UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA MECÂNICA

Relatório Final

Trabalho de Conclusão de Curso

Aplicações para sistemas embarcados utilizando paralelismo

Autor: **Otávio Netto Zani**

Orientador: **Prof. Dr. André Ricardo Fioravanti**

Campinas, novembro de 2015

UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA MECÂNICA

Relatório Final

Trabalho de Conclusão de Curso

Aplicações para sistemas embarcados utilizando paralelismo

Autor: **Otávio Netto Zani**

Orientador: **Prof. Dr. André Ricardo Fioravanti**

Curso: Engenharia de Controle e Automação

Trabalho de Conclusão de Curso apresentado à Comissão de Graduação da Faculdade de Engenharia Mecânica, como requisito para a obtenção do título de Engenheiro de Controle e Automação.

Campinas, 2015

S.P. – Brasil

Índice

Resumo

ZANI, Otávio Netto, *Aplicações para sistemas embarcados utilizando paralelismo*, Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, Trabalho de Conclusão de Curso (2015), 29 pp.

Utilizando-se de paralelismo computacional e de *hardwares* de alta performance buscaremos diminuir o tempo de execução do algoritmo clássico de cálculo da "Transformada Rápida de Fourier" (FFT). O objetivo nessa etapa do trabalho é utilizar, da melhor maneira possível, o máximo de recursos que nosso hardware pode nos prover.

Em seguida, visando os mesmos objetivos, porém aplicados de maneira distinta, vamos avaliar a performance da decodificação de um código LDPC com um algoritmo computacional paralelo criado para esse propósito.

Esses algoritmos são geralmente implementados em FPGA, seguindo uma lógica distinta da utilizada em linguagens de programação como C. Nosso objetivo final é verificar se podemos, com a utilização de paralelismo, obter resultados compatíveis com sua implementação em *hardware*.

Após isso, utilizaremos os conceitos desenvolvidos nessas etapas para desenvolver uma *Framework* de simulação de efeitos físicos (tal como diversos jogos digitais utilizam hoje). *Frameworks* como essa são também implementadas paralelamente em GPUs, como por exemplo a PhysX, produzida pela Nvidia.

Palavras Chave: Paralelismo, Parallella, Epiphany, FFT, LDPC, Parity-Check, Memória Compartilhada, Simulação de Colisões, TCP, C, Objective C.

Lista de Figuras

Lista de Quadros

Nomenclatura

Letras Latinas

F(x)	Transformada de Fourier do sinal x

Siglas

FFT	*Fast Fourier Transform*
RAM	*Random Access Memory*
DMA	*Direct Memory Access*
SDK	*Software Develop Kit*
LDPC	*Low Density Parity-Check*

Capítulo 1

Introdução

A utilização de paralelismo computacional está cada vez mais popular nos últimos anos, dado que processadores *multicore* tem se tornado mais acessíveis e necessários para se obter uma maior performance nos sistemas computacionais.

Para sistemas embarcados isso não é diferente. Temos visto *hardwares* dotados de processadores *multicore* e limites grandes de memória tomarem o lugar dos micro-controladores, permitindo tarefas que necessitam de muitos recursos computacionais serem executadas mais rapidamente nesses sistemas.

Tendo isso em vista, utilizaremos uma placa recentemente desenvolvida e comercializada (*Parallella Board*) para explorar as capacidades do processamento *multicore* nos sistemas embarcados.

Inicialmente, para fins de explorar as capacidades permitidas pelo hardware selecionado, revisaremos o algoritmo clássico da FFT, com a intenção de reduzir o seu tempo de processamento. Para avaliar a efetividade do desenvolvido, faremos uma comparação com a mesma implementação do algoritmo rodando no mesmo hardware, mas utilizando apenas um *core.*

Além disso, revisaremos a decodificação de códigos LDPC em uma implementação *multicore* e avaliaremos a máxima velocidade de transmissão de dados em que esse decodificador pode operar.

Após a avaliação dos resultados, implementaremos uma *Framework* de simulação de efeitos físicos (tal como diversos jogos digitais utilizam) capaz de utilizar toda capacidade permitida pelo hardware, visando reduzir o tempo excessivo que esse tipo de *Framework* normalmente leva para seus cálculos. Para avaliar a efetividade dessa *Framework*, faremos testes visuais para encontrar o limite de suas capacidades.

