apresentaram, em 1998, um trabalho [9] que foi considerado uma referência na simulação de tecidos pela comunidade gráfica. Neste trabalho, os autores apresentam um modelo onde a simulação se baseia na integração implícita de Euler e onde as partículas discretizadas se organizam nos vértices de uma malha de polígonos triangulares. A dinâmica do tecido era, no entanto, avaliada segundo um modelo contínuo para permitir especificar um tecido independentemente do nível de discretização usado.

Este trabalho, que será discutido nesta secção, foi usado como referência no simulador de superfícies deformáveis com realismo acrescido desenvolvido em [10].

Entende-se por método de integração implícita de Euler (ou *Backwards Euler Method*) o método que tenta descobrir o próximo estado do sistema procurando um estado a partir do qual se possa retornar para o estado actual retirando o passo do tempo. Este método pode ser transcrito no sistema de equações (2.1) [9] reproduzido na forma vectorial.

$$\begin{pmatrix} \Delta v \\ \Delta p \end{pmatrix} = \Delta t \begin{pmatrix} M^{-1} f(p_0 + \Delta p, v_0 + \Delta v) \\ v_0 + \Delta v \end{pmatrix} \tag{2.1}$$

Nesta equação, $\Delta p$ e $\Delta v$ representam os vectores com os valores da variação de posição e velocidade para cada partícula. $\Delta t$ representa o intervalo de tempo e $v_0$ e $p_0$ contêm as velocidades e posições de cada partícula no inicio da simulação. Em $M^{-1}$ encontram-se representados os inversos das massas de cada partícula sobre a forma de uma matriz diagonal. Assim $M_i^{-1} = \frac{1}{m_i}$, sendo $m_i$ a massa da partícula $i$. De realçar que este método força a resolução de um sistema esparso de equações de dimensões *nxn* em que *n* é igual ao número de partículas presentes na simulação. Apesar desta desvantagem, a possibilidade de utilização de maiores passos de integração compensa largamente a perda de performance causada pela resolução do sistema de equações [9].

No modelo proposto por Baraff e Witkin, ao contrário do que acontece nos sistemas explícitos, onde todas as forças internas do tecido são determinadas através de expressões directas, são usados vectores condições para definir as energias. Desta forma, cada vector condição é escrito de forma a ser nulo no estado de equilíbrio do tecido tendo em conta as posições das partículas. Neste método são avaliadas forças como as provocadas pelos fenómenos de deformação transversa, curvatura e tensão. Encontra-se em [18], [10] e [9] uma vasta descrição destas três forças internas enumeradas assim como a expressão correspondente usada pelos vectores condição.

Computacionalmente, podemos referir vários factores que contribuem para a eficiência deste método. Neles constam o facto de ser usado o método implícito que oferece vantagens já mencionadas, a possibilidade deste método permitir a satisfação de restrições em simultâneo

com a integração, sem que isso aumente a complexidade temporal do algoritmo e a obtenção de maior estabilidade na simulação, que por sua vez permite o uso de forças com maior rigidez sem que seja necessário reduzir o passo de tempo.

De notar que este método, ao contrário dos métodos explícitos, necessita que o sistema de equações descrito seja resolvido em ordem a $\Delta v$ e $\Delta p$. Este sistema é não linear e pode ser transformado no sistema equivalente linear descrito em (2.2) [9]. Em [9] ou [18], encontram-se as transformações necessárias para a sua obtenção.

$$\left(M - \Delta t \frac{\partial f}{\partial v} - \Delta t^2 \frac{\partial f}{\partial p}\right) \Delta v = \Delta t \left(f_0 + \Delta t \frac{\partial f}{\partial p} v_0\right) \tag{2.2}$$

O sistema de equações apresentado é um sistema linear do tipo $Ax = b$ que agora poderá ser resolvido por métodos iterativos tais como o método dos gradientes conjugados. Este método permite obter resultados aceitáveis em poucas iterações sem que para isso necessite de alterar a matriz do sistema de equações. De relevar que existem outras soluções para a resolução de sistemas de equações tais como o método de Jacobi, não sendo no entanto abordados neste documento. É importante também referir que o simulador alvo desta dissertação, desenvolvido em [10] usa o método dos gradientes conjugados para resolver sistemas de equações.

## 2.3 O Método dos Gradientes Conjugados

Relembrando, o método de integração implícita de Euler de primeira ordem pode resumir-se num conjunto de operações composto pela avaliação das forças e derivadas parciais do estado inicial, formulação o sistema de equações a ser resolvido em ordem a $\Delta v$ e, por fim, actualização das posições e velocidades das partículas com base no resultado da resolução do primeiro sistema.

É importante referir que nas modelações recorrentes a malhas de polígonos, como é o caso do simulador desenvolvido em [10], o cálculo das forças aplicadas numa determinada partícula é calculado apenas em função das posições e velocidades associadas a um pequeno grupo de partículas vizinhas. Este facto acaba por influenciar a esparsidade das matrizes que representam as derivadas parciais oferecendo-lhes uma ocupação muito baixa.

Desta forma, as estruturas de dados com fim de suportar as matrizes devem ser escolhidas tendo em conta o seu grau de ocupação de forma a permitir o uso de algoritmos eficientes que implementam as operações matemáticas necessárias.

O método dos gradientes conjugados é um método iterativo que visa resolver sistemas

definidos positivos [1] de equações lineares. Este é também capaz de resolver sistemas com elevado grau de esparsidade. Concretamente, consiste num algoritmo iterativo que, a cada iteração, progride para uma aproximação mais rigorosas do resultado do sistema de equações. O resultado obtido é então devolvido caso o número de iterações ultrapassar um determinado limite pré-estabelecido ou quando o erro associado à solução atingir um valor aceitável.

Operacionalmente, uma vertente do algoritmo do método dos gradientes conjugados usando pré-condicionadores pode ser descrito da seguinte forma:

**Algoritmo 1** Método dos Gradientes Conjugados Pré-condicionado [12]

1: Argumentos:
2: **A** - Matriz do sistema linear
3: **M** - Pré-condicionador do sistema
4: **b** - Termo independente do sistema linear
5: **x** - Solução do sistema
6:
7: $i \leftarrow 0; r \leftarrow b - Ax; d \leftarrow M^{-1}r;$
8: $\delta_{new} \leftarrow r^T d; \delta_0 \leftarrow \delta_{new};$
9: **while** $i < i_{max}$ and $\delta_{new} > \varepsilon^2 \delta_0$ **do**
10: $\quad \mathbf{q} \leftarrow \mathbf{Ad}$
11: $\quad \alpha \leftarrow \frac{\delta_{new}}{\mathbf{d}^T \mathbf{q}}$
12: $\quad \mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{d}$
13: $\quad \mathbf{r} \leftarrow \mathbf{r} - \alpha \mathbf{q}$
14: $\quad \mathbf{s} \leftarrow \mathbf{M}^{-1}\mathbf{r}$
15: $\quad \delta_{old} \leftarrow \delta_{new}$
16: $\quad \delta_{new} \leftarrow \mathbf{r}^T \mathbf{s}$
17: $\quad \beta \leftarrow \frac{\delta_{new}}{\delta_{old}}$
18: $\quad \mathbf{d} \leftarrow \mathbf{r} + \beta \mathbf{d}$
19: $\quad i \leftarrow i + 1;$
20: **end while**

O Algoritmo 1 requer, como dados de entrada, uma matriz esparsa **A**, os vectores densos **x** e **b**, uma matriz representante dos condicionadores a ter em conta **M**, assim como um número máximo de iterações $i_{max}$ e um valor que representa a tolerância de erro $\varepsilon < 1$.

Deste modo, podemos observar que o algoritmo 1 é operacionalmente composto por duas multiplicações de matrizes esparsas por um vector denso (linha 10 e 14), três multiplicações de vectores por escalares (linhas 12, 13 e 18) duas somas de vectores (linhas 12 e 18), uma

[1] Um sistema de equações é definido positivo quando a matriz associada cumpre o critério de *Sylvester*. Por outras palavras, a matriz *nxn* é definida positiva caso todos os determinantes dos sub-blocos *mxm* do canto superior esquerdo são positivos.

subtracção de dois vectores (linha 13) e dois produtos internos entre dois vectores (linhas 11 e 16).

## 2.4 O Método dos Gradientes Conjugados Pré-condicionado Modificado

Uma variante do método dos gradientes conjugados pré-condicionado foi adoptado por Baraff e Witkin em [9] com o objectivo de resolver o sistema de equações gerado pelo método implícito de Euler. Este método desenvolvido pelos autores permite a aplicação de restrições aos movimentos das partículas sem degradar a performance do algoritmo original. Estas restrições são verificadas em cada iteração do algoritmo, descartando assim todos os resultados que não se encontrem conformes.

O algoritmo 2 descreve o método dos gradientes conjugados pré-condicionado modificado proposto por Baraff e Witkin.

Baraff e Witkin alertam também para a necessidade de um procedimento com o objectivo de filtrar os vectores. Esta operação é depois executada através de um produto de uma matriz **S** por um vector, sendo **S** uma matriz diagonal de filtragem que é aplicada ao vector resultante de **Ac**

As diferenças mais evidentes do algoritmo descrito para o algoritmo respeitante ao método dos gradientes conjugados apresentado no algoritmo 1 residem na presença da matriz **S**, a partir da qual se efectuam modificações de forma a que o sistema continue a respeitar as restrições impostas.

O algoritmo do método dos gradientes conjugados pré-condicionado modificado apresentado em [10] apresenta ainda uma pequena alteração sugerida por Ascher e Boxerman em [7]. Nesta vertente, é permitido o uso da solução anterior na fase de inicialização do algoritmo. Assim, o algoritmo toma proveito da coerência temporal do sistema, reduzindo o tempo de execução do algoritmo uma vez que o número de iterações necessárias é diminuído.

Desta forma, o algoritmo apresentado por Birra [10] apresenta uma pequena diferença do algoritmo 2 listado. Assim, o passo de inicialização da solução $\Delta\mathbf{v} = \mathbf{z}$ (linha 11) é substituído por $\Delta\mathbf{v} = \mathbf{S}\Delta v_0 + (\mathbf{I} - \mathbf{S})\,\mathbf{z}$, com $\Delta v_0$ sendo a solução do problema anterior.

A execução deste algoritmo é integrada na fase *solve* do simulador e, como tal, foi alvo de optimização recorrendo à arquitectura Cell B.E.

**Algoritmo 2** Método dos Gradientes Conjugados Pré-condicionado Modificado [9]

Argumentos:
**A** - Matriz do sistema linear
**P** - Pré-condicionador do sistema
**S** - Matriz diagonal definida por $diag\{S_1, ..., S_n\}$
**b** - Termo independente do sistema linear
**z** - Alterações de velocidade desejadas para as partículas restringidas

Resultado:
$\Delta\mathbf{v}$ - Solução do sistema

$\Delta\mathbf{v} \leftarrow \mathbf{z}$
$\delta_0 \leftarrow (\mathbf{Sb})^T \mathbf{P} (\mathbf{Sb})$
$\mathbf{r} \leftarrow \mathbf{S}(\mathbf{b} - \mathbf{A}\Delta\mathbf{v})$
$\mathbf{c} \leftarrow \mathbf{SP}^{-1}\mathbf{r}$
$\delta \leftarrow \mathbf{r}^T\mathbf{c}$
**while** $\delta > \varepsilon^2\delta_0$ **do**
    $\mathbf{q} \leftarrow \mathbf{SAc}$
    $\alpha \leftarrow \frac{\delta}{\mathbf{c}^T\mathbf{q}}$
    $\Delta\mathbf{v} \leftarrow \Delta\mathbf{v} + \alpha\mathbf{q}$
    $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{q}$
    $\mathbf{s} \leftarrow \mathbf{P}^{-1}\mathbf{r}$
    $\delta_{old} \leftarrow \delta$
    $\delta \leftarrow \mathbf{r}^T\mathbf{s}$
    $\mathbf{c} \leftarrow \mathbf{S}\left(\mathbf{s} + \frac{\delta}{\delta_{old}}\mathbf{c}\right)$
**end while**

# 3. Cell/B.E.

A arquitectura Cell/B.E. [17] surgiu após a sugestão, originada por parte da Sony e Toshiba, de criar um sistema de alta performance, com baixos custos de produção e baixo consumo energético. A arquitectura criada deveria suportar um leque abrangente de aplicações, incluindo nomeadamente o suporte de uma consola de jogos.

O design desta arquitectura foi baseado na análise de um elevado número de *workloads*, nomeadamente na área da criptografia, transformação de gráficos, física, transformadas rápidas de Fourier e operações sobre matrizes. Assim, no ano 2000 a Sony e Toshiba uniram-se à IBM Research e à IBM Systems Technology Group com o propósito de desenvolver tal arquitectura.

Tendo em conta factores como a área da *board* utilizada, o custo, os recursos energéticos e o desempenho assumidos como alvo, a equipa de *developers* atacou o problema com o paralelismo através de um elevado número de elementos num multiprocessador. De forma a manter os requisitos de recursos energéticos baixos, tomou-se a decisão de adoptar uma configuração heterogénea combinando a flexibilidade de um processador IBM 64-bit Power Architecture com o desempenho de vários cores orientados para o processamento no modelo SIMD.

Todo o desenvolvimento deste projecto envolveu perto de quatrocentos engenheiros e custou perto de quinhentos milhões de dólares [13].

## 3.1 A Arquitectura Cell/B.E.

A CBEA (Cell Broadband Engine Architecture) pode descrever-se como uma estrutura de processadores orientada para o processamento distribuído [1]. O objectivo passa pela possibilidade de implementação de um vasto número de configurações de processadores de forma a satisfazer os requisitos de um abrangente leque aplicações com diferentes tipos de comportamento.

Fisicamente, um processador compatível com CBEA é composto por um *single chip*, um módulo *multichip* ou mesmo por múltiplos módulos *multichip* numa *board*. Este desenho depende da tecnologia usada e das características da performance pretendida.

Como grande parte dos sistemas, esta arquitectura pode operar em dois modos distintos. O User Mode Environment (UME) é composto por todos os recursos disponibilizados ao programador de uma determinada aplicação. O Privileged Mode Environment (PME) é usado por software privilegiado tais como os sistemas operativos.

Logicamente, o CBEA compreende alguns componenentes como o *PowerPC Processor Element* (PPE), o *Synergistic Processor Element*, um controlador de *interrupts* (*Internal Interrupt Controller* - IIC) e um *Element Interconnect Bus* (EIB). O *Synergistic Processor Element* (SPE) é uma unidade composta por um *Synergistic Processor Unit* (SPU), a sua própria memória local, um *Memory Flow Control* (MFC), um gestor de memória (MMU) e um gestor de TLB's e cache (*Replacement Management Table* - RMT). O *Element Interconnect Bus* (EIB) é responsável por estabelecer uma ligação entre os vários componentes dentro do processador.

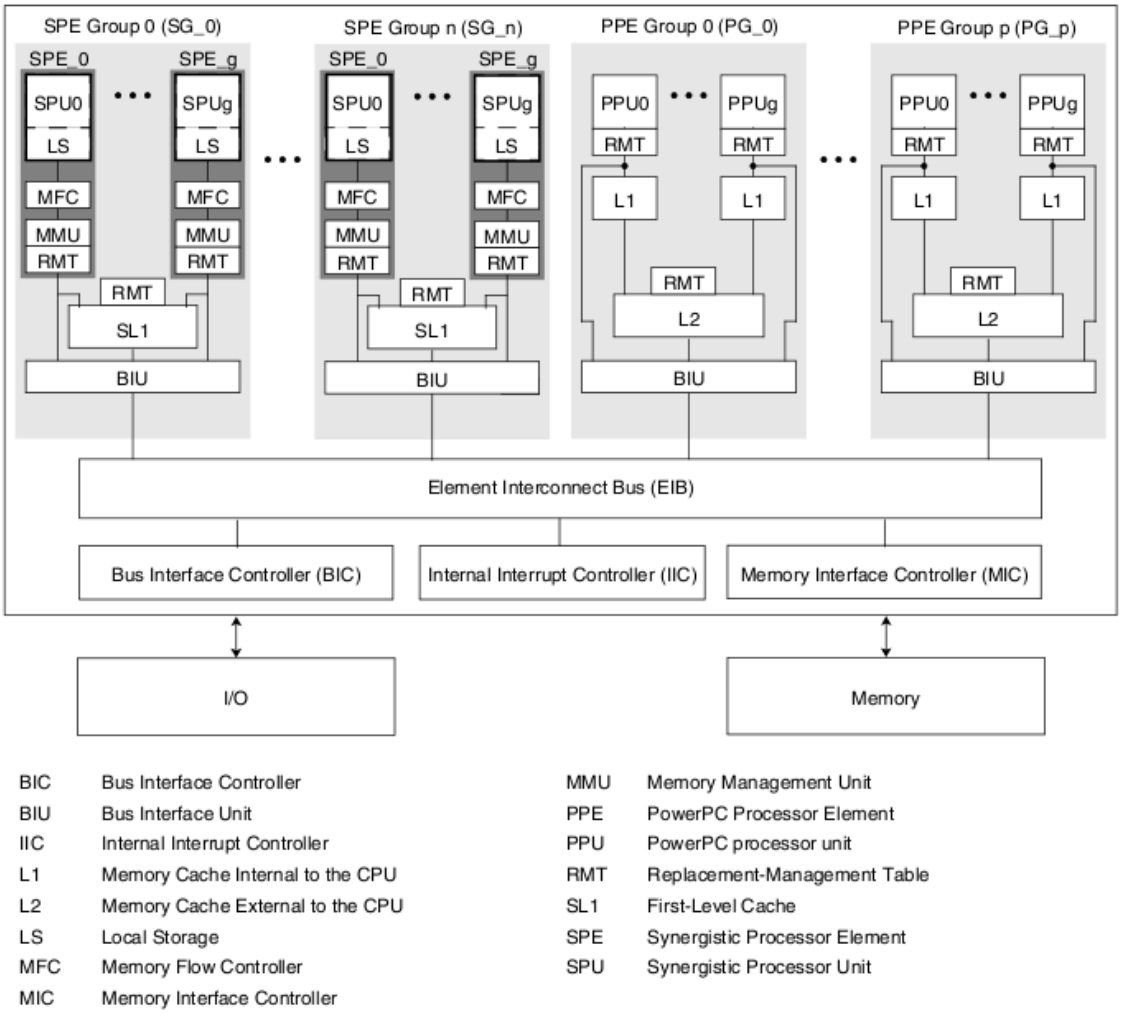


**Figura 3.1** Esquema Pormenorizado de um Processador de Acordo com Cell Broadband Engine Architecture [1]

A figura 3.1 representa um esquema de um processador de acordo com a arquitectura Cell/B.E. Na ilustração está esquematizada o agrupamento de SPE's e PPE's em grupos. Os

elementos contidos no mesmo grupo podem partilhar recursos tais como a *cache* sendo que, para o processador estar de acordo com CBEA, esta partilha deve ser transparente para as aplicações.

Resumindo, um sistema de acordo com o standard CBEA deve incluir um ou mais PPE's, um ou mais SPE's (composto por um SPU, uma memória local, um MFC e uma RMT), o controlador de *interrupts* (IIC) e um Element Interconnect Bus.

### 3.1.1 PowerPC Processor Element

Como referido anteriormente, o processador CBEA-*compliant* inclui um ou mais PPE's. Estes PPE's podem ser descritos como processadores 64 bits e respectiva cache conforme a arquitectura PowerPC descreve. Opcionalmente, o PPE pode incluir uma extensão para suportar SIMD.

Os PPE's são descritos em [1] como unidades de processamento, com um propósito generalista, que podem aceder a recursos de gestão de sistema. É permitido a estas unidades o endereçamento directo a qualquer um destes recursos. Uma das responsabilidades dos PPE's num sistema CBEA-*compliant* é a gestão e atribuição de tarefas aos SPE's.

### 3.1.2 Synergistic Processor Unit

Num sistema CBEA-*compliant* podem ser encontradas uma ou mais SPUs. Estas unidades são computacionalmente menos complicadas uma vez que não efectuam gestão do sistema. Estes componentes tem capacidade de efectuar cálculos usando o modelo SIMD. Tipicamente são encarregues de iniciar as transferências de dados necessárias e processar os dados de forma a executar o trabalho para o qual foram destacados. Os SPUs servem o propósito de permitir às aplicações computacionalmente densas o uso deste tipo de processamento.

### 3.1.3 Memory Flow Controler

Os MFC's são responsáveis pelas transferências de dados, protecção e sincronização entre a memória principal e a memória local a cada SPE, ou mesmo entre várias memórias locais. O objectivo da arquitectura, no que toca às transferências de dados entre memórias, passa pelo seu desempenho e pela *fairness* entre vários os pedidos, maximizando assim o *throughput* do processador CBEA-*compliant*.

São chamados comandos MFC aos comandos de transferência de dados. Estes comandos envolvem transferências DMA (Direct Memory Access) entre a memória local a cada SPU e a

memória principal. Cada MFC pode suportar várias transferências de dados simultaneamente. Para tornar isto possível, o MFC gere filas com pedidos de transferências de dados, usando para isso uma fila associada a cada SPU e uma outra para outros dispositivos e processadores. Segundo [1], uma fila FMC é mantida para cada SPU numa arquitectura CBEA-*compliant*, podendo no entanto em algumas arquitecturas esse MFC ser partilhado por um grupo de SPUs. Neste caso é exigido que, apesar desta partilha, cada MFC seja, do ponto de vista do software, dedicado a cada SPU.