Capítulo 2

Revisão Bibliográfica

Nesse capítulo introduziremos o *hardware* com que estamos trabalhando e, em seguida, trataremos dos algoritmos para FFT, LDPC e simulações de física na ordem em que foram implementados.

Primeiro abordaremos as principais características da arquitetura da "*Parallella Board*". Em seguida, iremos descrever as bases matemáticas que utilizaremos para implementação do algoritmo clássico da FFT. Após isso, trataremos do algoritmo de decodificação de códigos LDPC. Por último, abordaremos os métodos utilizados em *Frameworks* de simulação de física.

2.1. Arquitetura da *Parallella Board*

A *Parallella Board,* produzida pela *Adapteva*, é um hardware de alta performance e baixo consumo de energia.

Ele é dotado de 2 processadores, sendo um processador mestre (*dual-core* ARM A9 Zynq, com 1GB DDR3 de RAM disponível) e um co-processador (Epiphany 16-core CPU, com 32KB de memória Cache por core).

Além disso, ele possui um FPGA com 28K células lógicas, que é majoritariamente utilizado para implementar sua porta HDMI e a interface DMA entre o processador mestre e o co-processador.

Visto que grande parte de sua arquitetura é similar a de um sistema comum, focaremos nosso estudo no seu co-processador, que é seu principal diferencial.

2.1.1. Arquitetura do Epiphany 16-core CPU

O processador Epiphany consiste em uma matriz bidimensional de nós, conectados por uma malha de baixa latência.

Cada um desses nós é composto principalmente de 4 elementos:

- Uma CPU RISC de ponto flutuante.
- Uma memória local.

- Uma interface de rede para se comunicar com os outros nós.
- Um DMA para realizar acesso a memórias externas à do nó.

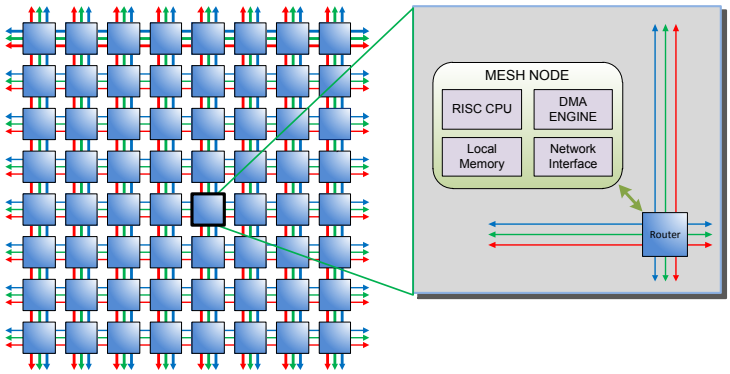

Figura 2.1 - Implementação da Arquitetura Epiphany (retirado de [2])

Devido a arquitetura de sua rede de comunicação entre os cores de seu co-processador, é possível se conectar mais de uma *Parallella* em série, a fim de se expandir a quantidade de núcleos de processamento.

Outra menção importante a se fazer é a seu mapeamento de memória.

Cada core possui memória interna mapeada de 0x00000000 a 0x00007FFF. Cada endereço de memória é composto de 1byte de 8bits. Os 4 primeiros bytes são utilizados para se identificar (dentro do escopo de memória compartilhada) qual core possui aqueles endereçamentos. Por exemplo, o endereço 0x00100021 implica na memória de endereço 0x00000021 no CORE_0_1. É importante ressaltar que os endereços de memória entre 0x00000000 e 0x00001FFF podem sobrescrever dados contidos neles, ou seja, para o programador estão disponíveis os endereços de 0x00002000 a 0x00007FFF.