Cada pedido de transferência de dados (MFC DMA command) é definido, entre outros parâmetros, por um endereço local (Local Storage Address - LSA) e um endereço efectivo. O endereço guardado em LSA diz respeito à uma área associada ao SPU. O endereço efectivo diz respeito à área da memória global, que por sua vez inclui as memórias locais de todos os SPUs. O MFC disponibiliza dois tipos de interfaces aos elementos presentes no sistema: (1) o SPU Channel que pode ser usado apenas por SPUs para controlar o MFC, que manipula apenas a fila de pedidos relativos a essa SPU e (2) o Memory-Maped Register, que pode ser usado por outros *devices* ou processadores em todo o sistema. O MFC também suporta reserva de largura de banda e sincronização de dados de forma a permitir o uso de memória central partilhada entre SPE's e um justo equilíbrio e coordenação nos acessos realizados por estes.

De referir ainda que o MFC não garante o cumprimento das transferências na ordem pela qual foram dados os comandos DMA. Todas as transferências são sujeitas a uma reordenação de forma a que o MFC obtenha um maior desempenho possível. Esta reordenação ocorre quando, por exemplo, se pretende aceder a duas zonas de memória central e apenas uma delas se encontra em *RAM*, encontrando-se a outra em *SWAP*. Neste caso o MFC força a que a primeira transferência efectuada corresponda aos dados armazenados em *RAM*.

No caso da *libspe2*, disponível no SDK, são oferecidos métodos que permitem organizar os comandos MFC DMA pela ordem pretendida. Assim são disponibilizados dois modelos que permitem ao programador manipular este mecanismo. Todas as transferências de memória podem ser agendadas de dois modos para além do *default*. São eles o *fence*, em que é ordenado ao MFC que termine todas as comandos identificados com o mesmo *tag* agendados previamente, e o *barrier*, onde o MFC é forçado também a terminar todas as transferências previamente agendadas identificadas com o mesmo tag, transferindo depois os dados em causa e, só então processar os comandos agendados posteriormente. Estes mecanismos são oferecidos pelos comandos *GETF*, *PUTF*, *GETB* e *PUTB*.

### 3.1.4 Internal Interrupt Controller

O Internal Interrupt Controller gere as prioridades dos *interrupts* apresentados aos PPE's. O seu objectivo principal passa por garantir o processamento de *interrupts* gerados por todos os componentes do processador CBEA-*compliant* sem que para isso se recorra ao principal

controlador de *interrupts* do sistema. Isto torna o IIC num controlador de *interrupts* de segundo nível uma vez que, enquanto o principal controlador de *interrupts* do sistema gere os *interrupts* originados em dispositivos externos (por exemplo dispositivos de I/O), o IIC gere apenas os *interrupts* internos ao processador. De salientar que o propósito deste elemento na arquitectura CBEA não passa pela substituição do controlador de *interrupts* principal.

### 3.1.5 Especificação do Computador Cell/B.E. Usado Nesta Dissertação

No caso particular desta dissertação, todo o desenvolvimento foi efectuado em computadores *IBM BladeCenter QS21*. Este modelo da IBM cumpre os requisitos impostos pela arquitectura Cell/B.E. e é caracterizado pelas seguintes especificações:

**Processador** Dois Processadores Cell/B.E. a 3.2 GHz, sendo cada um composto por um PPE equipado com 512 KB de *cache* de nível L2 e oito SPE's com 256 KB de memória local.

**Memória Central** 2 GB.

**Network** Dual Gigabit Ethernet

**Sistema Operativo** Red Hat Enterprise Linux 5

No desenvolvimento estavam disponíveis duas *blades* com as características acima descritas.

## 3.2 Aritmética Em Vírgula Flutuante

As operações sobre números representados em vírgula flutuante são uma constante em muitos processos de simulação como a animação de superfícies deformáveis. Uma vez que uma enorme parte do processo de animação se baseia em operações aritméticas de conjuntos de números reais, torna-se importante tomar conhecimento se o processador utilizado para tornar possível a aceleração cumpre as normas que regulam a aritmética de vírgula flutuante.

O compilador XL C/C++, também utilizado para compilar código desenhado para ser executado em SPU, e a arquitectura Cell B.E. suportam formatos de vírgula flutuante com precisões de *32-bit* e *64-bit* em PPU e SPU e *128-bit* em PPU [16]. Este compilador tira partido de instrução *madd* disponível nos processadros Cell. Esta instrução não conforme com a norma IEEE 754 [2] é responsável por computar operações do tipo $a + b * c$ num único ciclo de processador com o objectivo de aumentar o desempenho da aplicação. A instrução é caracterizada por não é efectuar nenhum arredondamento entre a multiplicação e a soma, produzindo assim resultados mais precisos, podendo no entanto levar a resultados diferentes de

um processador convencional. Esta pequena optimização pode até provocar que para uma operação do tipo $x * y - x * y$ se possa obter um resultado diferente de zero uma vez que o primeiro produto é arredondado, sendo depois subraído a este o produto dos mesmos coeficientes sem arredondamento.

Na bibliografia é ainda referido o facto dos SPU's procederem ao armazenamento de *floats* segundo a norma IEEE 754 apesar dos resultados calculados não serem os resultados esperados num processador de acordo com a norma [5].

A diferença entre os resultados de cálculos obtidos num processador conforme com a norma IEEE 754 e o SPU pode ser justificada com as seguintes propriedades [5]:

- Apenas o arredondamento para zero é suportado em SPU.
- Operandos não normalizados são tratados como zero e resultados não normalizados são forçados para zero.
- Números com o valor binário do expoente representado apenas por bits a 1 (um) são tratados como valores normalizados e não como infinito ou *NaN*.

Para a leitura de uma enumeração mais exaustiva sobre as não conformidades dos SPU's com a norma IEEE 754 aconselha-se a consulta de [5].

No contexto da aceleração do simulador de superficies deformáveis podemos concluir que os resultados obtidos, nomeadamente na multiplicação de matrizes ou no cálculo do produto interno de dois vectores, poderão diferir dos valores determinados para a mesma simulação nos CPU's convencionais.

No apêndice A encontra-se listado o código e o *output* obtido de um pequeno programa escrito em linguagem C desenvolvido com o objectivo de demonstrar a diferença dos resultados de uma operação executada em vários processadores. Nesse programa é apenas determinado o produto interno de dois vectores com três posições. Apesar de todos os valores operandos do cálculo possuírem a propriedade de serem representáveis em *floats* de 32 bits, o resultado obtido difere quando executado em PPU (PowerPC), SPU e Intel.

## 3.3 Modelos de Programação na Arquitectura Cell

Os principais modelos de programação em arquitectura Cell foram inicialmente identificados em [14]. No documento, os autores distinguem seis modelos distintos:

**Function offload model** Neste modelo estão incluídos todas as aplicações que são executadas sobre o PPE e usam os SPE's, através de uma biblioteca, para realizar tarefas computacionalmente pesadas de forma a atingir um melhor desempenho. Desta forma a componente lógica da aplicação não é modificada. Segundo os autores, este modelo poderá ser o modelo que apresenta uma maior rapidez de produção enquanto efectua um bom uso das capacidades da arquitectura.

**Device extension model** Este modelo é uma variante de *Function offload model*, onde os SPE's garantem os serviços anteriormente fornecidos por dispositivos externos. Tipicamente, neste modelo os SPE's agem como um *front-end* inteligente para dispositivos usando, para estabelecer comunicação com o SPE, *on-chip mailboxes* ou resgistos do SPE acessíveis através de memória mapeada.

**Computation acceleration model** Esta técnica é usada quando a aplicação é executada maioritariamente nos SPE's, processando estas uma grande parte computacional do problema. Aqui o PPE age como um motor de serviços ao dispor dos SPE's. Geralmente este modelo poderá tirar proveito dos mecanismos de memória partilhada, podendo no entanto usar uma interface de transferência de mensagens.

**Streaming model** Aqui os SPE's encontram-se organizados segundo um *pipeline*. Assim, cada SPE é responsável por efectuar uma determinada operação sobre um fluxo de dados, distribuindo assim diferentes tarefas por diferentes unidades. Desta forma, depois de processados os dados que deram entrada num SPE, este deverá enviar os resultados para posterior processamento por um SPE diferente. Esta técnica tira proveito do facto da largura de banda disponível entre SPE's ser superior à que diz respeito ao acesso a memória principal. De salientar que uma determinada aplicação só poderá tirar o máximo proveito desta técnica se as diferentes tarefas atribuídas aos SPE's partilharem entre si tempos equilibrados de execução para um determinado volume de dados.

**Shared memory multiprocessor model** Aqui os SPE's e os PPU's podem interoperar num modelo de programação de memória partilhada. No que diz respeito aos SPE's, todas os comandos DMA são coerentes com a *cache*. Assim um convencional acesso a memória é substituido pela execução de um comando DMA. De salientar ainda que leituras e escritas atómicas de valores em memória partilhada podem ser alcançadas através dos comandos DMA *lock*.

**Asymmetric thread runtime model** O uso desta técnica deverá ser conjugado com uma das acima descritas. Aqui é permitido a gestão de *threads* tanto no PPU como nos SPU's. Segundo o autor, esta técnica poderá atingir altos níveis de flexibilidade, apresentando no entanto um elevado custo de desempenho e recursos.

## 3.4 O Uso do Processador Cell em Aplicações do Domínio Científico

Em [19], os autores examinam o desempenho da arquitectura Cell/B.E. para fins científicos. Para o efeito foram testados nesta arquitectura um leque de computações cientificas: multiplicação de matrizes densas, multiplicação de matrizes esparsas por vectores, *stencil computations* [15] e transformadas rápidas de Fourier. Os autores analisam ainda os resultados do desempenho do Cell/B.E. quando comparados com os *benchmarks* de arquitecturas como AMD Opteron, Intel Itanlium2 e Cray X1E.

Em [19] é ainda descrito as vantagens de uma arquitectura onde a gestão de memória é controlada por software (como o Cell/B.E.) face a uma arquitectura de cache convencional. Mais concretamente, ao oferecer às aplicações o poder para controlar a memória, será dado um melhor uso à largura de banda disponível para transacções. Assim, cada aplicação terá permissão para efectuar *fetches* controlados cirurgicamente e garantir uma política de cache que tira proveito da informação de nível aplicacional. Este modelo pode ser suportado por uma arquitectura bastante menos complexa que os sistemas tradicionais. Segundo os autores do artigo, o controlo explícito do software sobre a memória aumenta a complexidade do modelo de programação mas obtém resultados onde se observa uma menor latência de memória, exigindo assim uma cache de menor capacidade que os sistemas convencionais.

Segundo algumas análises efectuadas [19], a arquitectura apresenta resultados de desempenho e consumo de energia mais satisfatórios que qualquer uma das outras três arquitecturas testadas para a maioria dos casos. Estes resultados podem ser justificados pela previsibilidade do comportamento das aplicações testadas, pelas grandes dimensões de dados transferidos que contribuem para o melhor uso da largura de banda disponível e pela previsibilidade do padrão de acesso a memória. Os testes executados demonstram no entanto algumas desvantagens nas operações de multiplicações de matrizes. A ausência de suporte de carregamento de memória desalinhada obriga o processador a permutar dados, diminuindo assim o desempenho nos testes. No entanto, apesar desta desvantagem, a arquitectura Cell continua a apresentar os melhores resultados quando comparados com as três arquitecturas analisadas.

| | Speedup vs. | | | Power Efficiency vs. | | |
|---|---|---|---|---|---|---|
| | X1E | AMD64 | IA64 | X1E | AMD64 | IA64 |
| GEMM | 0.8x | 3.7x | 2.7x | 2.4x | 8.2x | 8.78x |
| SpMV | 2.7x | 8.4x | 8.4x | 8.0x | 18.7x | 27.3x |
| Stencil | 1.9x | 12.7x | 6.1x | 5.7x | 28.3x | 19.8x |
| 1D FFT | 1.0x | 4.6x | 3.2x | 3.0x | 10.2x | 10.4x |
| 2D FFT | 0.9x | 5.5x | 12.7x | 2.7x | 12.2x | 41.3x |

**Tabela 3.1** Comparação de *Speedup* e Eficiência Energética de Cell/B.E. e Outras Arquitecturas [19]

Na tabela 3.1 pode-se observar os ganhos de *speedup* e eficiência energética observados em Cell quando comparados com arquitecturas como o AMD Opteron (AND64), Intel Itanlium2 (IA64) e Cray X1E (X1E).

# 4 . Tecnologias Relacionadas

Feita a descrição do problema origem da motivação para esta dissertação e da arquitectura Cell/B.E., encontramo-nos numa posição confortável para decrever algum trabalho relacionado que se pode ter em conta aquando do desenvolvimento da solução para este trabalho.

Assim, neste capítulo serão descritas algumas tecnologias que, de alguma forma, podem contribuir para a concretização da solução pretendida.

## 4.1 Basic Linear Algebra Subprograms

O BLAS (Basic Linear Algebra Subprograms) [11] é uma biblioteca que disponibiliza um conjunto de rotinas para operar sobre vectores densos e matrizes. As funções fornecidas dividem-se em três níveis. As rotinas de nível 1 efectuam operações sobre vectores tais como somas ou produtos internos, o nível 2 disponibiliza operações matriz-vector e o nível 3 operações matriz-matriz tais como a multiplicação de duas matrizes. Estas rotinas podem ter até quatro versões, podendo operar sobre números complexos ou reais, usando precisão simples ou dupla.

Esta biblioteca é desenvolvida tendo em conta aspectos como o desempenho das operações fornecidas e a sua portabilidade.

O BLAS é bastante usado noutras bibliotecas de mais alto nível como o LAPACK e o ScaLAPACK.

Estão também disponíveis bibliotecas BLAS optimizadas para algumas arquitecturas específicas. Estas bibliotecas são geralmente fornecidas pelo *vendor* da arquitectura em causa ou por um independente, permitindo assim a portabilidade do código sem sacrificar o seu desempenho.

### 4.1.1 BLAS em Cell/B.E.

O actual Software Devellopment Kit (SDK 3.1) disponibilizado pela IBM inclui nas suas bibliotecas o BLAS. Segundo a documentação [3], as rotinas contidas nesta versão da biblioteca operam sobre números reais e complexos. No entanto, nenhuma das operações sobre números complexos se encontra optimizada no que toca ao uso dos SPE's. Paralelamente, todas as rotinas sobre números reais encontram-se implementadas sobre PPE, estando algumas delas também implementadas sobre SPE. No SDK é também disponibilizado

uma interface para SPE, não estando no entanto conforme a interface standard definida no BLAS Technical Forum.

#### 4.1.1.1 Método dos Gradientes Conjugados Usando o BLAS

Ao analisarmos o problema, constatamos que o método dos gradientes conjugados, em cada iteração, é composto essencialmente por operações sobre vectores (soma, subtracção e multiplicação por um escalar), produtos internos de vectores e multiplicação de matrizes esparsas por vectores densos.

Podemos constatar que todas operações efectuadas se encontram implementadas no BLAS sobre SPU's [3]. De seguida é realizada uma listagem das funções que implementam as operações em causa.

**sscal_spu / dscal_spu** Escala um vector por uma constante (nível 1).

**saxpy_spu / daxpy_spu** Escala um vector e soma-o, elemento a elemento, a um vector destino (nível 1)

**sdot_spu / ddot_spu** Determina o produto interno entre dois vectores (nível 1).

**sgemv_spu / dgemv_spu** Multiplica uma matriz por um vector, somando o resultado a um vector destino (nível 2).

Tendo em conta a disponibilidade das rotinas apresentadas no SDK 3.1 da IBM e as operações presentes no método dos gradientes conjugados pré-condicionado modificado implementado em [10], torna-se então possível proceder à alteração de fracções do simulador de forma a potenciar a sua aceleração recorrendo ao BLAS. Este procedimento cai dentro do modelo de programação descrito em *Function Offload Model* (secção 3.3) e permite a sua concretização sem alterar a lógica do programa original. Esta biblioteca não suporta no entanto a multiplicação de matrizes esparsas por vectores.

## 4.2 Concurrent Number Cruncher

O CNC (Concurrent Number Cruncher) [12] é uma biblioteca de alta performance que implementa o método dos gradientes conjugados sobre a plataforma CTM da ATI, também disponível para CUDA. Esta foi desenvolvida com o intuito de tirar o maior partido das capacidades de um GPU na computação do método dos gradientes conjugados.

A biblioteca é baseada, entre outros factores menos relevantes para o assunto em causa, numa implementação optimizada de matrizes esparsas usando o formato BCRS e operações BLAS paralelizadas intensivamente. Com esta implementação, os autores apresentam como motivação o uso eficaz de processadores vectoriais e da largura de banda disponível no GPU.

### 4.2.1 Compressed Row Storage

O CRS (Compressed Row Storage), ilustrado na figura 4.1, é um formato usado para representar matrizes esparsas. Neste formato são guardados os valores não nulos da matriz num único *array* (Non-zero Values). Para gerir essa informação, é usado um sistema de endereçamento indirecto aos valores da matriz. Esta meta-informação é armazenada em dois *arrays* adjacentes: (1) o 'row pointer table' que permite determinar as posições do *array* que dizem respeito a uma determinada linha da matriz, e (2) o 'column index table' contendo informação em que coluna da matriz se encontra o elemento correspondente.

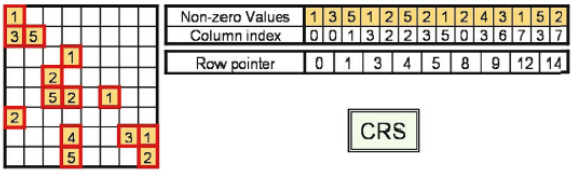


**Figura 4.1** Exemplos de Compressed Row Storage (CRS)

A título de exemplo podemos descrever o processo tomado para efectuar a leitura dos dados relativos à segunda linha da matriz. Como descrito, as posições relativas ao início de cada linha são indexados no vector *row pointer*. Assim podemos afirmar que o inicio da linha 1 se encontra na posição indicada por $rowpointer[1](1)$ e termina (exclusive) na posição indicada por $rowpointer[2](3)$. Ao observarmos o vector $columnindex$ e $non-zerovalues$ nas posições 1 e 2 podemos concluir que o valor 3 se encontra na posição $(1,0)$ e o valor 5 se encontra na posição $(1,1)$.

#### 4.2.1.1 Multiplicação de Matrizes Esparsas por Vectores Densos

A multiplicação tradicional de uma matriz **A** de dimensões *nxm* por um vector **x** pode ser descrito no algoritmo 3:

Assim, usando o formato CRS, a multiplicação de uma matriz esparsa por um vector pode ser implementada tal como indicado pelo algoritmo 4.

O número de operações envolvidas nesta rotina é da ordem de duas vezes o número de elementos não nulos da matriz. Logo, este formato reduz claramente o tempo de

**Algoritmo 3** Multiplicação de uma matriz por um vector

```
1: for i = 0 to n − 1 do
2:     y[i] = 0
3:     for j = 0 to m − 1 do
4:         y[i] = y[i] + A[i, j] ∗ x[j]
5:     end for
6: end for
```

**Algoritmo 4** Multiplicação de uma matriz por um vector denso usando o formato CRS

```
1: for i = 0 to n − 1 do
2:     y[i] = 0
3:     for j = row_pointer[i] to row_pointer[i + 1] − 1 do
4:         y[i] = y[i] + values[j] ∗ x[column_index[j]]
5:     end for
6: end for
```

processamento admitindo que na matriz esparsa, o número de elementos não nulos seja inferior a metade dos elementos da matriz densa, sendo por isso o número de operações necessárias neste caso, inferior à multiplicação de uma matriz densa ($2n^2$) [12].

#### 4.2.1.2 Block Compressed Row Store

A implementação do CNC [12] usa uma variante no formato das matrizes para retirar maior proveito dos mecanismos de funcionamento GPU. Segundo os autores, a implementação usando o formato *Block Compressed Row Storage* (BCRS) também obtém melhores desempenhos em CPU.

A melhor eficácia deste método é justificada pelo menor número de acessos a memória, quando comparado ao CRS, e por uma organização de dados que irá potenciar uma melhor utilização de cache e registos de processadores vectoriais. Este formato é em grande parte semelhante ao CRS com o pormenor de armazenar blocos da matriz original em vez de valores isolados desta. Neste formato cada bloco é composto pelos elementos daquela zona da matriz, contendo pelo menos um valor não nulo. Desta forma podemos considerar o CRS como um caso específico do BCRS em que os blocos têm tamanho igual a $1x1$.

Assim, a estrutura irá potenciar o uso de memória local presente nos SPE's ou mesmo a cache de um processador convencional para armazenar integralmente todo o conteúdo do bloco e efectuar a multiplicação dessa zona sem aceder a memória central. Este funcionamento irá evitar acessos redundantes que, por sua vez, diminuirá a latência observada. Outro factor importante é a contiguidade em memória dos dados correspondentes a dados de entradas

vizinhas na matriz, que permitirá o uso do modelo SIMD disponível nos SPE's. Este formato irá também reduzir o tamanho das tabelas de direccionamento utilizadas, sendo essa diferença mais notada para blocos de maiores dimensões.