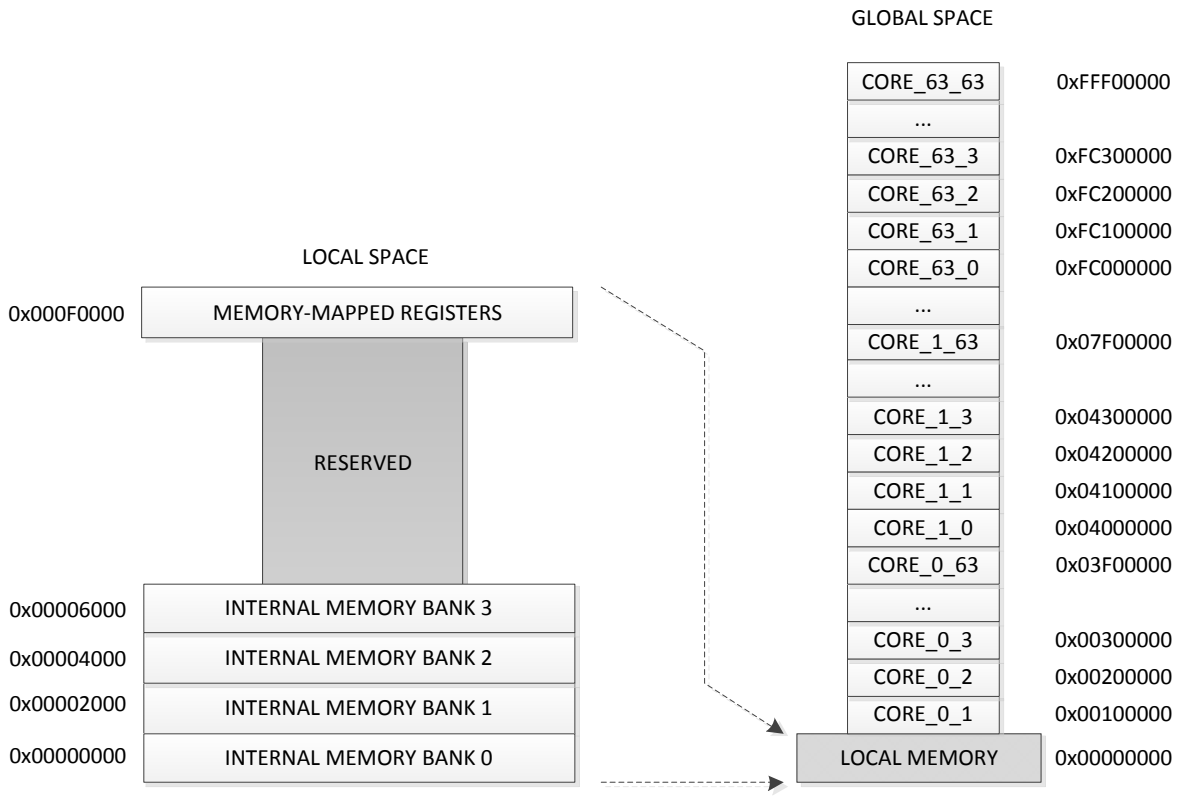

Figura 2.2 - Mapa de Endereços Globais da Epiphany (retirado de [2])

Apesar de sua grande capacidade de processamento, o *Epiphany* não dispõe de implementação em hardware de divisões (no caso de usá-las, seu compilador gera uma divisão em software) nem de precisão dupla (ou seja, ele não trabalha com dados do tipo *double* ou *long*).

2.2. Transformada Rápida de Fourier

A transformada rápida de Fourier é uma ferramenta utilizada para decompor sinais em suas componentes em frequência e amplitude. Seus usos mais gerais se encontram nas áreas de processamento de sinais, processamento de imagens (bidimensional) e processamento de som.

Queremos calcular a transformada discreta de Fourier. Para tal, partiremos de sua formulação.

Assumindo *N* elementos:

$$N = 2^k \qquad \text{(Equação 2.1)}$$

Temos a transformada discreta de Fourier dada por:

$$F(k) = \sum_{n=0}^{N-1} x(n) \cdot (W_N)^{kn} \qquad \text{(Equação 2.2)}$$

Onde:

$$W_N = e^{\frac{-2 \cdot \pi \cdot j}{N}} \qquad \text{(Equação 2.3)}$$

Podemos separar a somatória em duas partes, sendo uma para seus valores de índice ímpar e uma para seus valores de índice par. Com isso, obtemos:

$$F(k) = \sum_{m=0}^{\frac{N}{2}-1} x(2m) \cdot (W_N)^{2mk} + \sum_{m=0}^{\frac{N}{2}-1} x(2m+1) \cdot (W_N)^{(2m+1)k}$$

Sabendo que:

$$(W_N)^2 = \left(e^{\frac{-2\pi j}{N}}\right)^2 = e^{\frac{-2\pi j}{1} \cdot \frac{2}{N}} = W_{\frac{N}{2}}$$