No que diz respeito à multiplicação de matrizes por vectores, quando se processa um produto de um bloco com mais que uma linha, é possível ler com um único acesso os valores correspondentes do vector e guarda-los em registos para uso posterior aquando do processamento de cada linha do bloco da matriz.

A figura 4.2 ilustra como o número de acessos a memória é dependente do tamanho dos blocos utilizados.

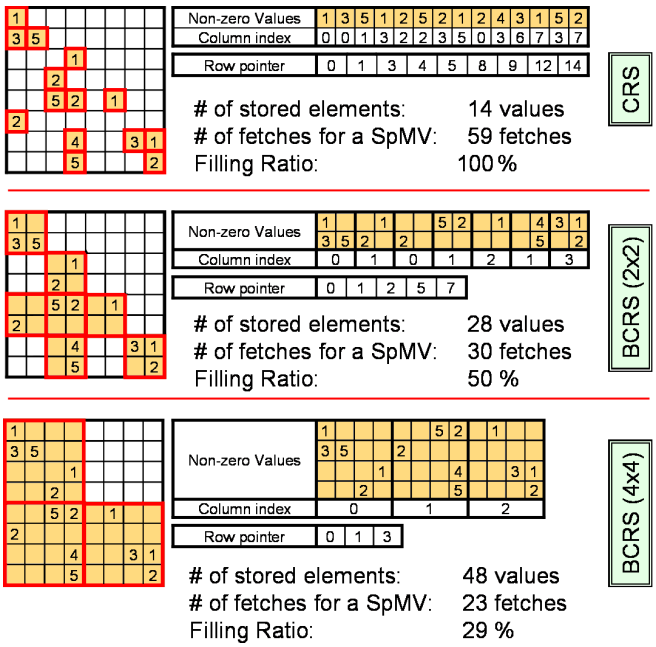


**Figura 4.2** Exemplos de Block Compressed Row Storage (BCRS) para a mesma matriz.

Apesar do facto dos blocos maiores causarem um maior aproveitamento das memórias locais ou cache de processador, tal só é aconselhado para matrizes com maior densidade.

O CNC implementa o formato BCRS $4x4$ e $2x2$ para casos onde blocos $4x4$ apresentarem uma percentagem de preenchimento muito baixo.

O uso desta biblioteca ou de um método equivalente pode tornar-se vital na aceleração do simulador desenvolvido em [10]. Como já foi referido, o simulador implementado faz uso de uma variante do método dos gradientes conjugados pré-condicionado. Uma vez que este método é composto, entre outras rotinas, por multiplicações de matrizes esparsas por vectores torna-se conveniente permitir a execução desta rotina de uma forma paralelizável.

Assim, o uso de formatos como o BCRS pode optimizar o tempo de execução e permitir uma pequena redução na quantidade de memória requerida para armazenar matrizes. Este facto revela-se de maior importância quando temos em conta factores como a reduzida quantidade de memória presente nos SPE's de um processador Cell. Este formato permite ainda uma fácil distinção de linhas na matriz permitindo assim uma divisão rápida e eficaz de processamento de forma a permitir a paralelização. Este tema será abordado com mais pormenor no capítulo 5 tendo em conta as características dos modelos de programação orientados para o desenvolvimento de aplicações na arquitectura Cell.

# 5 . Modelo de Paralelização da Animação de Superfícies Deformáveis

Como foi já referido, o simulador desenvolvido em [10] pode ser dividido em três etapas cíclicas ao longo do tempo. são elas:

**Evaluation** Fase de avaliação do modelo.

**Solve** Construção das matrizes das derivadas parciais e resolução do sistema de equações.

**DLOD** Fase da variação ao nível do detalhe.

No total de três fases que compõem a simulação, foi identificado a fase *Solve* como sendo a fase predominante no que toca a tempo de processamento [18]. Como descrito no capítulo 2, nesta fase é resolvido o sistema de equações gerada na fase *Evaluation* através do método dos gradientes conjugados pré-condicionado modificado apresentado na secção 2.3. Em [18] são ignoradas para fins de aceleração as restantes fases uma vez que estas representam pouco tempo de processamento, sendo assim menos notórios os ganhos caso fossem alvos de optimização.

## 5.1 Paralelização do Método dos Gradientes Conjugados

Depois da identificação das operações envolvidas no método dos gradientes conjugados pré-condicionado modificado implementado por Fernando Birra, podemos prosseguir à sua análise de forma a avaliar a possibilidade de paralelização do algoritmo. O cálculo envolvido na resolução do método dos gradientes conjugados majora assim o tempo de execução de toda a modelação desenvolvida [18]. Torna-se assim desejável a aceleração deste cálculo.

Como já foi referido, o método dos gradientes conjugados pré-condicionado modificado é composto essencialmente por:

- Soma e subtracção de vectores;
- Multiplicação de vectores por escalares;
- Produtos internos entre dois vectores;
- Multiplicação de matrizes esparsas por vectores densos.

A soma e subtracção de vectores, são operações facilmente paralelizáveis. Isto deve-se à possibilidade do processamento de cada índice do vector de forma independente. Para além de ser possível fragmentar os vectores e dividir o processamento por todas as unidades, a operação pode ainda gozar do modelo SIMD disponível em todos os SPE's.

A multiplicação de vectores por escalares, tal como a soma ou subtracção de vectores, goza também da propriedade de ser facilmente paralelizável.

O cálculo de produtos internos é também uma operação paralelizável uma vez que é constituído por uma soma de produtos de valores independentes. No entanto, para o cálculo desta operação terá de ser efectuada uma redução de todos os resultados parciais.

A multiplicação de matrizes esparsas por vectores densos pode ser paralelizada, por exemplo, dividindo as linhas da matriz. No fundo uma operação deste tipo pode ser dividida num número de produtos internos igual à dimensão do vector a multiplicar. Na secção 4.2 foi discutido o BCRS. O uso deste formato proporciona a possibilidade de tirar partido da natureza SIMD dos SPE's e ainda permite a fragmentação da matriz pelas várias unidades de processamento.

Estamos, então, perante vários problemas de natureza embaraçosamente paralelizável uma vez que todas as operações dependem unicamente de valores locais e não é exigida a comunicação entre os processos.

## 5.2 Aceleração GPU de Animação de Superfícies Deformáveis

Durante o ano de 2009, João Rocha [18] teve como principal objectivo da sua dissertação a aceleração do simulador de superfícies deformáveis desenvolvido por Fernando Birra [10] usando o poder de processamento oferecido pelos GPU's modernos através do modelo de programação oferecido pelo CUDA da NVIDIA. Assim, neste capítulo serão resumidas as técnicas utilizadas por Rocha e os resultados por ele obtidos. De relembrar que o simulador de superfícies deformáveis adoptado para discussão neste documento é o mesmo simulador adoptado por João Rocha para o desenvolvimento do seu trabalho.

Nesse trabalho procedeu-se à paralelização das rotinas envolvidas no método dos gradientes conjugados usado no simulador em causa e à sua programação em GPU. Para efeitos de comparação de desempenho o autor efectua *benchmarks* cujo objectivo é testar apenas uma das operações envolvidas no método dos gradientes conjugados. Concretamente, são testadas várias formas de efectuar operações como soma e subtracção de vectores, multiplicações por escalares ou produtos internos.

O autor efectua ainda um conjunto de testes finais envolvendo uma peça de tecido quadrado,

preso por duas pontas transversais a cair sobre uma esfera. Este teste é no entanto efectuado com diferentes números de partículas por forma a testar o desempenho do algoritmo com vários volumes de dados. A figura 5.1 apresenta os tempos de execução respeitantes à fase *Solve* do simulador obtidos por Rocha. Deve-se ainda referir que para os testes realizados o autor utilizou um processador Intel Core2Duo a 2.4GHz e uma placa gráfica NVIDEA GeForce 8800 GTS.

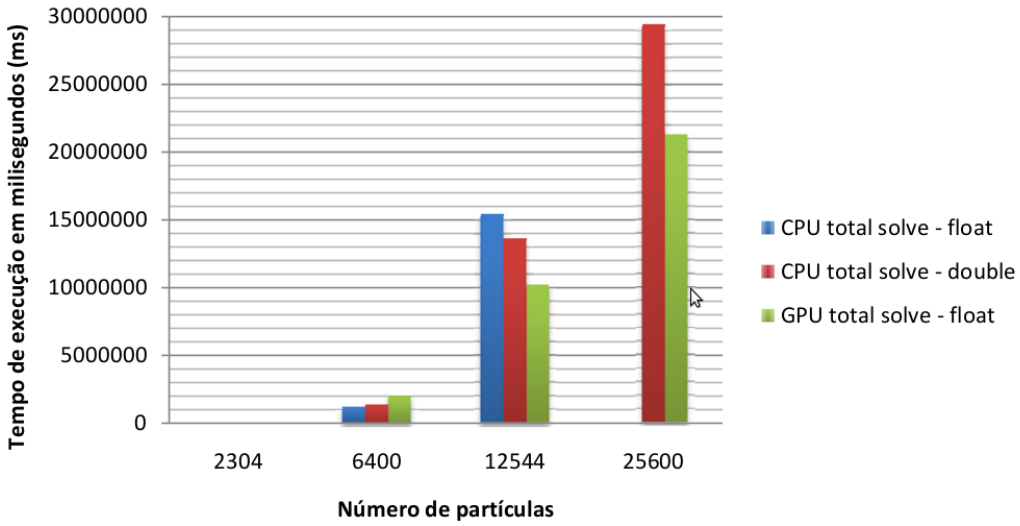


**Figura 5.1** Tempo de execução para o passo Solve usando CPU e GPU, para malhas com diferentes números de partículas

João Rocha justifica a baixa aceleração da fase *Solve* observada no gráfico com a falta de suporte de números de precisão dupla no GPU. Este facto influencia o número de operações necessárias para se observar a convergência dos valores aquando da execução do método dos gradientes conjugados. Assim, quando executado sobre GPU, o ciclo é executado entre 1.5 a 2 vezes mais até que o valor de $\Delta v$ seja determinado. A pouca aceleração é também justificada pelo facto da fase *Solve* ser também composta pela construção das matrizes envolvidas na resolução do sistema, não sendo esta rotina alvo de paralelização.

O autor refere ainda que os tempos de execução apontados na figura 5.1 são tempos que incluem o processamento envolvido na transformação das estruturas de dados e a sua transferência para o espaço de memória do GPU.

Na figura 5.2 são apresentados os tempos de execução de uma iteração do método dos gradientes conjugados pré-condicionado modificado. Torna-se aqui visível o *speedup* alcançado uma vez que os tempos referidos não incluem quaisquer *overheads* relacionados com a conversão das estruturas de dados e a sua transferência entre espaço de memória de GPU e memória central.

Assim, o maior *speedup* alcançado foi de 11$x$ para o teste mais complexo que foi possível realizar, com 36864 partículas.

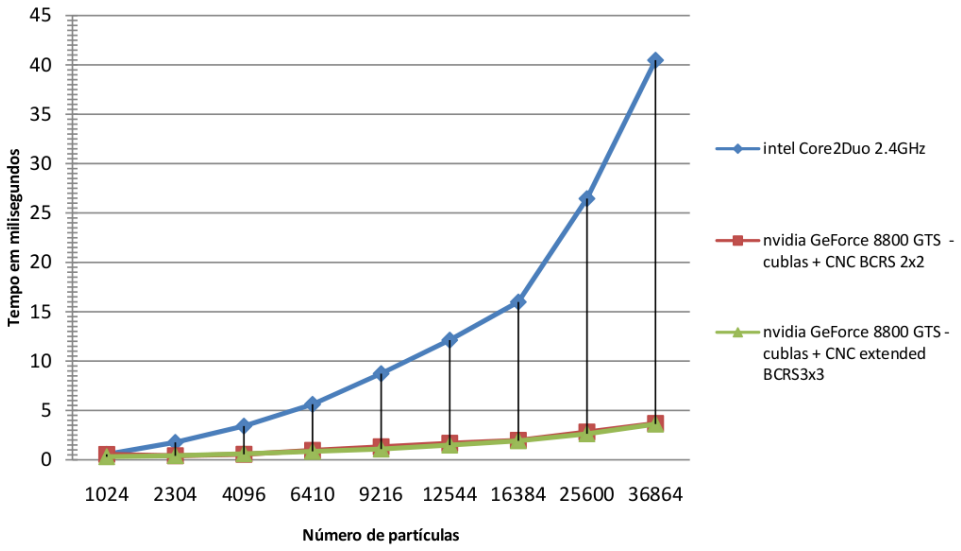


**Figura 5.2** Tempo de execução para o passo Solve usando CPU e GPU, para malhas com diferentes números de partículas

Concretamente, o trabalho desenvolvido pode resumir-se na implementação de rotinas com o objectivo de resolver as operações relacionadas com vectores, tais como as somas ou produtos internos e pela extensão do código do CNC [12] com o fim de suportar blocos de dimensões 3$x$3. Por omissão, o simulador, depois de sofrer as alterações efectuadas pelo autor do texto, recorre ao CUBLAS (implementação do BLAS sobre GPU) para efectuar as operações sobre vectores e à versão estendida do CNC para efectuar multiplicações matriz-vector.

## 5.3 Aceleração em Cell do Método dos Gradientes Conjugados

A arquitectura Cell/B.E. foi usada como suporte para permitir a aceleração da animação de superfícies deformáveis no contexto deste trabalho. Apesar desta arquitectura ser discutida mais pormenorizadamente no capítulo 3, torna-se importante referir alguns dos seus aspectos para avaliar a possibilidade de paralelização deste algoritmo.

Como foi referido anteriormente, a arquitectura Cell/B.E. é composta por um número positivo de PPU's e SPU's e um número positivo de SPE's orientados para o processamento Single Instruction Multiple Data (SIMD) (figura 5.3). Este facto permitiu-nos gozar de um sistema que oferece ao programador a possibilidade de efectuar processamento de rotinas distintas em diferentes SPE's e, simultaneamente usufruir de um modelo SIMD. Desta forma,

para além de permitir a paralelização de rotinas como soma de vectores através do modelo SIMD, a arquitectura Cell permite também a execução simultânea de várias tarefas distintas independentes.

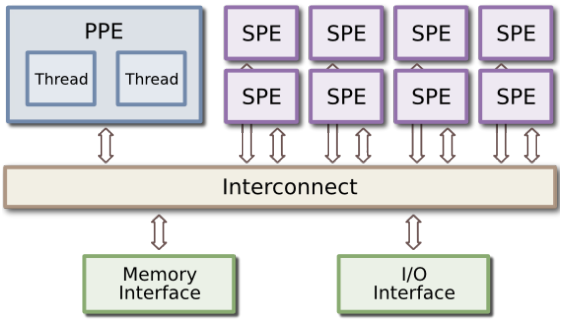


**Figura 5.3** Processador de Acordo com Cell Broadband Engine Architecture

Depois de analisados os mecanismos de funcionamento e a teoria adjacente ao simulador alvo de aceleração nesta dissertação podemos finalmente planear o método a seguir para tornar este objectivo possível.

### 5.3.1 Aceleração Através de Function Offload Model

Como foi referido anteriormente, a maior fracção de tempo de execução do simulador é tomado pelo método dos gradientes conjugados pré-condicionado modificado, sendo este composto por um conjunto de rotinas de operações entre vectores e multiplicações matriz-vector. Sendo estas as operações com um maior peso computacional, tendo em conta a sua natureza trivialmente paralelizável, podemos concretizar a sua aceleração recorrendo ao modelo *Function Offload Model*. Assim, substituindo as rotinas de operações vector-vector e matriz-vector por rotinas executadas em SPU, podemos proceder à paralelização do algoritmo retirando um grande proveito da arquitectura Cell/B.E. sem alterar lógica de funcionamento do método original.

Concretamente, neste modelo de programação, cada rotina que se pretenda executar tirando partido dos SPE's será substituída por um *wrapper*. Este pequeno método será encarregado de invocar o programa a ser executada no SPU e fornecer, sobre a forma de argumentos, os endereços em memória central das estruturas necessárias para efectuar o cálculo. Assim, o programa iniciado em SPE's será responsável pela cópia dos dados necessários para memória local, e após o processamento, pela transferência dos resultados para uma zona de memória central.

De realçar que o mecanismo descrito é apenas um exemplo onde todo o processamento é efectuado numa única SPE. É possível recorrer à utilização de vários SPE's com relativa

facilidade caso o problema seja trivialmente paralelizável. Assim, o processamento poderá ser dividido entre várias SPE's. Neste caso, o *wrapper* será responsável pela distribuição dos diferentes segmentos pelas diferentes SPE'S e caso necessário, pelo cálculo da redução do resultado, seguindo o modelo *Master-Worker*. Esta técnica permite-nos também, caso as operações não sejam trivialmente paralelizáveis, escalonar o cálculo de uma outra operação independente da primeira num outro conjunto de SPE's.

Este modelo pode ser facilmente adoptado para proceder à aceleração de qualquer rotina contida no método dos gradientes conjugados presente no simulador desde que os ganhos no tempo de processamento sejam elevados acabando por compensar os tempos de transferência dos dados relativos a cada operação.

### 5.3.2 Aceleração Através de Streaming Model

No *Streaming Model*, os SPE's são organizados segundo um *pipeline*, enviando os resultados de uma dada computação para alimentar a computação de um outro SPE. Assim, seguindo este modelo, as SPU's estarão organizadas sob a forma de um grafo acíclico, em que os fluxos de informação seguem os sentidos dos arcos.

Um exemplo bastante ingénuo, mas ilustrativo pode ser descrito por uma computação sobre três *arrays* composta por duas operações ($(\mathbf{a}+\mathbf{b})*\mathbf{c}$) em que os operadores $+$ e $*$ representam respectivamente a soma e multiplicação dos valores contidos no mesmo índice dos *arrays*. Nesta operação, poderemos organizar uma *pipeline* composta por dois SPE's.

Assim, o primeiro SPE na *pipeline* seria responsável por iniciar a transferência de dados relativos a **a** e **b** para espaço de memória local, efectuando então o cálculo da soma dos elementos dos vectores. Os dados relativos ao resultado da computação seriam então transferidos da memória local relativa ao primeiro SPE para o segundo na *pipeline*. Este segundo SPE passaria por iniciar a transferência dos dados relativos a **c** e efectuar a operação de multiplicação, transferindo o resultado final para memória central.

De salientar que o processo descrito é um processo cíclico e tem como objectivo a contínua sustentação do fluxo de dados entre os SPE's presentes na *pipeline*.

O mecanismo descrito pode acabar por obter melhores resultados a nível de desempenho, quando comparados com o *Function Offload Model* , uma vez que a transferência de dados entre memórias associadas a SPE's possui uma menor latência que a transferência de dados lidos ou escritos na memória central numa ordem de magnitude.

Neste pequeno exemplo, sendo $x$ o tamanho em bytes de todos os *arrays*, foram efectuadas transferências de memória que perfazem o total de $5x$ bytes, sendo que $2x$ foram efectuados entre duas unidades SPE.

Para a mesma computação, usando o *Function Offload Model*, os dados transferidos perfariam um total de 6$x$ sendo que todos eles teria tido origem ou destino em memória central, sendo assim caracterizados com uma maior latência.

Este modelo de programação poderia ser adoptado para efectuar determinadas computações no método dos gradientes conjugados adoptado por Birra, no seu simulador. Tendo em atenção o algoritmo 2, listado no capítulo 2, podemos observar a existência de diferentes grupos de rotinas paralelizáveis cujo resultado de uma é usado como argumento de outras. Tais casos podem ser observados nas linhas 17,18($\mathbf{q} = \mathbf{SAc}$ ; $\alpha = \frac{\delta}{\mathbf{c}^T\mathbf{q}}$) e 21,23($\mathbf{s} = \mathbf{P}^{-1}\mathbf{r}$ ; $\delta = \mathbf{r}^T\mathbf{s}$) onde a primeira instrução efectua uma multiplicação matriz-vector e a segunda efectua um produto interno entre o resultado e outro vector.

## 5.4 A Utilização do Formato BCRS

Na secção 4.2 do capítulo 4 deste documento foi introduzido o formato BCRS utilizado pelos autores de [12] afim de implementar uma biblioteca para GPU com capacidade de efectuar multiplicações matriz-vector. Este formato tem entre outros objectivos a compressão de matrizes esparsas afim de requererem menos memória e tempo de transmissão entre espaço de CPU e GPU. Este formato permite também que a complexidade desta operação desça de $O(n^2)$ (em que $n$ representa a dimensão da matriz) para $O(n)$ (proporcional ao número de elementos não nulos da matriz).

Podemos ainda referir que o processamento da matriz bloco a bloco irá diminuir o número de *fetchs* necessários para obter os valores necessários do vector argumento desta operação. Dá-se o acontecimento deste fenómeno uma vez que cada linha do bloco será multiplicada pela mesma fracção do vector, permitindo assim a armazenagem desse pedaço em registos para posterior uso.

De realçar ainda que, caso o processamento de um conjunto de blocos *nxn* pertencentes à mesma linha da matriz sejam atribuídos ao mesmo SPE, o resultado da totalidade do processamento irá ser armazenado num bloco *nx*1, reduzindo assim a quantidade de memória necessária e adiando a sua transferência até ao fim do processamento do conjunto dos blocos.