Podemos novamente reescrever nossa transformada para obter:

$$F(k) = \sum_{m=0}^{\frac{N}{2}-1} x(2m) \cdot \left(W_{\frac{N}{2}}\right)^{mk} + (W_N)^k \cdot \sum_{m=0}^{\frac{N}{2}-1} x(2m+1) \cdot \left(W_{\frac{N}{2}}\right)^{mk} = F1(k) + (W_N)^k \cdot F2(k)$$

$$F(k) = F1(k) + (W_N)^k \cdot F2(k) \qquad \text{(Equação 2.4)}$$

Onde *F1(k)* e *F2(k)* são as transformadas discretas de Fourier dos elementos de índice par e de índice ímpar respectivamente.

Utilizando também da propriedade:

$$(W_N)^{\frac{N}{2}+k} = -W_N^k \qquad \text{(Equação 2.5)}$$

Podemos chegar ao algoritmo computacional para o cálculo da Transformada discreta de Fourier, descrito abaixo em pseudocódigo:

```
FFT(Complex vector[], unsigned size){
    float PI = 3.1416;
    if (size == 1){
      return;
    }

    vector[0->size/2] = vector[even];
    vector[size/2->size] = vector[uneven];

    FFT(vector[0->size/2], size/2);
    FFT(vector[size/2->size], size/2);

    for(i=0->size/2){
      Complex Wnk;
      Wnk.real = cos(-2.*PI*i/size);
      Wnk.imag = sin(-2.*PI*i/size);
      vector[i] = vector[i]+Wnk*vector[size/2+i];
      vector[i+size/2] = vector[i]-Wnk*vector[size/2+i];
    }
    return;
}
```

Quadro 2.1 - Pseudocódigo da Transformada Rápida de Fourier

2.3. Decodificação de Códigos LDPC

Códigos LDPC são utilizados para reconstruir dados corrompidos através de testes de paridade. São comumente utilizados em sistemas que necessitam de altas taxas de transmissão, como por exemplo transmissão digital de televisão. Nesse trabalho abordaremos apenas a decodificação desse código utilizando o método de *hard decision* para corrigir os erros no código recebido.

O algoritmo de *hard decision[4]* é feito nos seguintes passos:

Dada uma matriz "H", esparsa, binária e um vetor de bits “r”, temos que a multiplicação dessa matriz por esse vetor deve resultar (em módulo 2) em um vetor nulo, ou seja:

$$H \cdot r = V \qquad \text{(Equação 2.6)}$$

$$V\%2 = 0 \qquad \text{(Equação 2.7)}$$

Caso o resultado dessa multiplicação seja diferente o vetor nulo, precisamos corrigir o código que nos foi enviado. Para isso, executamos a multiplicação citada na Equação 2.6 linha a linha e verificamos o resultado dessa multiplicação aplicando a Equação 2.7 para seu resultado.

No caso de esse resultado ser 0, fazemos a seguinte contagem para cada bit do vetor r: caso o bit avaliado seja 1, adicionamos 1 ao contador desse bit, e, caso seja 0, subtraímos 1 do contador desse bit.

Já, no caso de aquele resultado ser 1, invertemos os sinais da contagem, ou seja: caso o bit avaliado seja 0, subtraímos 1 do contador desse bit, caso contrário, adicionamos 1 ao contador desse bit.

Após terminarmos esse processo, verificamos o contador de cada um dos bits. Se o contador for maior que 0, significa que a probabilidade de seu bit ser 1 é mais alta do que ser 0, logo, alteramos esse bit para 1. Se o contador for menor que 0, significa que a probabilidade de seu bit ser 0 é mais alta do que ser 1, logo, alteramos esse bit para 0. No caso do contador ser 0, mantemos o antigo valor daquele bit.

Com nosso vetor corrigido dessa maneira, voltamos ao começo do algoritmo de decodificação e o executamos com esse nosso novo vetor.

Repetimos esses passos até o resultado da Equação 2.6 ser nulo (em módulo 2), tendo assim nossa mensagem decodificada.

2.4. Detecção de Colisões

Abordaremos nessa seção os algoritmos utilizados para detecção de colisões em dois tipos de objetos distintos: circunferências e polígonos (convexos e côncavos).

Demonstraremos também os métodos utilizados para se encontrar o vetor normal à colisão, a penetração dos objetos e as simplificações adotadas para determinação desses parâmetros.