Sendo assim, a aceleração da operação de multiplicação por matrizes será efectuada fragmentando os blocos em conjuntos representativos das linhas segundo o modelo *Function Offload*. A cada SPE livre será então atribuído ciclicamente uma linha da matriz, sendo depois retornada o fragmento calculado da solução, até que toda a matriz seja processada.

Para além da paralelização do método através da divisão do problema pelo conjunto de SPE's disponíveis, o processamento inerente a cada linha da matriz pôde ainda ser paralelizado devido à utilização do modelo de programação *Single Instruction Multiple Data* disponível em

cada SPU. Na secção 5.7 serão efectuadas algumas abordagens a como proceder à paralelização nos SPU's utilizando este modelo.

## 5.5 Utilização de Double-buffering e Multibuffering

Devido à reduzida capacidade de armazenamento local a cada SPE, o processamento de grandes volumes de dados é forçosamente efectuado em blocos de tamanho suficientemente pequenos de forma a permitir a reunião de todos os dados necessários para uma determinada operação na memória dos SPE's. Este facto vai provocar um repetitivo acesso a memória central de forma a obter as pequenas porções de operandos.

O processamento de grandes volumes de dados é então, como ilustrado na figura 5.4], realizado num ciclo onde geralmente, a cada iteração, é realizado um acesso a memória para obter os operandos, o processamento destes e um novo acesso de forma a transferir os resultados de volta.

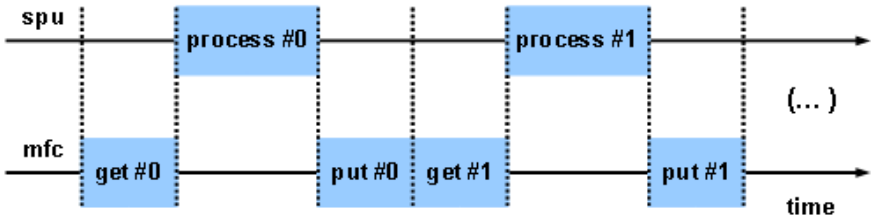


**Figura 5.4** Generalização do Fluxo de Execução e Transferências de Memória num SPU

Uma vez que as transferências de memória é efectuada apenas pelo MFC do SPE correspondente, não necessitando do processador, o problema deste algoritmo prende-se na perda de tempo de processamento e na taxa de utilização do *bus* interno, atingindo assim baixos valores para ambos.

### 5.5.1 Double-buffering

O *Double-buffering* é uma técnica que pode ser implementada com vista a atingir um maior desempenho. O seu conceito base explora a capacidade para processar um determinado volume de dados paralelamente à transferência do bloco seguinte e, em alguns casos, à transferência dos resultados do bloco anterior para memória central.

Seguindo o exemplo acima mencionado, caso seja possível determinar previamente os operandos relativos a uma determinada iteração, a transferência desses dados pode ser

executada paralelamente pelo MFC aquando do processamento do bloco anterior, tornando assim possível atingir valores da taxa de ocupação de SPU e MFC mais elevados.

Mais concretamente, o algoritmo associado a esta técnica pode ser descrito da seguinte forma:

1. O SPU agenda uma operação DMA GET para transferir uma porção dos dados do problema para o buffer #1.
2. O SPU agenda uma operação DMA GET para transferir uma porção dos dados do problema para o buffer #2.
3. O SPU aguarda que o buffer #1 acabe de ser preenchido.
4. O SPU processa o buffer #1.
5. O SPU agenda um DMA PUT para transferir os resultados contidos no buffer #1 para memória central e um DMA GET para obter dados de forma a voltar a preencher o buffer.
6. O SPU aguarda que o buffer #2 acabe de ser preenchido.
7. O SPU processa o buffer #2.
8. O SPU agenda um DMA PUT para transferir os resultados contidos no buffer #2 para memória central e um DMA GET para obter dados de forma a voltar a preencher o buffer.
9. O processo é repetido desde o passo 3 caso existam mais dados para serem processados.
10. O processo termina quando for efectuado o processamento de todos os blocos e todas as transferências de memória tiverem terminado.

Na figura 5.5 é ilustrado o entreleçamento no tempo entre as transferências de memória e o processamento dos blocos seguindo uma variante deste algoritmo onde se processa não duas, mas apenas uma porção em cada iteração. Nesta abordagem, antes do processamento de cada bloco, é agendado um DMA GET dos operandos do ciclo seguinte. Paralelamente ao processamento são também transferidos os resultados da interação anterior para memória central.

O funcionamento interno do MFC levanta um problema no passo 5 e 8 do algoritmo descrito. Internamente, o MFC, quando encarregue de mais que uma transferência, tenta ordenar estas de forma a obter o melhor desempenho possível. Por exemplo, se o MFC estiver encarregue de duas transferências e apenas uma das zonas se encontrar em memória principal (encontrando-se a outra em *swap*), o MFC encarregar-se-á primeiro da zona disponível.

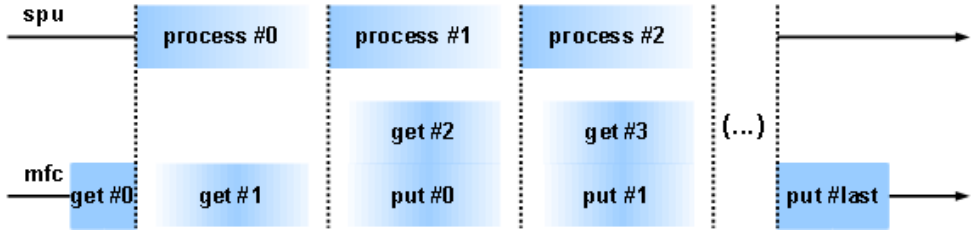


**Figura 5.5** Generalização do Fluxo de Execução e Transferências de Memória num SPU usando *double-buffer*

Desta forma será importante garantir que os resultados que se encontram no *buffer* sejam transmitidos para memória central antes que a transmissão dos operandos seja iniciada. Como foi já discutido no capítulo 3, no caso concreto do Cell/B.E. esta garantia pode ser assegurada através da utilização dos comandos DMA *GETF*, *PUTF*, *GETB* e *PUTB*.

Este tipo de abordagem, onde o trabalho é dividido em fases e o seu tratamento é sobreposto por forma a aumentar o throughput pode também ser denominado de software pipelining. Neste caso, devido ao procedimento ser dividido em duas fases (*get*/*put* e processamento), o algoritmo pode ser descrito como *two-stage pipeline*.

### 5.5.2 Multibuffering

O processamento paralelo de dados aquando da sua transferência pode ser processado de variadas formas. Para além do convencional *double-buffering* pode também ser usado *multibuffering*. A utilização desta técnica permite ao programador explorar a utilização de múltiplos *buffers* e transferências simultâneas.

Concretamente, são agendados um conjunto de transferências de memória para múltiplos *buffers*. O MFC fica então encarregue de transferir as várias porções de operandos para a memória local de cada SPE. Esta transferência é efectuada por uma determinada ordem (estimada pelo MFC como sendo a melhor ordem) sujeita a factores como a localização física dos dados (*RAM* ou *SWAP*). Os dados são então processados pelo SPU aquando da finalização da trasnferência e um novo comando DMA GET é agendado de forma a preencher novamente o buffer processado.

O funcionamento desta técnica pode ser descrito da seguinte forma:

1. Agendar um DMA GET para todos os buffers disponíveis. Identificar cada comando com uma DMA tag ID única.

2. Se todos os buffers foram processados e não se encontrarem no alvo de nenhum comando

DMA GET, esperar que todas os comandos DMA PUT concluam e sair.

3. Esperar apenas que um dos buffers se encontre cheio.
4. Processar o buffer.
5. Agendar um DMA PUT para transferir os resultados de volta para memória central.
6. Agendar um DMA GETB para preencher novamente o buffer após os dados nele contidos serem descarregados.
7. Voltar ao passo 2.

A utilização desta técnica contribui para uma utilização mais uniforme do BUS interno do processador e oferece alguma margem de manobra ao MFC para efectuar as transferências na melhor ordem possível com o objectivo de alcançar um melhor desempenho. De salientar também que, em caso de sobrecarga do BUS interno, o processamento pode avançar sucessivamente para o próximo buffer sem exigir que todas as trasnferências sejam finalizadas.

## 5.6 O Modelo de Programação *Master-Worker*

O modelo de programação *master-worker* é geralmente composto por dois tipos de entidades. Em grande parte das implementações, este modelo é traduzido numa interacção entre um processo/*thread*/nó denominado *master* e um conjunto de várias entidades com capacidade de processamento denominadas *workers*. Assim, o *master* é responsabilizado por iniciar a computação e criar uma carteira de tarefas para distribuir por todos os *workers*. Os *workers* são simples entidades que se limitam a receber dados relativos a um determinado problema e, após efectuarem o processamento necessário, entregar os resultados calculados ao *master*.

Neste padrão de programação distinguem-se duas abordagens no que toca à distribuição de trabalho. A atribuição estática de trabalho baseia-se na divisão do trabalho a priori, distribuindo assim porções de igual tamanho a cada *worker*. A atribuição dinâmica baseia-se no princípio que as várias porções de trabalho são caracterizadas por distintas cargas de computação. Desta forma, as tarefas são divididas em fracções de trabalho mais reduzidas de forma a poderem ser distribuídas dinamicamente ao longo do tempo pelos vários *workers* de forma a atingir um maior equilíbrio na distribuição de trabalho.

De salientar que, de forma a permitir a utilização deste modelo, a operação em causa deve ser preparada para que não se dê a necessidade de utilização dos dados computados numa outra entidade. Assim, o processamento dos dados em cada nó deve ser efectuado de forma independente, não exigindo assim sincronização e troca de valores entre *workers*.

É de relativa facilidade estabelecer uma ponte entre os componentes descritos no modelo *master-worker* e as entidades presentes no processador Cell. O PPU oferece, como unidade de processamento convencional, um óptimo suporte para efectuar o trabalho destinado a um *master*. A capacidade de processamento vectorial presente nos SPU's e a elevada largura de banda a estes oferecida no acesso a memória central torna-os óptimos candidatos a *workers*.

## 5.7 Paralelização de Processamento Usando o Modelo SIMD em SPE's

Como foi referido no capítulo 3, os SPE's presentes na arquitectura Cell/B.E. são compostos, entre outros elementos, por uma unidade de processamento orientada para o modelo *Single Instruction Multiple Data*.

Nestas unidades de processamento da arquitectura Cell/B.E. encontra-se um processador vectorial nativo com uma capacidade de 128 bits. Assim, é permitido ao programador a utilização deste registo para o processamento de um a dezasseis elementos, dependendo do tipo de dados pretendidos.

Aquando da programação para SPU, em C, é fornecido ao utilizador uma API para facilitar o processamento deste tipo de dados. A biblioteca *spu_intrinsics.h* contém rotinas de processamento de vectores a nível aritmético, lógico e orientadas para o byte.

A utilização deste modelo pode ainda dar-se de forma implícita uma vez que o compilador, ao reconhecer alguns padrões de programação, efectua alterações pontuais, como *loop unrolling*, de forma a que o programa gerado goze do modelo oferecido.

Esta optimização, automática ou não, é apenas permitida desde que os valores dos operandos não dependa de chamadas a funções nem de outros tipos de dados. É necessário ainda que os operandos se encontrem contíguos em memória. De referir que nem todas as operações se encontram disponíveis para todos os tipos de dados.

A utilização deste modelo poderá tornar-se numa medida muito favorável na aceleração das operações do método dos gradientes conjugados. As recorrentes operações sobre vectores, tais como as somas ou produtos internos, podem ser assim facilmente optimizadas. No que toca às operações de multiplicação matriz-vector, usando o formato BCRS, este modelo poderá também ser vantajoso na medida em que todas as operações envolvidas na multiplicação dos blocos por vectores podem tirar proveito de processamento vectorial.

# 6 . Implementação

Nos capítulos anteriores foram discutidas as capacidades da arquitectura Cell/B.E., o método dos gradientes conjugados pré-condicionado modificado, que consome a maior parte do tempo de processamento aquando da execução do simulador e as operações por este efectuadas, assim como algumas tecnologias relacionadas com este tipo de processamento.

Desta forma encontramo-nos em condições de descrever os vários detalhes de implementação e proceder à sua justificação. Neste capítulo, serão tratadas as particularidades relacionadas concretamente com a implementação tais como o modelo de programação utilizado, a descrição das estruturas de dados e a apresentação das várias operações envolvidas no algoritmo alvo de aceleração. Serão também expostos os tempos de execução das várias operações implementadas, assim como o *speedup* atingido usando as capacidades do processador Cell.

## 6.1 Modelo de Programação

No capítulo 5 deste documento foi discutido a possibilidade da utilização de duas técnicas de programação de forma a atingir um melhor desempenho do simulador em causa. As técnicas descritas estavam intimamente relacionadas com a arquitectura Cell/B.E. e discriminavam a organização tomada pelos vários componentes do processador Cell e o fluxo de dados entre estes.

Como descrito, o *function-offload-model* era caracterizado por permitir uma maior velocidade de desenvolvimento e, simultâneamente, não forçar uma abordagem muito invasiva no projecto alvo de aceleração. Este modelo oferecia também um elevado aproveitamento das capacidades da arquitectura Cell/B.E.

Considerando a natureza do problema alvo de aceleração, podemos afirmar que, como descrito na secção 5.1, todas as rotinas processadas aquando da execução do método dos gradientes conjugados pré-condicionado modificado pertencem ao grupo de problemas trivialmente paralelizáveis. Assim, tendo em conta as especificações do processador Cell, torna-se prático o desenvolvimento de uma solução usando o modelo *master-worker*.

Podemos então afirmar que a aplicação da técnica *function offload model* através de uma implementação em que ocorre uma distribuição de trabalho pelos vários SPE's segundo o modelo *master-worker* tem capacidade de oferecer resultados satisfatórios sem a necessidade de uma alteração violenta do simulador alvo de aceleração. Nesta secção serão discutidas as várias decisões tomadas no desenvolvimento da solução ao problema proposto.

Para permitir a um processo usufruir da capacidade de processamento oferecida por um SPU é necessário proceder previamente ao seu carregamento com o programa desejado. Concretamente, após o carregamento do programa pretendido em espaço SPU, este é iniciado pela chamada de um método bloqueante. Assim, a inicialização de um SPE encontra-se intimamente ligada com uma sequência de operações como a criação de uma *thread*, carregamento do programa desejado em espaço SPE e o seu inicio de processamento. De notar que o conjunto destas operações acaba por provocar um *overhead* no tempo de execução. Uma outra solução passava pela criação de um conjunto de threads SPE por cada programa necessário ao processamento e usufruir do mecanismo de troca de contexto em SPE's. De notar que também este procedimento vem introduzir algum *overhead* no tempo de processamento verificado.

Podemos então confirmar que se torna vantajoso o desenvolvimento de um único programa com suporte para todas as operações pretendidas de forma a eliminar o *overhead* causado durante a execução. O programa seria então carregado em todos os SPE's no inicio da execução evitando assim sucessivos carregamentos que afectariam o *speedup* alcançado. Este procedimento vem introduzir a necessidade de forçar a sincronização entre o PPU e os SPE's uma vez que, seguindo esta abordagem, o tempo de vida dos programas para SPE crescem praticamente até à totalidade do tempo de simulação, obrigando à comunicação entre PPU e SPE em vez da tradicional passagem de parâmetros aquando da inicialização do programa em SPE.

Uma possível solução para o problema que enfrentamos passa pelo desenvolvimento de um mecanismo de sincronização e comunicação entre os vários componentes do processador Cell de forma a permitir uma eficaz distribuição de trabalho segundo o modelo *master-worker*. Na figura 6.1 pode ser observado uma esquematização do fluxo de execução presente na solução desenvolvida. O procedimento descrito pode dividir-se claramente em três fases facilmente identificáveis.

Na primeira fase o PPU é encarregue da criação de um número de *threads* igual ao número de SPU's disponíveis no processador em causa. As *threads* procederão de forma a carregar o programa desenvolvido para SPU em cada um dos SPE's e pela sua inicialização. O PPU aguarda então que todos os SPE's se encontrem preparados antes de prosseguir a execução. No solução adoptada, este procedimento ocorre aquando da inicialização do objecto *MPCGSolver* responsável por integrar o sistema de partículas no tempo.

Na segunda fase são processados as operações implementadas. O PPU divide o trabalho e prepara os argumentos de cada SPE numa zona de memória cujo endereço vai ser enviado ao SPE de forma a permitir o tratamento desses dados. Paralelamente, os SPU's ficam então encarregues de aguardar uma mensagem originada no PPU contendo os dados relativos à próxima operação a ser executada. Após o termino da execução por parte do SPU é enviada uma mensagem ao PPU indicando o fim do processamento por ele requisitado. Este

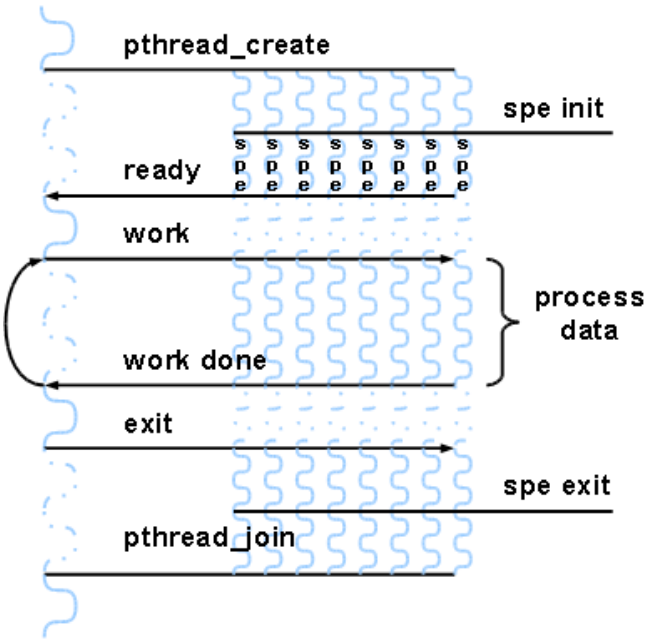


**Figura 6.1** Fluxo de execução e sincronização entre o PPU e os SPU's

procedimento é repetido por cada operação requisitada pelo PPU ou, mais concretamente, por cada rotina presente no método dos gradientes conjugados implementado.

A terceira fase é composta apenas por uma mensagem enviada por parte do PPU ordenando a cada SPU o fim da execução. O PPU aguarda então o termino de cada *thread* lançada na primeira fase e prossegue a execução.

De salientar que o procedimento implementado não elimina por completo o *overhead* causado no processamento das rotinas pretendidas. O tempo de processamento levado pelo PPU aquando da divisão do trabalho e a latência inerente à passagem de mensagens provoca ainda um acréscimo no tempo de processamento tomado. Testes executados com o propósito de medir este *overhead* apontam para um valor médio de 0.165 milissegundos por operação realizada.

## 6.2 Estruturas de Dados

De forma a tornar possível a aceleração do simulador desenvolvido em [10] torna-se necessário efectuar a transformação das estruturas de dados por este utilizadas em estruturas que minimizem os tempos de transferência e processamento em SPE's. Estas novas estruturas encontrar-se-ão criadas em memória e serão utilizadas durante a execução do método dos

gradientes conjugados pré-condicionado modificado.

Aquando do termo da execução do método dos gradientes conjugados pré-condicionado modificado, as estruturas originais adoptam os valores resultantes do processamento de forma a permitir a continuação da simulação, nomeadamente da fracção não acelerada.

### 6.2.1 Estruturas de Dados Originais

A fase denominada de *solve*, em que se divide o simulador alvo de aceleração nesta dissertação, é constituída pela construção das matrizes das derivadas parciais e pela resolução do sistema de equações gerado pelo método implícito. No âmbito da resolução desse sistema, o simulador recorre a estruturas de dados para proceder ao armazenamento dos vectores e matrizes envolvidos no cálculo.

Ao analisarmos o método Solve da classe ModifiedPCGSolver podemos observar que este efectua maioritariamente operações sobre vectores e matrizes esparsas. Uma terceira estrutura de dados é usada para armazenar matrizes diagonais de forma a tornar possível a filtragem de resultados aquando da execução do método Solve.

Assim os valores como as forças, posições, velocidades e acelerações são armazenadas num vector, ilustrado na figura 6.2, em que os dados contidos na posição $n$ dizem respeito aos valores da partícula $n$. Uma vez que estes valores são compostos por três componentes, respeitantes às três dimensões do espaço, em cada posição do vector é armazenado o conjunto dessas componentes.

Relativamente às matrizes que definem a interacção entre as partículas, podemos verificar que são armazenadas em conjuntos de blocos onde cada conjunto representa uma linha. Concretamente, a estrutura de dados de uma matriz é composta por uma lista onde em cada posição são armazenados os dados relativos a uma linha. Cada linha é composta por um conjunto de pares respeitantes a cada bloco onde um elemento armazena o bloco da matriz propriamente dito e o segundo nos informa a que coluna corresponde. Acerca do bloco da matriz podemos apenas mencionar que se trata de um *array* bidimensional de valores reais.

De salientar que a simetria presente nas matrizes envolvidas no processamento influenciou o seu desenho, sendo apenas armazenada a diagonal superior desta.

Relativamente às matrizes diagonais usadas no processo de filtragem, são armazenadas num vector de blocos, correspondendo o bloco da posição $n$ do vector ao bloco que se encontra na posição $(n,n)$ da matriz representada.

A esquematização de cada uma destas estruturas de dados enunciadas pode ser observada na figura 6.2. Concretamente estão representados (1) os vectores de valores tridimensionais relativos a cada partícula, (2) as matrizes esparsas e (3) as matrizes diagonais usadas no

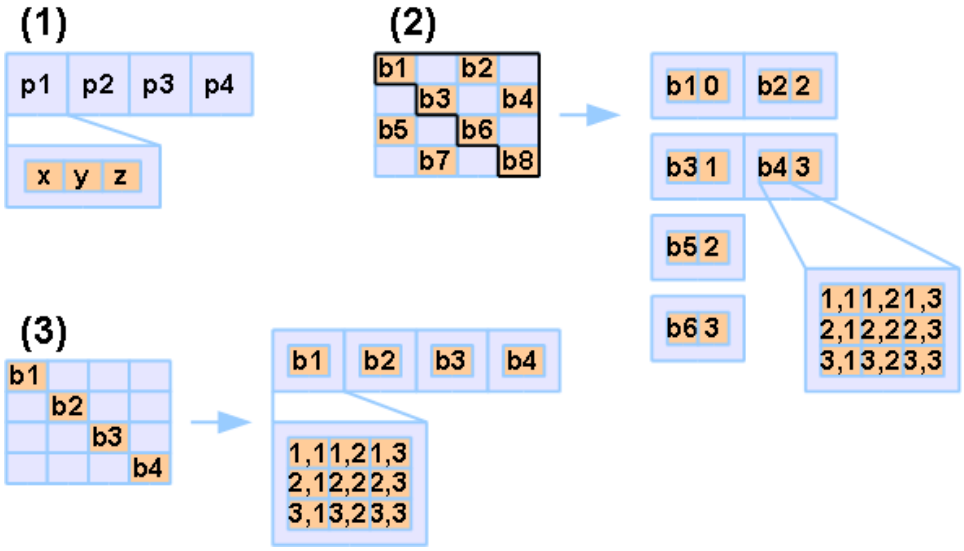


**Figura 6.2** Esquematização das estruturas de dados originais usadas no simulador

simulador.

Todos os valores armazenados nos vectores e nas matrizes são números reais que podem assumir a precisão de um *float* ou de um *double* em tempo de compilação.

### 6.2.2 Estruturas de Dados Utilizadas

Como foi referido anteriormente, cada SPE da arquitectura Cell/B.E. é composto, entre outros elementos, por uma memória local de pequena dimensão (256KB no *IBM BladeCenter QS21*, utilizado para o desenvolvimento desta dissertação), de um processador vectorial de 128 bits e de um MFC que processa transferências de memória de forma paralela ao processamento. Assim, o desenho das estruturas de dados a serem utilizadas será fortemente influenciado pelas características de todos estes elementos que compõem o processador CELL/B.E.

Desta forma as estruturas de dados deverão ser dotadas de características que permitem e favorecem a aceleração das rotinas alvo:

- O alinhamento de memória a dezasseis bytes é obrigatório devido à incapacidade por parte do MFC de efectuar transferências em que os endereços de origem e destino não consistam em múltiplos de dezasseis.
- O MFC força ainda o uso de *paddings* uma vez que poderá surgir a necessidade de se efectuar transferências de pequenas zonas de memória cujo valor será obrigatoriamente

um múltiplo de dezasseis.

- O uso de um processador vectorial, como é o caso do SPU, contribui também para o uso de *paddings* de forma a retirar deste um maior proveito.
- A pequena quantidade de memória local disponível em cada SPE sugere (1) a minimização do espaço ocupado pelas estruturas e (2) o cuidado na organização destas de forma a que todos os dados necessários para efectuar o processamento se consigam armazenar em memória local. De salientar ainda que a memória local é usada para armazenar não só os dados, como também as instruções do programa a ser executado.
- A contiguidade dos dados em memória torna-se também importante uma vez que uma única transferência atinge melhores desempenhos que um maior número de transferências para o mesmo volume de dados.
- Embora esteja mais relacionado com a natureza do problema, torna-se aconselhável a escolha de uma estrutura de dados que permita a divisão de trabalho segundo o modelo *master-worker* de forma a definir os operandos do processamento efectuado em cada SPE.

De seguida serão apresentadas as estruturas de dados utilizadas de forma a acelerar o processamento do método dos gradientes conjugados pré-condicionado apresentado no algoritmo 2.

Na versão acelerada do simulador de superfícies deformáveis, as estruturas de dados usadas resumem-se a uma representação de vectores tridimensionais e de matrizes esparsas.

Sendo os vectores de valores tridimensionais processados como se de um vector convencional se tratasse, no desenho das estruturas de dados desenvolvidas para o seu armazenamento, pode ser ignorado o facto de se traterem de grupos de valores. Assim sendo, os vectores poderão ser simplesmente representados numa secção contígua de memória alinhada a dezasseis bytes. No entanto, o processamento do resultado da operação de multiplicação de matrizes por vectores obriga a que os valores dos vectores sejam acompanhados de um *padding* de forma a que as várias partículas sejam armazenadas em blocos de tamanho múltiplo de dezasseis bytes. Este pormenor será justificado na secção 6.5. Na figura 6.3 pode observar-se a representação dos vectores utilizados na implementação realizada.

A estrutura de dados implementada para o armazenamento de matrizes foi fortemente inspirada no formato BCRS discutido na secção 4.2. Esta estrutura de dados pode ser descrita como um conjunto de pequenos *arrays* bidimensionais, representantes dos blocos não nulos da matriz original, acompanhados de meta informação que permite determinar a sua localização.

Concretamente, este formato é composto por dois *arrays*. O primeiro (chamemos-lhe *blocks*) é usado para armazenar os valores não nulos da matriz em pequenos blocos. O

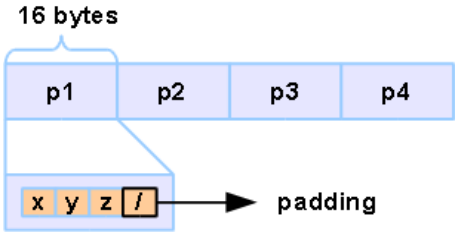


**Figura 6.3** Representação da Estrutura de Dados dos Vectores Usada na Aceleração do Simulador

segundo (chamemos-lhe *rows*) é simplesmente um índice do primeiro, armazenando apenas a informação da posição em que é iniciada cada linha da matriz em *blocks*.

Desta forma torna-se possível extrair informação acerca dos endereços e tamanho das zonas de memória onde estão armazenadas as diversas linhas, encontrando-se cada uma destas contígua em memória. A informação necessária para determinar a coluna correspondente de cada bloco é encontrada nos elementos armazenados no vector *blocks*, encontrando-se esta acompanhada do bloco de valores propriamente dito.

O bloco de valores é simplesmente um *array* de duas dimensões em que são armazenados os valores correspondentes. De salientar que nestes *arrays* é usado um *padding* por forma a que cada linha do bloco possua um tamanho múltiplo de dezasseis bytes. A utilização deste *padding* está relacionada com a utilização do *padding* em vectores e, como já foi enunciado, será justificada na secção 6.5.

Uma vez que as matrizes usadas no algoritmo se tratam de matrizes diagonais superiores, poder-se-ia retirar algum proveito dessa característica e armazenar apenas a porção da informação não redundante da matriz. Esta estratégia não foi seguida, sendo as duas triangulares armazenadas integralmente na estrutura de dados, forçando a utilização de informação redundante de forma a facilitar a operação de multiplicação de matrizes por vectores. A representação integral da matriz oferece-nos uma estrutura de dados onde todas as linhas se encontram contíguas em memória, não forçando cada SPE a efectuar múltiplos acessos de forma a transferir uma única linha.

Uma representação gráfica desta estrutura de dados pode ser observada na figura 6.4. Nesta figura é representada uma matriz esparsa de tamanho 4x4 em que metade dos blocos são compostos de valores nulos. Na figura podemos identificar os limites da linha 2 da matriz nas posições 2 e 3 do vector com os valores 4 e 6(exclusive) respectivamente. Assim podemos concluir que os blocos que compõem a linha 2 da matriz se econtram na posição 4 e 5 do vector de dados. Podemos observar os blocos *b5* e *b6* nessas posições acompanhados do valor da coluna onde se encontram.

De realçar que a conversão dos dados para os formatos pretendidos para a execução em Cell

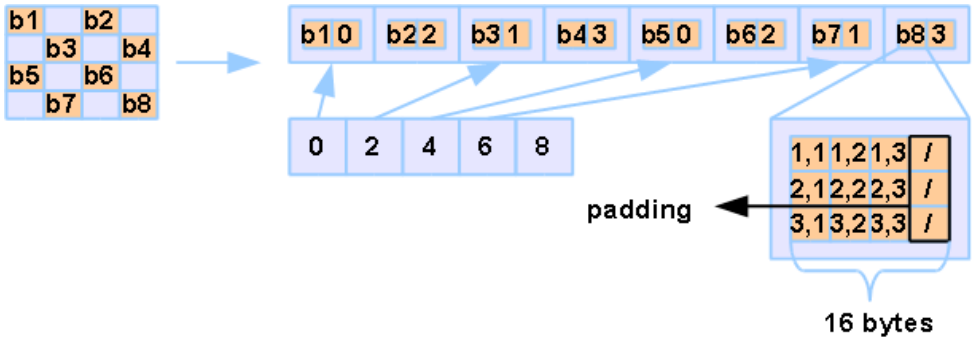


**Figura 6.4** Representação da Estrutura de Dados das Matrizes Usada na Aceleração do Simulador

e vice-versa provoca um *overhead* considerável no tempo de processamento do método dos gradientes conjugados pré-condicionado modificado implementado no simulador.

## 6.3 AXPY - Soma, Subtracção e Escalagem de Vectores

A soma de vectores é uma operação de complexidade temporal $O(n)$ em que $n$ é o número de elementos contidos nos vectores argumentos. Nesta operação, o processamento de dados resume-se ao cálculo de cada um dos elementos do vector de destino somando os valores das posições correspondentes dos vectores operandos. Assim, este algoritmo pode ser descrito simplesmente no algoritmo 5.

**Algoritmo 5** Soma de Dois Vectores

**for** $i = 0$ to $lenght$ **do**
  $out[i] \leftarrow in1[i] + in2[i]$
**end for**

Como podemos observar, trata-se de uma operação trivialmente paralelizável e pode ser facilmente acelerada usando os SPE's do processador Cell segundo o modelo *master-worker*.

Analogamente, as operações de subtracção e multiplicação por escalares em vectores gozam das mesmas propriedades que a soma de vectores. Encontramo-nos então perante três problemas bastante idênticos a nível de processamento.

Como foi já referido na secção 6.2, na memória local de cada SPE são armazenados os dados necessários para o processamento e as instruções do programa que se encontra em execução. Assim, é aconselhado alguma diligência no desenvolvimento do *kernel* de computação utilizado.

Uma vez que se tratam de três métodos suficientemente semelhantes, pôde-se desenhar uma única rotina com vista a processar diferentes operações que envolvem vectores. Desta forma a quantidade de memória necessária para o alojamento das instruções é reduzido, cedendo-a assim para o armazenamento de dados necessários à computação.

Uma outra vantagem, não menos importante que a mencionada, passa pela aglomeração de várias operações na mesma rotina de forma a poupar transferências de memória. Como exemplo podemos descrever uma sequência de operações sobre dois vectores composta pela multiplicação de um vector por um escalar e pela soma do vector resultante com o segundo vector. A aglomeração das rotinas efectuada irá potencialmente beneficiar a aceleração conseguida uma vez que o número de transferências entre memória central e local a cada SPE é reduzido.

Inspirando-nos no método *AXPY* suportado pelo BLAS [3], podemos proceder ao desenvolvimento de uma rotina generalista com capacidade de efectuar somas, subtracções e multiplicação por escalares em vectores. Concretamente, esta rotina efectua a multiplicação de um vector por um escalar e soma o resultado a um segundo vector, armazenando o resultado num terceiro.

Como foi já referido na secção 3.2, alguns dos compiladores disponibilizados para a arquitectura Cell/B.E. tiram proveito da instrução *madd* disponível nos SPU's e optimizam as operações do tipo $a + b * c$ de forma a que o cálculo seja efectuado num único ciclo de processador. Assim, para o caso específico em que o programador deseje apenas somar vectores, sendo forçado a escalar o primeiro vector pelo elemento neutro da multiplicação (1.0), o desempenho da operação não sai prejudicado uma vez que a multiplicação não adiciona nenhum *overhead* à operação.

A título de exemplo, ao analisar o método dos gradientes conjugados presente no simulador desenvolvido em [10], podemos observar três conjuntos de operações compostas por multiplicações de vectores por escalares e soma do resultado obtido a um segundo vector (listagem 6.1). O desenvolvimento de uma rotina dedicada à escalagem de vectores permitiu, seguindo o modelo master-worker, obrigar à execução de duas rotinas (a escalagem e a soma do resultado) e à consequente transferência de memória. Assim, desta forma, as várias operações presentes no código original do simulador foram substituídas por uma única rotina que é executada exclusivamente em SPU e não requer nenhuma transferência de memória adicional. A Listagem 6.2 exibe as instruções que substitiram os grupos de operações da listagem 6.1.

```
*alfac = *c;
*alfac *= alfa;
deltav += *alfac;

*q *= alfa;
*r -= *q;

*c *= beta;
*c += *s;
```

**Listing 6.1** Listagem de operações consecutivas com objectivo de somar um vector a ao resultado de uma escalagem

```
//deltav = deltav + (alfa * c)
axpy(alfa, c, deltav, deltav);

//r = r - (alfa * q)
axpy(-alfa, q, r, r);

//c = (beta * c) + s
axpy(beta, c, s, s);
```

**Listing 6.2** Listagem de operações que substituiram os conjuntos de operações listados em 6.1

Na listagem 6.3 encontram-se listados os cabeçalhos das funções em linguagem *C* desenvolvidas nesta dissertação dos métodos em espaço PPU e SPU.

```
void k_axpy(
  float scalar,
  float *in1, float *in2, float *out,
  long n);

void k_axpy(
  double scalar,
  double *in1, double *in2, double *out,
  long n);

void spu_axpy(
  float scalar,
  float *mm_in1, float *mm_in2, float *mm_out,
  unsigned long elements);

void spu_axpy(
  double scalar,
  double *mm_in1, double *mm_in2, double *mm_out,
  unsigned long elements);
```

**Listing 6.3** Cabeçalhos do Método *AXPY* em Espaço PPU e SPU

O PPU é responsável pela divisão do trabalho por todos os SPU's segundo o modelo *master-worker*. O controlo é então passado aos SPU's, sendo estes encarregados de efectuar a computação necessária sobre a sua porção do vector. Uma vez que o EIB (Element Interconnect Bus) encontrado no processador Cell não suporta transferências de memória superiores a 16KB, o processamento é efectuado em blocos do mesmo tamanho. Numa primeira aproximação, o código do SPU poderia ser descrita através do algoritmo apresentado em 6.

**Algoritmo 6** Transferência de Memória no Método *AXPY*

**for all** *block* in *blocks* **do**
  $local\_buffer\_in1 \leftarrow main\_buffer\_in1[block]$
  $local\_buffer\_in2 \leftarrow main\_buffer\_in2[block]$
  $wait(local\_buffer\_in1)$
  $wait(local\_buffer\_in2)$
  $local\_buffer\_out \leftarrow local\_buffer\_in1 * scalar + local\_buffer\_in2$
  $main\_buffer\_out[block] \leftarrow local\_buffer\_out$
  $wait(local\_buffer\_out)$
**end for**

Como descrito na secção 5.5, este tipo de algoritmo é caracterizado por uma fraca utilização de tempo de processamento e taxa de ocupação do *bus* interno. A aplicação da técnica de *double-buffer* ao algoritmo descrito resulta no procedimento exibido no algoritmo 7. Neste

algoritmo são usados dois *buffers* para cada vector e, idealmente, um deles é processado pelo SPU em paralelo com a transferência do outro por parte do MFC.

**Algoritmo 7** Transferência de Memória no Método *AXPY* Utilizando *Double-Buffer*

$local\_buffer\_in1[0] \leftarrow main\_buffer\_in1[0]$
$local\_buffer\_in2[0] \leftarrow main\_buffer\_in2[0]$
**for all** $blockinblocks$ **do**
  **if** $block+1 < total\_blocks$ **then**
5:  $local\_buffer\_in1[(block+1)\%2] \leftarrow main\_buffer\_in1[block+1]$
    $local\_buffer\_in2[(block+1)\%2] \leftarrow main\_buffer\_in2[block+1]$
  **end if**
  $wait(local\_buffer\_in1[block\%2])$
  $wait(local\_buffer\_in2[block\%2])$
10:  $wait(local\_buffer\_out[block\%2])$
  $local\_buffer\_out[block\%2] \leftarrow local\_buffer\_in1[block\%2] * scalar + local\_buffer\_in2[block\%2]$
  $main\_buffer\_out[block] \leftarrow local\_buffer\_out[block\%2]$
**end for**
$wait(local\_buffer\_out[0])$
15:  $wait(local\_buffer\_out[1])$

Depois de concluída a transferência da memória necessária ao cálculo de cada bloco é necessário proceder à computação dos dados em espaço SPE. Como descrito no capítulo 3, os SPUs são dotados da capacidade de processamento vectorial em registos de 16 bytes. A computação dos blocos propriamente ditos pode então ser efectuada tirando proveito desta capacidade.

Sem necessidade de entrar em grandes detalhes, podemos descrever o processamento vectorial destes blocos com base no código em linguagem C apresentado na listagem 6.4.

```
vec_float4 vscalar, *vin1, *vin2, *vout;

vscalar[0] = vscalar[1] = vscalar[2] = vscalar[3] = scalar;

vin1 = (vec_float4*) local_buffer_in1;
vin2 = (vec_float4*) local_buffer_in2;
vout = (vec_float4*) local_buffer_out;

for(i = 0; i < block_elements / 4; i++){
    vout[i] = vscalar * vin1[i] + vin2[i];
}
```

**Listing 6.4** Processamento Vectorial Associado ao Método *AXPY*

A utilização de *double-buffer* e o consequente aumento nas taxas de ocupação do SPU e do BUS interno do processador traduz-se num ganho de *speedup* na operação. Nas tabelas 6.1 e 6.2 podem ser consultados os tempos de execução da operação *axpy* quando computados no PPU (*PowerPC*) a 3.2GHz e nos SPU's do processador Cell disponível. Estas tabelas descriminam os tempos e *speedups* alcançados, quando comparados com o PPU, para o processamento de vectores de *floats* e *doubles*. Os valores apresentados são também discriminados para os dois algoritmos acima descritos. Na última coluna de ambas as tabelas encontra-se registado o ganho obtido com a utilização de *double-buffer* quando comparado com a execução em SPU sem a aplicação desta técnica.

Nas tabelas apresentadas podemos avaliar os tempos de execução da rotina *AXPY* observados. Podemos então salientar o crescente speedup acompanhado do aumento no número de elementos dos vectores argumento. Este facto justifica-se com o *overhead* constante presente em cada rotina que prefaz uma maior fracção em computações de pequenos vectores.

De relevar o *speedup* obtido nas operações sobre *floats* quando o número de elementos é dez milhões. Este número começa a estagnar em valores que rondam as quatro dezenas. Podemos também claramente comprovar as vantagens da abordagem que tira proveito do *double-buffering* devido ao constante decrescimo no tempo de execução aquando da utilização desta técnica. Uma pequena excepção a este facto ocorre na execução do método em causa para vectores com tamanho inferiror a 10000 elementos. Nestes casos, o volume dos dados em causa não proporciona o ganho de *speedup* quando comparadas as duas abordagens.

O baixo *speedup* da operação *axpy* para valores de precisão dupla, quando comparado com os mesmos valores para precisão simples, pode ser justificado com os tempos de execução das rotinas em PPU. Como podemos observar, na operação em causa, os tempos associados à soma e multiplicação de *doubles* em PPU é inferior ao valor correspondente nos *floats*.