Nas *Frameworks* comerciais, o ponto onde ocorre a colisão entre os objetos não é determinado com precisão, visto que os erros associados a essa decisão não são relevantes para se obter um resultado realista.

Nesse capítulo, utilizaremos a seguinte notação por similaridade com o produto vetorial e produto escalar em espaço tridimensional:

Dados dois vetores V1 e V2, podemos calcular:

$$V1 = (X1, Y1)$$

$$V2 = (X2, Y2)$$

$$V1xV2 = X1 \cdot Y2 - X2 \cdot Y1 \qquad \text{(Equação 2.8)}$$

$$V1 \cdot V2 = X1 \cdot X2 + Y1 \cdot Y2 \qquad \text{(Equação 2.9)}$$

2.4.1. Detecção de Colisão entre Duas Circunferências

O caso mais simples de colisão que podemos tratar é o entre duas circunferências.

É fácil notar que a colisão entre duas circunferências ocorre quando a distância entre seus centros geométricos é menor do que a soma dos raios das circunferências. Nesse caso, o vetor normal da colisão é o vetor unitário que aponta do centro de uma das circunferências para o centro da outra, e a penetração é igual à soma de seus raios menos módulo da distância entre os centros das circunferências. O ponto a colisão será, nesse caso, o ponto médio da semi-reta definida pelos centros das circunferências.

Desse modo, para os circunferências 1, de centro C1 e raio r1, e 2, de centro C2 e raio r2:

$$N = \frac{(C1\text{-}C2)}{|C1\text{-}C2|} \qquad \text{(Equação 2.10)}$$

$$Pen = (r2+r1) - |C1\text{-}C2| \qquad \text{(Equação 2.11)}$$

$$P = \frac{(C1+C2)}{2} \qquad \text{(Equação 2.12)}$$

2.4.2. Detecção de Colisão entre dois Polígonos

Trataremos aqui de duas maneiras de se detectar a colisão entre dois polígonos quaisquer.

A primeira delas (e mais utilizada na literatura) consiste em separar o seu polígono em diversos triângulos e então determinar se qualquer vértice de um dos polígonos se encontra interno a qualquer um dos triângulos de outro objeto, esse método não será abordado a fundo devido sua simplicidade, porém será abordado um método para separação de um polígono em diversos triângulos.

A solução da literatura, no entanto, não aborda um dos casos de colisão que pode ocorrer, onde há colisão entre ambos os objetos, porém nenhum vértice se encontra interno a qualquer triângulo dos objetos (conforme Figura 2.3). Nesse caso, propomos uma outra maneira de determinar nossas colisões. Esse novo método consiste em verificar se há pelo menos um tipo de intersecção entre as arestas dos objetos conforme os tipos de intersecção mostrados abaixo.

Figura 2.3 - Caso não coberto pela literatura (há uma colisão entre os polígonos porém nenhum vértice está interno ao outro polígono).

2.4.2.1. Caso Proposto: Intersecção entre duas Arestas

Inicialmente é necessário determinar se, dadas duas arestas, elas se interceptam. Seja "p" o par de arestas de polígonos distintos e “i” e "f" os pontos inicial e final de cada aresta respectivamente. Desse modo, conseguimos definir 4 pontos: p1i, p1f, p2i, p2f. Com esses pontos, podemos definir a reta que passa por p1i e p1f (P), e a reta que passa por p2i e p2f (Q).

A partir disso, temos:

$$P = p1i + k \cdot (p1f - p1i) \quad \text{(Equação 2.13)}$$

$$Q = p2i + r \cdot (p2f - p2i) \quad \text{(Equação 2.14)}$$

Se as arestas se cruzarem, a solução do sistema acima será do tipo:

$$P = Q \ onde \ 0 < k, r < 1 \quad \text{(Equação 2.15)}$$

Definindo:

$$P1 = p1f - p1i$$

$$P2 = p2f\text{-}p2i$$
$$I12 = p2i\text{-}p1i$$

Podemos simplificar nossa solução para:

$$k = \frac{P2xI12}{P2xP1} \quad \text{(Equação 2.16)}$$

$$r = \frac{P1xI12}{P2xP1} \quad \text{(Equação 2.17)}$$

Determinado se duas arestas se interceptam, podemos determinar o ponto de intersecção entre elas a partir da Equação 2.13.