De salientar ainda que os tempos resultantes da execução na arquitectura x86 são cerca de dez vezes menores que os resultados observados no PPU do processador Cell e cereca de três a

| **axpy - float** | x86 | PPU | SPU (sem *double-buffer*) | | SPU (com *double-buffer*) | | Δ% |
|---|---|---|---|---|---|---|---|
| elementos | tempo (ms) | tempo (ms) | tempo (ms) | *speedup* | tempo (ms) | *speedup* | |
| 1000 | 0,015 | 0,042 | 0,163 | 0,258 | 0,158 | 0,266 | 3,16% |
| 10000 | 0,147 | 0,388 | 0,167 | 2,323 | 0,171 | 2,269 | -2,34% |
| 100000 | 0,932 | 5,398 | 0,236 | 22,873 | 0,205 | 26,332 | 15,12% |
| 1000000 | 4,573 | 54,651 | 1,301 | 42,007 | 1,161 | 47,072 | 12,06% |
| 10000000 | 45,213 | 540,519 | 11,829 | 45,694 | 10,046 | 53,804 | 17,75% |

**Tabela 6.1** Tempos de execução em milissegundos e *speedup* alcançados para a execução da operação *axpy* processada em x86 (Intel Core2 Duo 2.66GHz), PPU (PowerPC 3.2GHz) e SPU para precisão simples

| **axpy - double** | x86 | PPU | SPU (sem *double-buffer*) | | SPU (com *double-buffer*) | | Δ% |
|---|---|---|---|---|---|---|---|
| elementos | tempo (ms) | tempo (ms) | tempo (ms) | *speedup* | tempo (ms) | *speedup* | |
| 1000 | 0,013 | 0,007 | 0,177 | 0,040 | 0,157 | 0,045 | 12,74% |
| 10000 | 0,125 | 0,086 | 0,208 | 0,413 | 0,181 | 0,475 | 14,92% |
| 100000 | 0,843 | 3,823 | 0,374 | 10,222 | 0,310 | 12,332 | 20,65% |
| 1000000 | 5,937 | 40,864 | 2,727 | 14,985 | 2,236 | 18,275 | 21,96% |
| 10000000 | 57,574 | 406,153 | 28,646 | 14,178 | 21,324 | 19,047 | 34,34% |

**Tabela 6.2** Tempos de execução em milissegundos e *speedup* alcançados para a execução da operação *axpy* processada em x86 (Intel Core2 Duo 2.66GHz), PPU (PowerPC 3.2GHz) e SPU para precisão dupla

quatro vezes maior que nos SPU's, para um número elevado de elementos.

## 6.4 DOT - Produto Interno de Vectores

O cálculo do produto interno entre vectores é, em semelhança à soma ou subtracção, uma operação de complexidade temporal $O(n)$ em que $n$ é o número de elementos contidos nos vectores argumentos. Concretamente, esta operação é atingido calculando o somatório dos produtos entre posições correspondentes dos vectores argumentos. O algoritmo associado a esta operação encontra-se descrito na listagem 8.

**Algoritmo 8** Produto Interno de Dois Vectores

$result \leftarrow 0.0$
**for** $i = 0$ to $lenght$ **do**
  $result \leftarrow result + (in1[i] * in2[i])$
**end for**

A operação descrita é caracterizada por ser trivialmente paralelizável e por isso facilmente acelerada usando o modelo *master-worker* na arquitectura Cell/B.E.

Assumindo este algoritmo algumas semelhanças com a soma/subtracção de vectores, poderá ser alvo das mesmas técnicas de programação usadas no desenvolvimento da rotina *axpy*. Particularmente as diferenças residem na forma como o output da rotina é processado. Enquanto que na operação *axpy*, o output produzido consistia num vector que continha o resultado das somas/subtracções, na determinação do produto interno o resultado reduz-se a um único valor real que representa o somatório dos produtos entre posições correspondentes dos vectores operandos. No algoritmo 9 encontra-se representado um procedimento possível onde são discriminadas as transferências de memória efectuadas para realizar a operação pretendida. Neste algoritmo já é abrangida a técnica de utilização de *double-buffering*.

De salientar que o resultado final resulta de uma redução de todos as somas parciais

**Algoritmo 9** Transferência de Memória no Método *DOT* Utilizando *Double-Buffer*

$local\_result \leftarrow 0$
$local\_buffer\_in1[0] \leftarrow main\_buffer\_in1[0]$
$local\_buffer\_in2[0] \leftarrow main\_buffer\_in2[0]$
**for all** $blockinblocks$ **do**
  **if** $block+1 < total\_blocks$ **then**
    $local\_buffer\_in1[(block+1)\%2] \leftarrow main\_buffer\_in1[block+1]$
    $local\_buffer\_in2[(block+1)\%2] \leftarrow main\_buffer\_in2[block+1]$
  **end if**
  $wait(local\_buffer\_in1[block\%2])$
  $wait(local\_buffer\_in2[block\%2])$
  $local\_result \leftarrow local\_result + (local\_buffer\_in1[block\%2] \cdot local\_buffer\_in2[block\%2])$
**end for**
$main\_result[spe\_index] \leftarrow local\_result$
$wait(main\_result[spe\_index])$

determinadas por cada SPU, copiadas para zonas de memória independentes no fim da computação. Esta redução é elaborada pelo PPU do processador Cell, somando todas os valores calculados pelos SPU's, evitando assim acesso concorrente a memória partilhada e a sincronização entre SPE's.

O processamento vectorial efectuado pelos SPU's associado a este cálculo é ilustrado na listagem 6.5. Neste procedimento é calculado o produto entre os elementos dos dois vectores em blocos de 16 bytes. O resultado dessas sucessivas multiplicações é então somado a um vector de quatro elementos, sendo este reduzido no final da computação.

```
vec_float4 vscalar, *vin1, *vin2, vout;

vin1 = (vec_float4*) local_buffer_in1;
vin2 = (vec_float4*) local_buffer_in2;
vout = (vec_float4*) local_buffer_out;

for(i = 0; i < block_elements / 4; i++){
    vout = vout + vin1[i] * vin2[i];
}

return vout[0] + vout[1] + vout[2] + vout[3];
```

**Listing 6.5** Processamento Vectorial Associado ao Método *DOT*

Os cabeçalhos dos métodos em linguagem *C* desenvolvidos para a determinação do produto internos entre vectores em espaço PPU e SPU encontram-se listados em 6.6. Valores como o

tempo de processamento de produtos internos entre dois vectores de *floats* e *doubles* em SPU e PPU podem ser observados nas tabelas 6.3 e 6.4. As tabelas incluem também os *speedups* alcançados face ao PPU para várias dimensões dos vectores operandos.

```
float k_dot(float *in1, float *in2, long n);

double k_dot(double *in1, double *in2, long n);

float spu_dot(
  float *mm_in1, float *mm_in2,
  unsigned long elements);

double spu_dot(
  double *mm_in1, double *mm_in2,
  unsigned long elements);
```

**Listing 6.6** Cabeçalhos do Método *DOT* em Espaço PPU e SPU

| **dot - float** | x86 | PPU | SPU | |
|---|---|---|---|---|
| elementos | tempo (ms) | tempo (ms) | tempo (ms) | *speedup* |
| 1000 | 0,011 | 0,037 | 0,172 | 0,215 |
| 10000 | 0,114 | 0,379 | 0,170 | 2,227 |
| 100000 | 0,986 | 5,277 | 0,179 | 29,480 |
| 1000000 | 3,474 | 52,987 | 0,823 | 64,383 |
| 10000000 | 34,733 | 518,760 | 6,868 | 75,533 |

**Tabela 6.3** Tempos de execução em milissegundos e *speedup* alcançados para a execução da operação *dot* processada em x86 (Intel Core2 Duo 2.66GHz), PPU (PowerPC 3.2GHz) e SPU para precisão simples

| **dot - double** | x86 | PPU | SPU | |
|---|---|---|---|---|
| elementos | tempo (ms) | tempo (ms) | tempo (ms) | *speedup* |
| 1000 | 0,011 | 0,003 | 0,163 | 0,018 |
| 10000 | 0,012 | 0,059 | 0,168 | 0,351 |
| 100000 | 0,916 | 3,741 | 0,223 | 16,776 |
| 1000000 | 3,847 | 38,641 | 1,568 | 24,643 |
| 10000000 | 37,445 | 380,346 | 15,357 | 24,767 |

**Tabela 6.4** Tempos de execução em milissegundos e *speedup* alcançados para a execução da operação *dot* processada em x86 (Intel Core2 Duo 2.66GHz), PPU (PowerPC 3.2GHz) e SPU para precisão dupla

Dos valores apresentados podemos apenas salientar o *speedup* obtido para a operação com o maior número de elementos. Tal como na operação *axpy*, é também possível identificar nesta rotina o crescente *speedup* com o aumento do número de elementos dos vectores operandos. Mais uma vez observa-se o menor desempenho do PPU e os casos de vantagem da utlização de SPU's quando comparamos os tempos com os obtidos numa arquitectura x86.

## 6.5 SPMV - Produto Entre Matrizes Esparsas e Vectores

A multiplicação de matrizes esparsas por vectores é caracterizada por ser a operação com o maior nível de complexidade de todas as operações alvo de aceleração em SPU. Nesta rotina enfrentamos uma imensidade de problemas que não se encontravam nas operações descritas anteriormente.

A operação em causa, tem como argumentos uma matriz de dimensões *nxm* e um vector de dimensão *m* e pode ser definida como a determinação de um vector com *n* elementos em que em cada posição $(i)$ deste se encontra determinado o produto interno entre a linha *i* da matriz e o vector operando.

Desta forma, o procedimento sequencial, como descrito no algoritmo 10, para matrizes quadradas *nxn*, é caracterizado por uma complexidade temporal $O(n2)$ em que *n* é é igual ao número de linas/colunas da matriz a multiplicar.

**Algoritmo 10** Multiplicação de uma matriz por um vector

```
1: for i = 0 to n − 1 do
2:     y[i] = 0
3:     for j = 0 to n − 1 do
4:         y[i] = y[i] + A[i, j] ∗ x[j]
5:     end for
6: end for
```

Como referido no capítulo 2, as matrizes usadas pelo simulador são matrizes que representam a interacção entre partículas, gozando assim de um elevado grau de esparsidade. A utilização de um formato idêntico ao BCRS torna esta característica bastante proveitosa no que toca à computação associada a este método.

Desta forma, a complexidade temporal do algoritmo pode ser reduzida de $O(n2)$, em que *n* representa o número de linhas/colunas de uma matriz, para $O(n)$ em que *n* é igual ao número de blocos não nulos da matriz.

Este formato trás-nos no entanto alguns inconvenientes como a utilização de meta-dados que representam um índice para os dados propriamente ditos. Este tipo de estruturas de dados obriga a transferências de memória adicionais entre RAM e memória local a cada SPE devido às indirecções no acesso aos elementos.

Nesta secção será discutida a implementação da rotina responsável pela multiplicação de matrizes por vectores assim como algumas implicações que a natureza da arquitectura veio reflectir nas estruturas de dados usadas na solução desenvolvida.

Uma das primeiras decisões a tomar no desenho de uma solução com vista a acelerar a

multiplicação de matrizes por vectores segundo o modelo *master-worker* passa pela correcta divisão do trabalho a realizar pelos vários SPU's. Como será descrito de seguida, a incorrecta divisão do tratamento dos dados pode levar a acessos concorrentes de leitura e escrita em memória principal, forçando assim o uso de técnicas de sincronização entre os vários SPU's.

Como exemplo podemos descrever uma situação em que o trabalho relativo à multiplicação de uma matriz de 8x8 blocos e um vector é dividido entre os vários SPU's de forma a que cada SPU processa uma coluna da matriz.

O processamento a realizar por cada SPE passaria pela determinação do produto entre cada bloco a ele atribuído e a fracção correspondente do vector. O resultado calculado iria então ser somado à fracção do vector resultado em questão, onde estariam já armazenados valores determinados por outros SPE's. Neste pequeno exemplo, ilustrado na figura 6.5, aquando da soma do resultado parcial com os valores já contidos no vector resultado, o SPE iria forçosamente efectuar uma leitura sobre os valores do vector resultado e uma escrita depois de efectuar a soma com o resultado parcial.

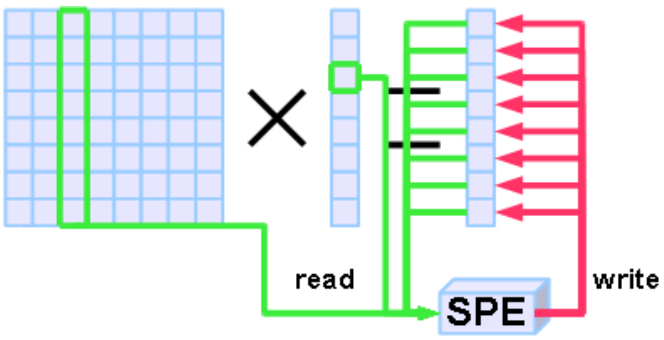


**Figura 6.5** Representação da Operação *SPMV* Efectuando Uma Divisão de Trabalho por Colunas

Este tipo de abordagem poderia causar *race conditions* e assim forçar a a aplicação de técnicas de exclusão mútua na secção crítica em causa. No entanto, o uso deste tipo de restrições provoca geralmente uma degradação no desempenho do processamento em causa.

De forma a eliminar por completo os acessos concorrentes às mesmas zonas de memória por parte dos vários SPE's, a distribuição do trabalho a ser realizado terá de ser efectuada de forma a que cada valor do vector resultado seja escrito apenas por um SPE. Assim podemos afirmar que cada valor contido no vector resultado seria escrito apenas uma vez com o valor definitivo do cálculo. Desta forma são evitados quaisquer problemas de partilha de memória e consequentemente eliminada a necessidade da utilização de técnicas de sincronização que poderiam degradar o desempenho obtido na operação.

Analisando a operação de multiplicação de matrizes por vectores podemos observar que para a determinação de um determinado valor do vector resultado, apenas são necessários os

dados correspondentes ao vector argumento e à linha em causa da matriz. Uma vez que os dados correspondentes a cada linha se encontram contíguos em memória, uma boa aproximação ao problema passa por, como ilustrado na figura 6.6, dividir o trabalho em aglomerados de linhas pelos diferentes SPE's.

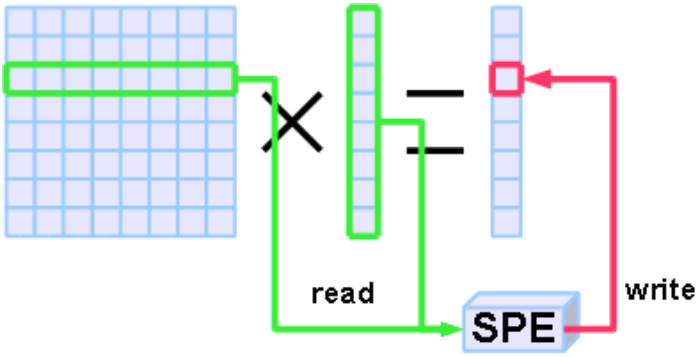


**Figura 6.6** Representação da Operação *SPMV* Efectuando Uma Divisão de Trabalho por Linhas

Desta forma o SPE fica responsável por efectuar a leitura da linha em causa, do vector argumento e, após o cálculo do produto interno da linha pelo vector argumento, a escrita do resultado na posição correspondente.

Mais concretamente, na rotina desenvolvida para o efeito nesta dissertação, cada SPE é responsável pela determinação de aproximadamente um oitavo do vector argumento devido à existência de 8 SPU's na arquitectura disponível. Uma vez que o número de blocos não nulos em cada linha da matriz representa a interacção de uma partícula com as partículas vizinhas e cada partícula apresenta aproximadamente o mesmo número de interacções, podemos concluir que, atribuindo aproximadamente um oitavo das linhas a cada SPE, a divisão do trabalho é suficientemente justa de forma a não se apresentar um desequilíbrio na computação efectuada pelas várias unidades de processamento vectoriais.

Assim, a ideia geral do algoritmo de multiplicação de matrizes, tomando em consideração as características do MFC disponível nos SPE's da arquitectura Cell B.E. e a estrutura de dados, passa pelo processamento individual sucessivo de cada uma das linhas da matriz. O programa é então responsável pela cópia da porção que lhe foi atribuída do vector *rows* (ver secção 6.2) de forma a determinar os endereços de memória e tamanhos dos dados que dizem respeito a cada linha. Após a cópia de cada linha da matriz, o processamento prossegue multiplicando-as pela porção do vector argumento correspondente.

Devido à reduzida quantidade de memória disponível em cada SPE, é importante perceber como várias aproximações ao problema poderão ou não influenciar o desempenho alcançado pela computação efectuada. Como indicado na secção 6.2, cada elemento do vector *blocks* contém informação relativa ao bloco da matriz propriamente dito e ao número da coluna a que esse bloco pertence. Esse valor, que representa o índice da coluna, é também usado para

determinar a porção do vector operando ao qual se vai multiplicar o bloco em causa. Assim, aquando do processamento de cada linha da matriz, apenas é necessário aceder a pequenas fracções do vector argumento. Podemos então identificar imediatamente duas aproximações relativamente simples a este problema.

Uma das aproximações (denominemo-la de *copy-on-demand* para efeitos de referência), passa por efectuar apenas a cópia de pequenas fracções do vector argumento quando se verificasse a necessidade de aceder a esses valores. Desta forma, a cada bloco a ser processado seria transferido a fracção correspondente do vector necessária ao processamento. Nesta abordagem, apenas são transferidos os valores absolutamente necessários para o problema, procedendo à libertação da memória após a computação de cada linha, não desperdiçando capacidade de armazenamento local. No entanto o facto de determinadas colunas conterem valores não nulos em múltiplas linhas da matriz vai obrigar a que sejam efectuadas algumas transferências de memória redundantes, sacrificando assim algum desempenho na operação. De notar que esta aproximação não exige grandes capacidades de memória, contribuindo assim para o suporte à multiplicação de matrizes com dimensões apenas limitadas pela memória virtual do computador.

Numa outra abordagem (denominemo-la de *pre-fetch* para efeitos de referência) será efectuada a cópia integral do vector argumento no inicio da operação. Desta forma não seriam efectuadas quaisquer transferências de memória redundantes. Por outro lado, a cópia integral do vector no inicio da operação e consequente permanência em memória durante toda a computação vem impor um limite na largura da matriz e dimensão do vector argumento devido à reduzida memória local disponível.

Agora que foi efectuada uma pequena descrição de ambas as abordagens seguidas no que toca ao método de armazenamento do vector argumento, retomemos a aproximação denominada *copy-on-demand*. Uma das dificuldades impostas por esta aproximação está relacionada com complexidade de programação imposta pela existência de índices nos dados a processar. Concretamente, o *array rows* contém informação acerca da posição do primeiro bloco e tamanho de cada linha no vector *blocks*. Por outro lado, cada bloco do vector contém informação acerca dos valores da matriz e do índice da porção do vector pelo qual tem de ser multiplicado o próprio bloco. Este pormenor vem elevar a complexidade associada à aplicação da técnica de *double-buffer* em relação às operações anteriores.

Para a correcta aplicação desta técnica, o *double-buffering* terá também de ser aplicado às transferências de memória associadas ao ciclo mais interior do processamento. O método de funcionamento do algoritmo passaria então a agendar a transferência da informação associada ao vector na iteração anterior à sua utilização. Para tornar isto possível, a informação relativa ao vector *blocks* seria forçosamente transferida duas iterações antes. Por outras palavras, imaginando que na iteração *i* seria efectuado a multiplicação da linha *i* da matriz pelo vector, na mesma iteração seria agendado a cópia da informação relativa aos blocos (e

consequentemente aos índices das colunas) da linha $i+2$ e, uma vez que essa informação acerca da linha $i+1$ já se encontra idealmente disponível, seria também agendada a transferência dos valores do vector necessários à operação.

O procedimento que representa as transferências de memória associado a esta abordagem encontra-se listado no algoritmo 11. Entenda-se por $local\_buffer\_blocks[n]$ o n-esimo buffer local dedicado ao armazenamento de blocos, $main\_buffer\_blocks[n]$ a porção do buffer de memória principal que representa a n-esima linha da matriz e por $local\_buffer\_array[m][n]$ a n-esima porção do m-esimo buffer de armazenamento do array argumento. De salientar que, de forma a apresentar alguma simplicidade no algoritmo descrito, não se encontram discriminadas as transferências associadas ao vector *rows*. Caso a porção do vector *rows* atribuída ao SPU ultrapasse os 16 KB, o processo descrito deverá ser repetido para cada fracção transferida.

Quanto ao processamento vectorial, pode ser descrito pela listagem 6.7. Este algoritmo determina, para cada linha, o conjunto dos valores em que resultam o produto interno da linha em questão e do vector argumento. Concretamente, em cada linha, o processo itera sobre todos os blocos, efectuando em cada um um conjunto de produtos entre cada linha do bloco e a porção correspondente do vector. O resultado de cada multiplicação é então somado ao valor acumulado ao longo da linha processada. No fim do processamento de cada linha este valor é reduzido através de uma soma e transferido para memória central. Podemos observar na figura 6.7 uma descrição gráfica do processamento vectorial do produto entre matrizes e vectores. Na figura encontra-se realçado com tracejado a fase da redução dos valores obtidos.

**Algoritmo 11** Transferência de Memória na Abordagem *copy-on-demand* do Método *AXPY*

```
    //prefetch the blocks from line 0
    if total_lines > 0 then
      local_buffer_blocks[0] ← main_buffer_blocks[0]
 5: end if
    //prefetch the blocks from line 1
    if total_lines > 1 then
      local_buffer_blocks[1] ← main_buffer_blocks[1]
    end if
10: //prefetch the array relative to the first line
    if total_lines > 0 then
      wait(local_buffer_blocks[0])
      for block = 0 to local_buffer_blocks[0].lenght do
        local_buffer_array[0][block] ← main_buffer_array[block.column]
15:   end for
    end if
    //for each line
    for line = 0 to total_lines do
      //prefetch the blocks from line + 2
20:   if total_lines > line + 2 then
        local_buffer_blocks[(line + 2)%3] < −main_buffer_blocks[line + 2]
      end if
      //prefetch the array relative to line + 1
      if total_lines > line + 1 then
25:     wait(local_buffer_blocks[(line + 1)%3])
        for block = 0 to local_buffer_blocks.lenght do
          local_buffer_array[(line + 1)%3][block] ← main_buffer_array[block.column]
        end for
      end if
30:   wait(local_buffer_array[line%3])
      wait(local_buffer_result[line%3])
      //process
      local_buffer_result[line%3] ←
      process_line(local_buffer_blocks[line%3], local_buffer_array[line%3]);
35:   //send the result to main memory
      main_buffer_result[line] < −local_buffer_result[line%3]
    end for
    wait(local_buffer_result[0])
    wait(local_buffer_result[1])
40: wait(local_buffer_result[2])
```

```
vec_float4 result_tmp[4] =
  {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
float result_portion[4] =
  {0,0,0,0};
long y;
vec_float4 *v, *l;

//para cada bloco a processar na linha
for(current_block = 0; current_block < n_blocks; current_block++)
{
  for(y = 0; y < BLOCK_SIZE; y++)
  {
    v = (vec_float4*) array[current_block];
    l = (vec_float4*) blocks[current_block].values[y];
    result_tmp[y] += (*v) * (*l);
  }
}

for(i = 0; i < 4; i++)
  for(j = 0; j < 4; j++)
    result_portion[i] += result_tmp[i][j];
```

**Listing 6.7** Processamento Vectorial Inerente ao Produto de Blocos por Fracções de Vectores

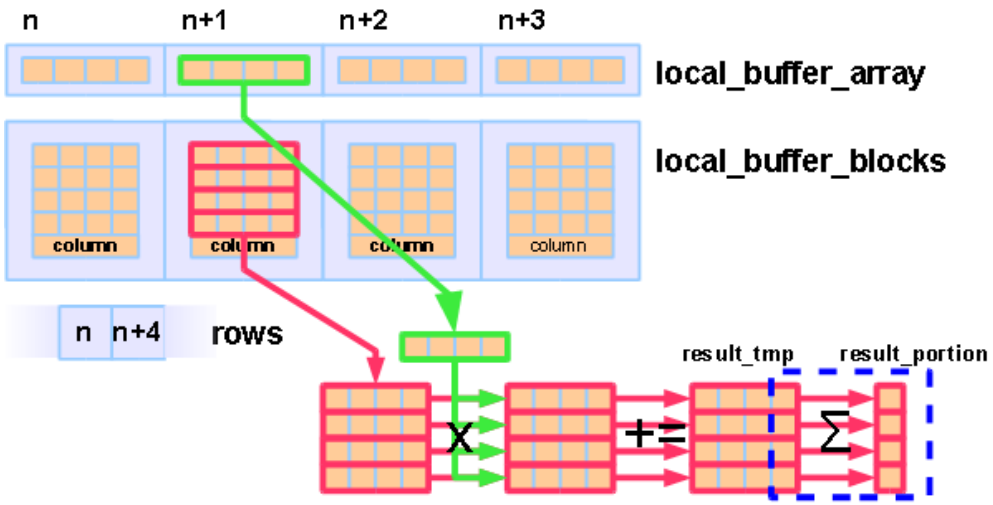


**Figura 6.7** Representação da Computação Efectuada Aquando do Processamento de Uma Linha na Operação *SPMV*

Na secção 6.2 foram descritas as estruturas de dados usadas para a representação de matrizes e vectores na versão acelerada do simulador. A descrição dessas estruturas referia um *padding*

obrigatório de forma a que cada conjunto de três valores perfizesse um total de 16 bytes tanto na representação dos vectores como das matrizes. A justificação para tal característica está intimamente ligado com o processamento vectorial e com as características do MFC.

Uma vez que os endereços de origem e destino de todas as transferências efectuadas pelo MFC são forçosamente um múltiplo de 16 bytes, a transferência dos resultados calculados pelos SPU's remete-nos para um problema já discutido nesta secção. No caso em que os blocos contidos na representação da matriz possuem uma dimensão 3x3, os SPU's determinam, por linha, apenas três posições do vector resultado. Uma vez que nos SPU's da arquitectura Cell/B.E. os *floats* são armazenados em 4 bytes e os *doubles* em 8 bytes, grande parte dos grupos de três valores armazenados no vector (75%) terão um endereço de memória que não perfaz um múltiplo de 16 bytes. Aquando do armazenamento do valor calculado, o SPE teria de transferir os três valores e a zona circundante desses valores (de forma a efectuar um comando MFC legal), substituir os valores em causa pelos três valores calculados e agendar uma transferência desses dados para memória central, substituindo os dados anteriores.

O procedimento descrito pode novamente levar a problemas de acesso concorrente nas zonas de memória contíguas à fronteira delineada para o processamento segundo o modelo *master-worker*. Por forma a evitar o uso de quaisquer mecanismos de sincronização entre SPE's é usado o *padding* referido na secção 6.2. Este *padding* é aplicado de forma a que todos os conjuntos de três ou quatro valores usado nos vectores durante a computação se encontrem num endereço de memória múltiplo de 16 bytes.

O *padding* presente nos blocos da matriz existe apenas com o intuito de simplificar a complexidade associada ao desenvolvimento da rotina de processamento vectorial no produto de blocos por porções do vector argumento, não forçando assim uma maior distinção nas rotinas de processamento de valores de precisão simples e dupla.

Uma outra variante do processo descrito é a abordagem referida como *pre-fetch*. Esta aproximação difere do algoritmo apresentado na forma como aborda a cópia para memória local do vector argumento à operação. Como foi já mencionado, todo o *array* é transferido para a memória do SPE no inicio da operação, evitando assim acessos redundantes e recorrentes a memória central.

Sem grande necessidade de entrar em pormenores uma vez que a generalidade do processo não difere da variante *copy-on-demand*, o procedimento, no que toca a transferências de memória, pode ser descrito segundo a algoritmo 12.

O processamento vectorial de ambas as abordagens seguidas apenas diferem num ponto quando comparadas. Enquanto que na abordagem *copy-on-demand* cada posição do vector *blocks* era multiplicada por cada segmento de um vector paralelo de igual dimensão, na vertente *pre-fetch* cada bloco é multiplicado pelo segmento do *array* argumento indexado pelo campo

**Algoritmo 12** Transferência de Memória na Abordagem *pre-fetch* do Método *AXPY*

```
    local_array ← main_array

    //prefetch the blocks from line 0
 5: if total_lines > 0 then
      local_buffer_blocks[0] ← main_buffer_blocks[0]
    end if
    //for each line
    for line = 0 to total_lines do
10:   //prefetch the blocks from line + 1
      if total_lines > line + 1 then
        local_buffer_blocks[(line + 1)%2] < −main_buffer_blocks[line + 1]
      end if

15:   wait(local_buffer_array[line%2])
      wait(local_buffer_result[line%2])

      //process
      local_buffer_result[line%2] ← process_line(local_buffer_blocks[line%2], local_array);
20:
      //send the result to main memory
      main_buffer_result[line] < −local_buffer_result[line%2]
    end for
    wait(local_buffer_result[0])
25: wait(local_buffer_result[1])
```

*column* do bloco em causa.

Nas tabelas 6.5 e 6.6 estão apresentados os tempos de execução e os *speedups* obtidos na determinação de produtos entre matrizes esparsas e vectores. Os testes executados em função do número de linhas testavam a rotina desenvolvida segundo as abordagens *copy-on-demand* e *pre-fetch* e tinham como argumento uma matriz quadrada com uma média de dez blocos 3x3 não nulos por linha. De salientar que os *speedups* alcançados comparam a implementação desenvolvida à multiplicação efectuada em CPU convencional usando o formato BCRS em ambos. De relembrar mais uma vez que a versão *pre-fetch* apresenta limitações no que toca ao valor da dimensão do vector argumento devido à reduzida quantidade de memória disponível em cada SPE. Assim, as linhas apresentadas com fundo cinzento expõem os valores obtidos para os testes executados com a maior dimensão possível para a matriz/vector argumento a que a abordagem *pre-fetch* oferece suporte.

| **spmv - float** | x86 | PPU | SPU (*copy-on-demand*) | | SPU (*pre-fetch*) | | Δ% |
|---|---|---|---|---|---|---|---|
| linhas | tempo (ms) | tempo (ms) | tempo (ms) | *speedup* | tempo (ms) | *speedup* | |
| 100 | 0,275 | 0,689 | 0,171 | 4,029 | 0,164 | 4,201 | 4,27% |
| 1000 | 1,450 | 8,259 | 0,884 | 9,343 | 0,242 | 34,128 | 265,29% |
| 10000 | 7,583 | 85,073 | 8,403 | 10,124 | 1,381 | 61,602 | 508,47% |
| 14000 | 10,533 | 119,102 | 11,736 | 10,148 | 1,909 | 62,390 | 514,77% |
| 100000 | 74,836 | 862,753 | 115,363 | 7,479 | n/a | n/a | n/a |
| 1000000 | 750,931 | 8285,844 | 1143,873 | 7,244 | n/a | n/a | n/a |

**Tabela 6.5** Tempos de execução em milissegundos e *speedup* alcançados para a execução da operação *spmv* processada em x86 (Intel Core2 Duo 2.66GHz), PPU (PowerPC 3.2GHz) e SPU para precisão simples

| **spmv - double** | x86 | PPU | SPU (*copy-on-demand*) | | SPU (*pre-fetch*) | | Δ% |
|---|---|---|---|---|---|---|---|
| linhas | tempo (ms) | tempo (ms) | tempo (ms) | *speedup* | tempo (ms) | *speedup* | |
| 100 | 0,285 | 0,049 | 0,17 | 0,287 | 0,169 | 0,290 | 1,18% |
| 1000 | 1,186 | 2,617 | 0,709 | 3,691 | 0,280 | 9,346 | 153,21% |
| 7000 | 6,130 | 20,305 | 4,673 | 4,345 | 1,322 | 15,359 | 253,48% |
| 10000 | 8,705 | 25,334 | 6,680 | 3,793 | n/a | n/a | n/a |
| 100000 | 85,970 | 270,872 | 60,620 | 4,468 | n/a | n/a | n/a |
| 1000000 | 854,560 | 2735,478 | 587,098 | 4,659 | n/a | n/a | n/a |

**Tabela 6.6** Tempos de execução em milissegundos e *speedup* alcançados para a execução da operação *spmv* processada em x86 (Intel Core2 Duo 2.66GHz), PPU (PowerPC 3.2GHz) e SPU para precisão dupla

Ao observar as tabelas apresentadas podemos notar a clara discrepância nos valores de *speedup* obtidos na bateria de execuções efectuadas que relacionam as duas abordagens desenvolvidas. Enquanto que os *speedups* obtidos com a abordagem *copy-on-demand* dificilmente ultrapassam as dez unidades, na aproximação *pre-fetch* todos os testes,

independente do número de linhas da matriz argumento, resultam num *speedup* bastante superior.

De forma a ultrapassar este problema, poderia ser desenhada uma rotina que aglomerasse as duas abordagens: O método procederia de forma a copiar a maior fracção possível do vector argumento de maneira a possibilitar o seu uso quando necessário. Para todos os valores não contidos na fracção do vector copiada, a rotina seguiria o procedimento segundo a abordagem *copy-on-demand* copiando o valor em causa quando necessário.

Na listagem exibida em 6.8 são apresentados os cabeçalhos em linguagem *C* dos métodos desenvolvidos em espaço PPU e SPU para fornecer suporte à multiplicação de matrizes por vectores.

```
//matrix block
struct spmv_block{
    float values[BLOCK_SIZE][4];
    long column;
} __attribute__ ((aligned(16)));

//vector block
struct spmv_vector{
  float values[BLOCK_SIZE];
} __attribute__ ((aligned(16)));

void k_spmv(
  struct spmv_vector * vector, long vector_length,
  long * rows, struct spmv_block *blocks, unsigned long lines,
  struct spmv_vector * result);

void spu_spmv(
  long mm_local_first, long mm_local_last,
  long mm_local_size, long mm_total_size,
  struct spmv_vector *mm_vector, long mm_vector_length,
  struct spmv_block *mm_blocks, long *mm_rows,
  struct spmv_vector *mm_result);
```

**Listing 6.8** Cabeçalhos do Método *SPMV* em Espaço PPU e SPU

# 7 . Avaliação de Resultados

O principal objectivo na elaboração desta dissertação passou pela avaliação das capacidade de processamento da arquitectura Cell/B.E. no que toca à computação associada a simulações físicas. Para o efeito foi alvo de aceleração através do uso desta arquitectura o simulador de superfícies deformáveis com realismo acrescido desenvolvido por Fernando Birra [10]. A animação efectuada pelo simulador em causa enquadra-se nos processos de modelação que recorrem a sistemas de partículas, discretizando as propriedades do tecido ao longo da sua superfície.

Depois de avaliada a arquitectura em causa e o simulador de superfícies procedeu-se então ao desenvolvimento da solução aqui proposta. Esta solução passa por deslocar o processamento associado à resolução de um sistema de equações por métodos implícitos do processador convencional e tirar proveito dos múltiplos cores disponíveis na arquitectura Cell/B.E. de forma a acelerar o processo.

Esta deslocação do processamento é efectuada segundo o modelo de programação *master-worker* de forma a dividir o processamento dos dados por todos os SPE's presentes no simulador.

Tendo em conta as várias fases do simulador descritas no capítulo 1, podemos afirmar que a animação da superfície avança ciclicamente e em cada iteração deste ciclo é resolvido o sistema de equações recorrendo ao método dos gradientes conjugados pré-condicionado modificado. Uma vez efectuado o deslocamento do processamento do método dos gradientes conjugados para SPU's deparamo-nos com um processamento iterativo que, a cada iteração efectua uma fracção do processamento nos SPU's (fracção acelerada) e o restante no PPU (fracção não acelerada).

Depois de implementada a solução proposta neste documento procedeu-se à execução de uma bateria de testes de forma a avaliar o desempenho obtido pela arquitectura em causa. Os testes efectuados basearam-se na medição do desempenho obtido pela versão acelerada para um processador Cell e comparação com os valores observados noutras arquitecturas. As simulações efectuadas baseavam-se na animação de uma peça de tecido quadrado em queda até entrar em contacto com o chão. Para as medições realizadas fez-se variar o número de partículas de forma a estudar a influência do volume de dados no desempenho obtido.

As medições apresentadas neste documento incluem, para cada número de partículas, os tempos tomados por um processador *Intel Core2 Duo* a 2.66GHz, pelo PPU presente no processador Cell (de acordo com a norma PowerPC) a 3.2 GHz e pela conjugação do PPU e SPU's presentes no computador Cell disponível. Para cada uma das configurações enumeradas é apresentado o desempenho medido aquando do processamento da simulação recorrendo a números de vírgula flutuante de precisão simples e dupla. São também apresentados os

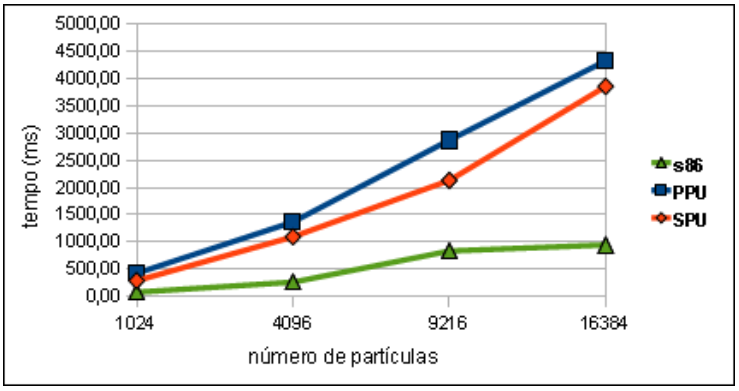


**Figura 7.1** Tempo médio de execução do passo *Solve* usando precisão simples

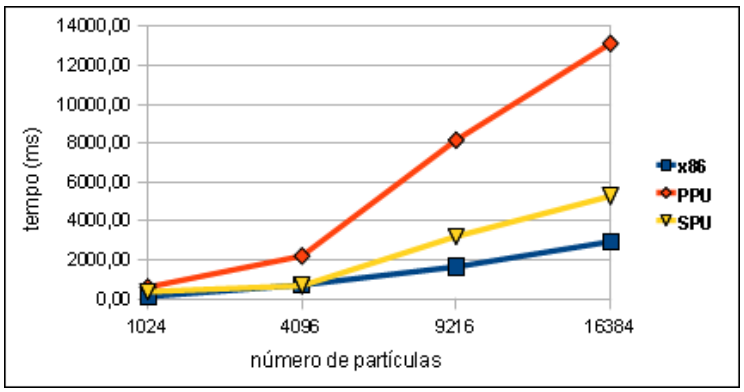


**Figura 7.2** Tempo médio de execução do passo *Solve* usando precisão dupla

resultados da solução baseada em programação GPU e *speedups* obtidos por esta face a um processador Core2 Duo a 2.4GHz.

Como referido no capítulo 1 deste documento, o perfil de execução do sumulador em causa é dividido em três fases, a serem repetidas ciclicamente. Foi identificada como a mais computacionalmente intesiva a fase denominada *Solve*. Nesta fase do simulador é construida a matriz representante do sistema de equações e resolvido através do método dos gradientes conjugados. A figura 7.1 e 7.2 apresentam os tempos de execução médios do passo *Solve* para os vários números de partículas em várias arquitecturas. Os gráficos em questão demonstram claramente que, para qualquer precisão e número de partículas com que seja efectuado a simulação, os tempos de execução tirando proveito do processador Cell são inferiores aos tempos de execução exclusivamente em SPU. No entanto, podemos ainda reparar que os tempos médios tomados pelo passo Solve no processador Intel são claramente inferiores a qualquer uma das configurações testadas.

De notar que os tempos exibidos nos gráficos apresentados incluem a execução da fracção

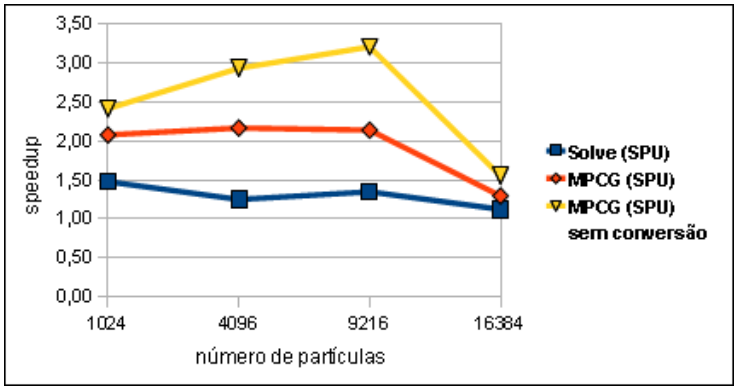


**Figura 7.3** *Speedup* médio obtido em relação à execução em PPU usando precisão simples

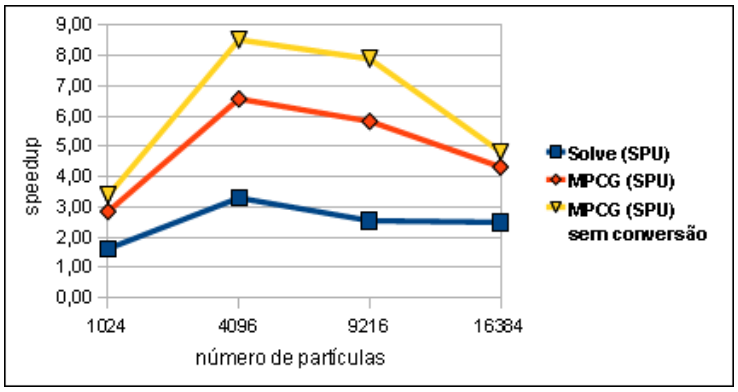


**Figura 7.4** *Speedup* médio obtido em relação à execução em PPU usando precisão dupla

não acelerada do passo *Solve* do simulador, sendo composto pelo tempo levado pela construção das matrizes que definem o sistema de partículas.

Podemos referir ainda o *overhead* introduzido no passo *Solve* devido à conversão em PPU das estruturas de dados em formatos favoráveis à aceleração usada nos SPU's.

De forma a estudar a relevância do *overhead* provocado pela construção das matrizes representativas do sistema de partículas e pela conversão de estruturas de dados podemos analisar os gráficos exibidos nas figuras 7.3 e 7.4 onde são apresentados os speedups obtidos pela solução implementada quando comparado coma execução exclusiva no PPU do processador Cell. Neles podemos observar os *speedups* obtidos na totalidade do passo *Solve*, apenas na execução do método dos gradientes conjugados implementado e na execução do mesmo método ignorando o *overhead* associado à conversão de estruturas de dados.