Desse modo, podemos separar nossa colisão em três casos distintos. Para esses casos, chamaremos o objeto em que duas arestas foram interceptadas de “penetrante” e o objeto em que apenas uma aresta foi interceptada duas vezes de “penetrado” :

- Um par de arestas do objeto A (penetrante) colidindo com uma aresta do objeto B (penetrado).
- Um par de arestas do objeto B (penetrante) colidindo com uma aresta do objeto A (penetrado).
- Um par de arestas do objeto A (penetrante) colidindo comum par de arestas do objeto B (penetrante).

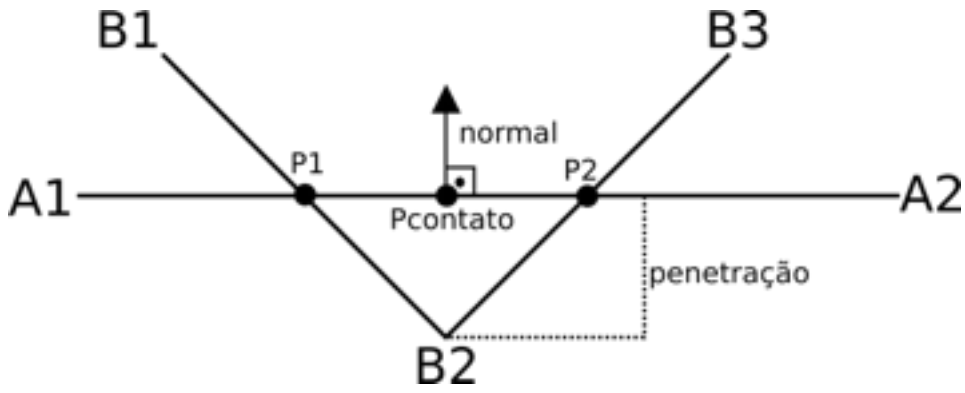

Figura 2.4 - Colisão entre objeto penetrante e objeto penetrado.

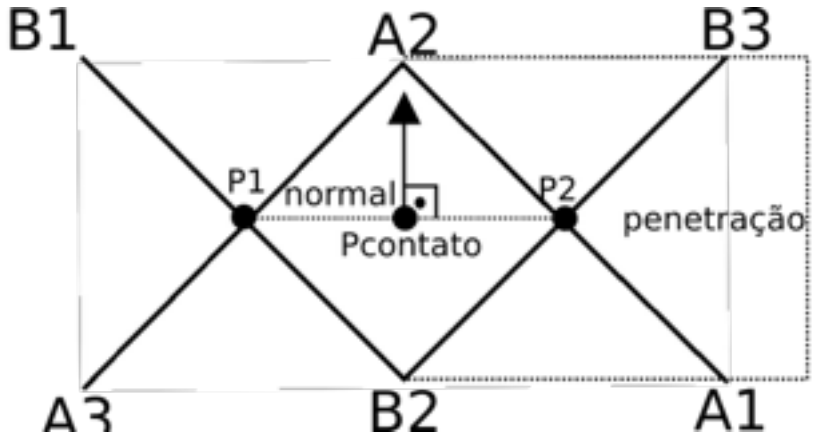

Figura 2.5 - Colisão entre dois objetos penetrantes.

O primeiro e segundo caso (onde um objeto é penetrante e um é penetrado) são análogos em seus cálculos, tendo como única diferença o sinal considerado na penetração do objeto.

Para se encontrar o ponto de contato entre os dois objetos, calculamos apenas a média dos dois pontos de intersecção entre os objetos penetrante e penetrado.

Para se encontrar o vetor normal à colisão entre esses objetos, utilizamos o vetor aresta do objeto penetrado e o rotacionamos em 90 graus no sentido horário. Em seguida normalizamos esse mesmo vetor.

A fim de se determinar a penetração, calculamos o módulo da distância entre o ponto de contato dos dois objetos e o vértice central do projeto penetrante na direção do vetor normal. Esse valor será negativo do ponto de vista do objeto penetrante e positivo do ponto de vista do objeto penetrado.

Para o terceiro caso (onde ambos os objetos são penetrantes), obtemos o ponto de contato entre os objetos como a média dos dois pontos de contato (entre suas arestas) encontrados.

O vetor normal, nesse caso, terá direção perpendicular ao vetor entre os pontos de contato e terá sentido sempre para fora do objeto penetrante (nesse caso, os objetos terão um vetor normal cada, com sentidos opostos e mesma direção).

A penetração será calculada como o módulo da distância entre os vértices centrais de ambos objetos, e terão sempre sinal negativo.

2.4.2.2. Separação de um Polígono Convexo em Triângulos

A partir das equações acima, conseguimos verificar se há colisão entre dois triângulos. Precisamos agora ter a capacidade de separar em triângulos um polígono qualquer. Começaremos pelo caso mais simples: um polígono convexo.

É fácil notar que ao fixarmos um vértice e traçarmos retas a partir dele para todos os outros vértices, conseguiremos atingir nosso objetivo.

2.4.2.3. Separação de um Polígono Côncavo em Triângulos

Para separar um polígono côncavo em triângulos, utilizaremos um algoritmo recursivo em dois passos distintos.

Primeiramente, encontraremos um vértice do polígono que seja interno a ele (Vj), conforme a Figura 2.6. Encontraremos então um triângulo possível de ser formado por esse vértice e outros dois vértices localizados imediatamente antes ou após ele. Verificamos se há algum outro vértice do polígono interno a esse triângulo, caso não haja, removemos esse triângulo e repetimos o processo para o novo polígono formado. Quando o polígono se tornar convexo, executamos o algoritmo descrito na seção anterior.

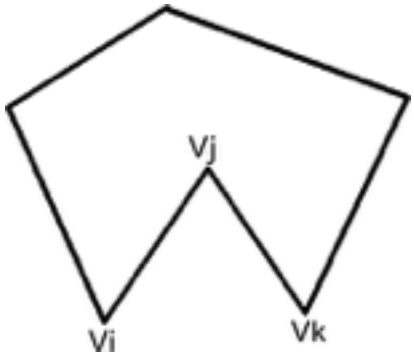

Figura 2.6 - Seleção de vértice interno ao polígono côncavo

Para encontrar um vértice interno ao polígono percorremos todos os seus vértices no sentido anti-horário. O vértice Vj será interno ao polígono se a condição descrita na Equação 2.18 for satisfeita:

$$(Vj-Vi)x(Vk-Vj)<0 \qquad \text{(Equação 2.18)}$$

Onde Vi é o vértice imediatamente anterior a Vj e Vk o vértice imediatamente posterior.

2.4.3. Otimizações na detecção de colisões

Como a detecção de colisões em dois polígonos é um processo demorado devido ao grande número de verificações que devem ser feitas, algumas otimizações são adicionadas de maneira a melhorar a performance do programa.

A fim de realizar essa otimização, separaremos nossa colisão em duas etapas:

A primeira etapa é a “Colisão Grosseira” (*Coarse Collision*). Nela, envolvemos nossos polígonos por circunferências e verificamos se há colisão entre essas circunferências.

A segunda etapa é apenas executada se houve colisão durante a parte “Grosseira”. Ela é chamada de “Colisão refinada” (*Refined Collision*), e executa os métodos descritos acima para colisão entre dois polígonos tais como eles são.

É importante que a circunferência determinada para a "colisão grosseira" seja a menor possível, de modo a minimizar a quantidade de vezes que calcularemos a colisão refinada.

2.4.3.1. Menor círculo envolvente - Algoritmo aleatorizado de Welzl

Para maximizar a eficácia da "colisão grosseira” determinaremos o menor círculo envolvente de cada um de nossos polígonos. Para isso, utilizaremos o Algoritmo de Welzl.

O algoritmo é capaz de calcular o menor círculo envolvente em complexidade de tempo O(n). Ele consiste na seguinte ordem de passos:

- Inicialmente, seleciona-se dois vértices quaisquer do polígono. Determina-se, então, o círculo que possui o segmento de reta entre esses dois pontos como seu diâmetro. Chamamos esse círculo de D(0).
- A cada passo i do algoritmo, calcularemos um círculo D(i). Para se calcular esse círculo, pegamos um vértice aleatório Vi.
- Caso Vi esteja dentro do círculo D(i-1), temos que D(i) = D(i-1). Seguimos então para a próxima iteração do algoritmo, onde um novo vértice será considerado.
- Caso Vi esteja fora do círculo D(i-1), calcularemos um novo círculo D(i) utilizando recursivamente o algoritmo de Welzl para o conjunto de vértices que já consideramos. Nesse caso, é possível se adicionar a seguinte restrição: Se Vi está fora do círculo D(i-1), Vi se encontrará sobre o arco do menor círculo que contém os pontos até então considerados.