Como podemos observar, os *speedups* apresentados, sofrem um decrescimo a partir de um determinado número de partículas em ambas as precisões. A quebra nos valores apresentados

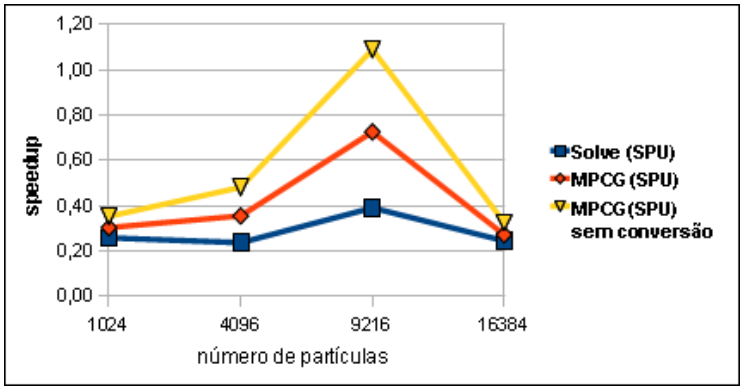


**Figura 7.5** *Speedup* médio obtido em relação à execução em x86 usando precisão simples

entre as 9216 e 16384 partículas no caso da precisão simples e entre as 4096 e 9216 partículas no caso da precisão simples devem-se à limitação imposta pela memória local dos SPE's na abordagem *pre-fetch* da operação *spmv*. A capacidade máxima desta rotina, no que toca à dimensão das matrizes em blocos (valor igual ao número de partículas) ronda as 14000 na versão de precisão simples e as 7000 em precisão dupla. Este facto vem forçar o uso da abordagem *copy-on-demand* para além dessas dimensões que, ao não atingir valores de *speedup* tão elevados, provoca um acréscimo no tempo de execução para valores superiores aos referidos.

Nas figuras referidas podemos ainda observar os valores absolutos de *speedups* obtidos. Como esperado, o *speedup* registado na totalidade do passo *Solve* apresenta valores bastante inferiores aos obtidos apenas na execução do método dos gradientes conjugados. É também possível observar o *overhead* provocado pela conversão das estruturas de dados pela diferença entre os *speedups* obtidos apenas no método dos gradientes conjugados e no mesmo método ignorando a transformação de dados.

Durante o desenvolvimento da solução apresentada foram efectuados testes relativos a cada rotina num processador Intel Core2 Duo a 2.66GHz, revelando estes melhores resultados que o PPU do processador Cell disponível. Torna-se assim importante verificar o comportamento da arquitectura x86 na execução das animações em causa.

Nas figuras 7.5 e 7.6 podemos consultar os *speedups* obtidos pela solução descrita neste documento em SPU quando comparada com os tempos de execução no processador Intel. De salientar que o elevado desempenho do processador Intel Core2 Duo disponível origina tempos de processamento para o método dos gradientes conjugados ignorando o *overhead*, em grande parte das configurações, menores que os obtidos na utilização dos SPU's do processador Cell. Se observarmos os valores relativos à totalidade do passo *Solve* podemos reparar que em apenas um dos testes executados se verificaram alguns ganhos no que toca a tempos de processamento.

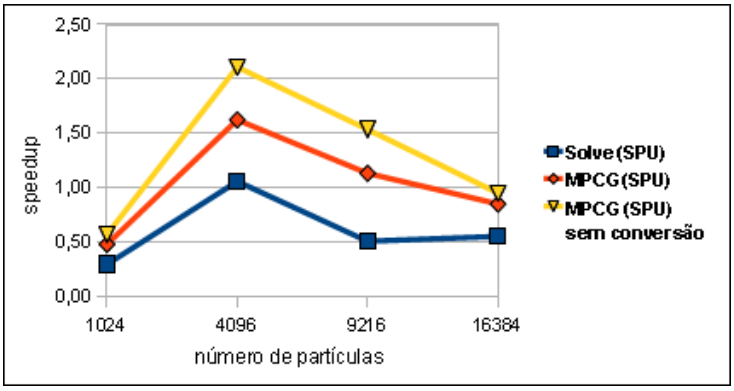


**Figura 7.6** *Speedup* médio obtido em relação à execução em x86 usando precisão dupla

Podemos ainda, ao observar a figura 7.6, que diz respeito à aceleração obtida pela execução em SPU's face ao processador Intel, verificar o ganho de desempenho aquando do processamento do *MPCG*. Este ganho torna-se ainda mais relevante no caso em que não são contabilizados os tempos associados à conversão das estruturas de dados a cada execução do passo *Solve*. Deparamo-nos então com uma situação em que a solução apresentada demonstra algum ganho no desempenho na fracção do processamento acelerada. No entanto, este ganho não acompanha o resto do processamento associado ao passo *Solve*. Este facto deve-se à pouca eficácia do PPU aquando do processamento da fracção não acelerada do processo, anulando em alguns casos os ganhos obtidos pelos SPU's.

Podemos assim concluir que a utilização conjunta do PPU com os SPU's presentes no processador Cell acabam por oferecer uma maior capacidade de processamento face ao processador Intel disponível. Por outro lado o processador Cell encontra-se bastante limitado devido à pouca eficácia do PPU aquando da computação não paralelizável de dados.

Na figura 7.7 são apresentados os speedups alcançados através da programação em GPU desenvolvida por João Rocha [18]. Os *speedups* observados no gráfico exibido podem apenas ser comparados aos *speedups* obtidos pela versão acelerada do simulador face à execução em PPU. No que toca à aceleração face ao processador Intel, apesar do processador disponível ser ligeiramente superior ao utilizado por João Rocha, o *speedup* alcançado para qualquer número de partículas é consideravelmente inferior ao obtido através da programação em GPU.

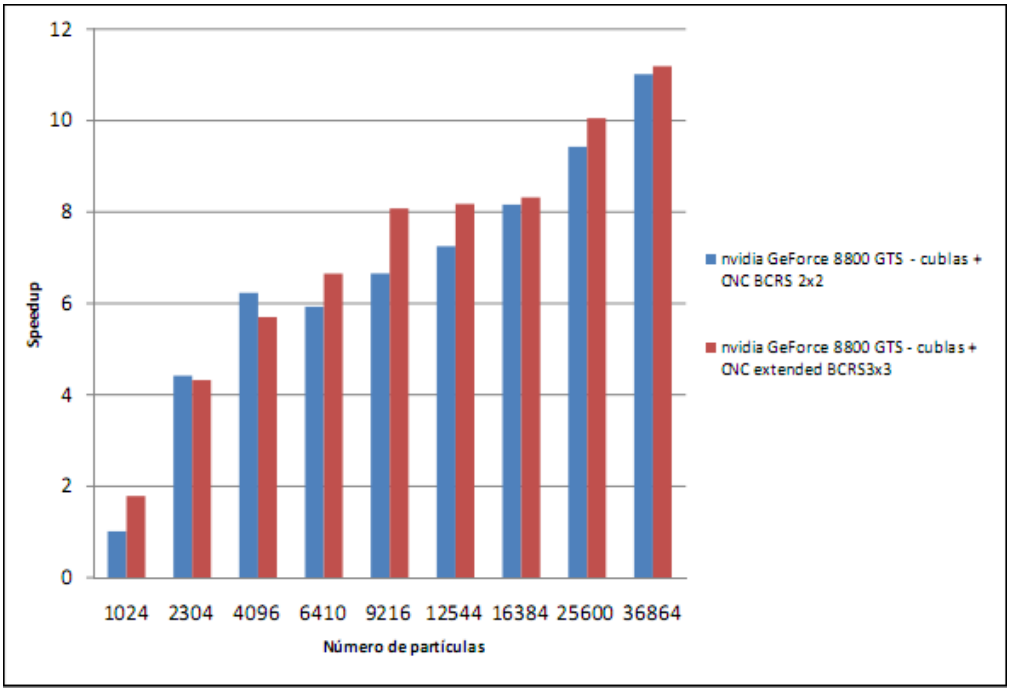


**Figura 7.7** *Speedup* obtidos em GPU face ao Core2 Duo 2.4GHz em função do número de partículas, recorrendo ao formato BCRS 2$x$2 e BCRS 3$x$3 [18]

# 8. Conclusões e Trabalho Futuro

A principal motivação para a elaboração desta dissertação passou pela aceleração do simulador de superfícies deformáveis com realismo acrescido desenvolvido em [10] tirando proveito das potencialidades da arquitectura Cell/B.E.

Neste documento foram apresentadas e discutidas algumas das técnicas de simulação de superfícies deformáveis, dando maior ênfase à simulação através de sistemas de partículas. Nestes modelos, geralmente denominados de modelos discretos, as características da superfície são discretizadas ao longo destas, formando assim um sistema de partículas representativas das propriedades físicas associadas a uma determinada zona da superfície em causa. Depois de construído o esquema representante das forças exercidas em cada partícula, um sistema de equações é resolvido de forma a determinar as novas posições das partículas depois de decorrido algum tempo. Esta fase é caracterizada por ser computacionalmente intensiva e, por isso, foi tida como o principal alvo de aceleração durante a realização desta dissertação.

O desenvolvimento do processador Cell, como descrito no capítulo 3, deu-se com o objectivo de criar uma arquitectura com capacidade de suportar um elevado leque de aplicações com elevado desempenho aliado ao baixo consumo energético.

O processador em causa pode ser visto como um chip *multicore* heterogéneo composto por uma unidade de acordo com a norma PowerPC e oito cores com capacidade de processamento vectorial (SIMD). Esta combinação oferece ao programador uma elevada flexibilidade no desenvolvimento de aplicações permitindo-lhe a aplicação de vários modelos de programação e facilitando a paralelização de um determinado processamento.

Depois de esclarecer claramente as vantagens e desvantagens de cada modelo de programação candidato no desenvolvimento da solução, procedeu-se à sua implementação e realização de testes de desempenho associados a cada rotina desenvolvida. No capítulo 6 deste documento encontram-se justificadas todas as tomadas de decisão envolvidas no seu desenvolvimento.

Procedeu-se então à aceleração do simulador concentrando os esforços na aceleração do método dos gradientes conjugados implementado. Este método representa uma elevada parte do processamento tomado pelo simulador. Assim, a aceleração da animação passou pela paralelização de todas as rotinas que compõem o método dos gradientes conjugados retirando proveito da arquitectura Cell/B.E. utilizando o modelo *master-worker*.

Devido ao elevado número de restrições no que toca ao formato dos dados que a arquitectura Cell/B.E. impõe, as estruturas utilizadas pelo simulador foram forçosamente transformadas de forma a permitir a sua paralelização e contribuir para o melhor desempenho da operação. Os dados, depois de proceder à sua tradução, são tratados quase exclusivamente

pelos SPE's presentes no processador Cell. A computação efectuada pelos SPU's, depois de efectuada, determina os resultados que terão de ser novamente traduzidos para os formatos originais usados no resto do simulador.

De salientar que o tempo de processamento levado nestas transformações, efectuadas no PPU que, como disctutido no capítulo anterior, oferece um fraco desempenho, causa um grande *overhead* na computação. Uma forma de reduzir ou mesmo eliminar este *overhead* passa pela utilização de estruturas de dados que facilitem a conversão efectuada ou pela utilização das mesmas estruturas de dados em toda a computação efectuada pelo simulador. O desenho de um formato que ofereça um suporte eficaz a todas os procedimentos efectuados e, simultaneamente, à aceleração utilizando a arquitectura Cell/B.E. contribuiria para um aumento de desempenho superior ao que se verificou na solução desenvolvida nesta dissertação.

Podemos então apontar o PPU como o maior *bottleneck* presente no processador Cell/B.E. Face a este problema, pode ser sugerida como potencial solução a utilização de um computador contendo dois (ou mais) processadores recorrendo ao sistema de memória partilhada. Entre estes processadores estaria presente um processador Cell/B.E. e um outro de outra arquitectura com maior desempenho no que toca a processamento não paralelizável, p.e. Intel x86.

A abordagem, tendo como base de trabalho um computador dotado destas características, passaria por organizar a computação de forma a que a porção que permitisse aceleração fosse efectuada no processador Cell, ocorrendo o restante processamento no outro processador. O PPU presente no processasdor Cell passaria então de uma unidade de processamento convencional a um elemento responsável por trocar mensagens com os SPE's de forma a efectuar alguma coordenação entre estes. Encontram-se actualmente tanto concluidas como sobre desenvolvimento algumas soluções que cumprem a descrição acima efectuadas, conjugando o poder do processamento paralelo e distribuído do processador Cell com os altos desempenhos obtidos por processadores mais potentes no que toca a processamento sequencial como as instalações Roadrunner e Aquasar [4, 6].

Relembrando os resultados exibidos no capítulo 7, apesar dos valores de *speedup* obtidos quando comparando a solução elaborada à execução exclusiva em PPU, serem geralmente maiores que a unidade, tal não se verifica aquando da comparação com a execução do simulador no processador Intel Core2 Duo a 2.66GHz. Podemos assim concluir que, apesar do sucesso obtido na elaboração de uma solução que tirasse proveito da arquitectura heterogénea do processador Cell, tal esforço não foi suficiente para que os resultados obtidos com este processador ultrapassassem o desempenho demonstrado pelo processador Intel disponível.

De realçar ainda que os resultados obtidos na programação GPU [18] se revelaram bastante mais satisfatórios que os alcançados através da aceleração do processamento utilizando a arquitectura Cell/B.E. utilizada nesta dissertação.

Uma outra abordagem, para além da descrita neste documento, que poderia ter contribuído

para o aumento do desempenho verificado na animação de superfícies deformáveis passava pela utilização do modelo de programação *streaming model*. Devido à elevada complexidade associada a esta técnica, não se realizou a aceleração de nenhuma das rotinas desta forma. Este modelo contribuiria para o aumento de desempenho do simulador devido ao menor número de transferências efectuada entre memória local a cada SPE e memória principal e poderia ser considerado como trabalho futuro.

O desenvolvimento de *kernel*'s com capacidade de efectuar um maior número de operações sequenciais sobre os mesmos dados poderia também proporcionar o ganho de algum desempenho. Podemos inspirar-nos no método *axpy* com capacidade de escalar um vector e somar o resultado a um segundo. A vantagem deste tipo de rotinas veio demonstrar-se na redução do número de transferências de memória efectuadas. Apenas a título de exemplo podemos imaginar uma operação com capacidade de determinar o produto interno entre o resultado de duas somas de vectores. Este tipo de rotinas compostas por mais que uma operação poderiam ser facilmente desenvolvidas depois de identificadas no código relativo ao método dos gradientes conjugados implementado.

Apesar dos computadores disponíveis para a realização desta dissertação estarem equipados com dois processadores Cell, não foi tirado proveito desse facto. Tendo em vista trabalhos futuros, poderíamos considerar o desenvolvimento de uma solução que beneficiasse da presença do segundo processador Cell com acesso a memória partilhada.

Uma outra solução mais complexa passava pela aceleração do simulador utilizando uma rede de computadores, onde a computação era dividida entre os vários nós. De salientar que o processador Cell com velocidades idênticas às presentes no IBM Blade Center QS21 se encontra disponível nas consolas PlayStation3 a um custo reduzido. Como referido no capítulo 3, esta arquitectura é caracterizada por um baixo consumo energético, que tornaria viável esta aproximação para projectos de maior duração. Analisando os dados apresentados na tabela 3.1 podemos verifacar claramente o ganho dos processadores Cell/B.E. face a outras arquitecturas no que toca aos recursos energéticos despendidos.

# A . Demonstração da Não-Conformidade dos SPU's Cell Com a Norma IEEE-754

## A.1 Código Fonte

### A.1.1 PPU

```
#include <stdio.h>
#include <assert.h>
#include <libspe2.h>
#include <pthread.h>

extern spe_program_handle_t epsilon_spu;

void *ppu_thread_function(void *arguments){
  spe_context_ptr_t ctx;
  unsigned int entry = SPE_DEFAULT_ENTRY;

  //create context
  assert((ctx = spe_context_create(0,NULL)) != NULL);
  //load program
  assert(spe_program_load(ctx, &epsilon_spu) == 0);
  //run
  assert(spe_context_run(ctx, &entry, 0, NULL, NULL, NULL) >= 0);

  pthread_exit(NULL);
}

int main(){
  pthread_t th;

  float x1 = -0.00000000000000002631267484231591415565210123617135;
  float x2 = -0.00001435546892025740817189216613769531250000000000;
  float x3 = -0.00000000033109007202547502402012469246983528137207;

  float y1 = -0.00000000010430811769879255734849721193313598632812;
  float y2 = -0.00000000000000000000000000000000000000000000000000;
  float y3 = 0.00423046899959444999694824218750000000000000000000;

  printf("PPU:\n");
  printf("x:\n%.50f\n%.50f\n%.50f\n", x1, x2, x3);
  printf("y:\n%.50f\n%.50f\n%.50f\n", y1, y2, y3);

  printf("x1*y1 + x2*y2 + x3*y3 =\n%.50f\n\n", x1*y1 + x2*y2 + x3*y3);

  assert(pthread_create(&th, NULL, &ppu_thread_function, NULL) == 0);
  assert(pthread_join (th, NULL) == 0);
```

```
  return 0;
}
```

### A.1.2 SPU

```
#include <stdio.h>

int main(unsigned long long speid,
         unsigned long long argp,
         unsigned long long envp){


  float x1 = -0.00000000000000002631267484231591415565210123617135;
  float x2 = -0.00001435546892025740817189216613769531250000000000;
  float x3 = -0.00000000033109007202547502402012469246983528137207;

  float y1 = -0.00000000010430811769879255734849721193313598632812;
  float y2 = -0.00000000000000000000000000000000000000000000000000;
  float y3 = 0.00423046899959444999694824218750000000000000000000;

  printf("SPU:\n");
  printf("x:\n%.50f\n%.50f\n%.50f\n", x1, x2, x3);
  printf("y:\n%.50f\n%.50f\n%.50f\n", y1, y2, y3);

  printf("x1*y1 + x2*y2 + x3*y3 =\n%.50f\n\n", x1*y1 + x2*y2 + x3*y3);

  return 0;
}
```

## A.2 Output

```
Resultado da execução em PPU:
x:
-0.00000000000000002631267484231591415565210123617135
-0.00001435546892025740817189216613769531250000000000
-0.00000000033109007202547502402012469246983528137207
y:
-0.00000000010430811769879255734849721193313598632812
-0.00000000000000000000000000000000000000000000000000
0.00423046899959444999694824218750000000000000000000
x1*y1 + x2*y2 + x3*y3 =
-0.00000000000140066625832491187253481257357634603977

Resultado da execução em SPU:
x:
-0.00000000000000002631267484231591415565210123617135
-0.00001435546892025740817189216613769531250000000000
-0.00000000033109007202547502402012469246983528137207
y:
-0.00000000010430811769879255734849721193313598632812
-0.00000000000000000000000000000000000000000000000000
0.00423046899959444999694824218750000000000000000000
x1*y1 + x2*y2 + x3*y3 =
-0.00000000000140066614990469462398436917283106595278

Resultado da execução em x86:
x:
-0.00000000000000002631267484231591415565210123617135
-0.00001435546892025740817189216613769531250000000000
-0.00000000033109007202547502402012469246983528137207
y:
-0.00000000010430811769879255734849721193313598632812
-0.00000000000000000000000000000000000000000000000000
0.00423046899959444999694824218750000000000000000000
x1*y1 + x2*y2 + x3*y3 =
-0.00000000000140066628577726289239357046558556832471
```

**Listing A.1** Output gerado pela execução dos programas apresentados em A.1.2 e A.1.1. A diferença pode ser constatada na última linha de cada output gerado.

# Bibliografia


[1] *Cell Broadband Engine Architecture, version 1.02*. IBM Systems and Technology Group, 2007.

[2] 754-2008 ieee standard for floating-point arithmetic. *IEEE Xplore Digital Library*, 2008.

[3] Basic linear algebra subprograms library programmer's guide and api reference, version 3.1. *IBM*, 2008.

[4] Roadrunner platform overview. *Roadrunner Technical Seminar Series*, 2008.

[5] *Software Development Kit for Multicore Acceleration - Porogramming Tutorial*. IBM Systems and Technology Group, 2008.

[6] Ibm and eth zurich unveil plan to build new kind of water-cooled supercomputer. `http://www-03.ibm.com/press/us/en/pressrelease/27816.wss`, June 2009.

[7] M. Ascher and Eddy Boxerman. On the modified conjugate gradient method in cloth simulation. *The Visual Computer*, 19(7-8):526–531, 2003.

[8] David Baraff. Physically based modeling, implicit methods for differencial equations. 2001.

[9] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 43–54, New York, NY, USA, 1998. ACM.

[10] Fernando Birra. Técnicas eficientes de simulação de tecidos com realismo acrescido. *Dissertação para a obtenção do grau de Doutor em Informática pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia*, 2007.

[11] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Soft.*, 28(2):135–151, 2002.

[12] L. Buatois, G. Caumon, and B. Lévy. Concurrent number cruncher: An efficient sparse linear solver on the gpu. *High Performance Computation Conference (HPCC), Springer Lecture Notes in Computer Sciences*, 2007.

[13] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, and G. Bosilca. *A Rough Guide to Scientific Computing On the PlayStation 3, version 1.0*. 2007.

[14] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[15] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60, New York, NY, USA, 2006. ACM.

[16] IBM Research. Ibm compiler information center f or multicore acceleration for linux. `http://publib.boulder.ibm.com/infocenter/cellcomp/v101v121/index.jsp`, November 2008.

[17] IBM Research. The cell architecture. `http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.innovation.html`, January 2010.

[18] João Rocha. Aceleração gpu da animação de superfícies deformáveis. *Dissertação para a obtenção do grau de Mestre em Informática pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia*, 2008.

[19] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, 2006.

[20] Andrew Witkin and David Baraff. Introduction to physically based modelling. course notes. 1995.