- Repetimos os passos acima até que todos os vértices do polígono sejam considerados no cálculo.

2.4.4. O Par de Colisões

Determinadas todas as colisões, podemos gerar uma estrutura de dados conhecida por par de colisões (*collision pair*). Essa estrutura recebe as seguintes informações:

- Referência para cada um dos objetos que colidiram entre si.
- A penetração da colisão.
- O ponto onde ocorreu a colisão.
- O vetor unitário normal à colisão.

É importante notar que, para objetos côncavos, pode haver mais de um par de colisões para os mesmos dois objetos. No entanto, isso não é um problema, visto que podemos apenas resolver cada uma dessas colisões separadamente e o resultado ainda será consistente.

2.5. Resolução da Física em Colisões

Tendo em posse os pares de colisão, somos capazes de determinar a resolução da colisão e o estado final dos objetos envolvidos nela. É importante ressaltar que algumas aproximações que serão adotadas tornam nosso resultado visualmente realista, mas fisicamente não.

Para cada par de colisões, executamos então os seguintes passos:

Dadas as propriedades físicas dos dois objetos que colidiram (velocidades "v1" e "v2", velocidades angulares "ω1" e "ω2", massas "m1" e "m2", momentos de inércia em relação ao centro de massa "I1" e "I2" e os centros de massa "cm1" e "cm2") e as informações do par de colisão (vetor normal à colisão "n", ponto onde ocorreu a colisão "p" e penetração entre os objetos "pen") podemos obter as seguintes relações:

No ponto "p", as velocidades dos objetos serão dadas por:

$$v_{1p} = v_1 + \omega_1 x(p\text{-}cm_1) \qquad \text{(Equação 2.19)}$$

$$v_{2p} = v_2 + \omega_2 x(p\text{-}cm_2) \qquad \text{(Equação 2.20)}$$

Definindo os índices “i" como referente ao estado antes da colisão e “f" como referente ao estado após a colisão e o coeficiente de restituição como “e”, temos:

$$v_{12i} = (v_{1pi} - v_{2pi})\cdot n \qquad \text{(Equação 2.21)}$$

$$v_{12f} = (v_{1pf} - v_{2pf})\cdot n \qquad \text{(Equação 2.22)}$$

$$v_{12i} = -e\cdot v_{12f} \qquad \text{(Equação 2.23)}$$

Desse modo, nosso objetivo é calcular a velocidade angular e a velocidade linear final de todos nossos objetos. Assim sendo, tentaremos determinar o impulso aplicado entre os objetos no ponto definido como ponto de colisão. Definindo o impulso como “j”, temos:

$$v_{1f} = v_{1i} + \frac{j\cdot n}{m_1} \qquad \text{(Equação 2.24)}$$

$$v_{2f} = v_{2i} - \frac{j\cdot n}{m_2} \qquad \text{(Equação 2.25)}$$

$$\omega_{1f} = \omega_{1i} + \frac{(p\text{-}cm_1)x(j\cdot n)}{I_1} \qquad \text{(Equação 2.26)}$$

$$\omega_{2f} = \omega_{2i} - \frac{(p\text{-}cm_2)x(j\cdot n)}{I_2} \qquad \text{(Equação 2.27)}$$

Substituindo as equações 2.24, 2.25, 2.26 e 2.27 em 2.19 e 2.20 e em seguida em 2.23, obtemos, com algumas simplificações:

$$j = \frac{-(1+e)\cdot V_{12i}\cdot n}{\frac{1}{m_1} + \frac{1}{m_2} + \frac{\left[(p\text{-}cm_1)x(n)\right]^2}{I_1} + \frac{\left[(p\text{-}cm_2)x(n)\right]^2}{I_2}} \qquad \text{(Equação 2.28)}$$

Onde:

$$\left[(p\text{-}cm_1)x(n)\right]^2 = \left[(p\text{-}cm_1)x(n)\right]\cdot\left[(p\text{-}cm_1)x(n)\right] \qquad \text{(Equação 2.29)}$$

Com isso, podemos substituir “j” nas equações 2.24, 2.25, 2.26 e 2.27, para obter nossas velocidades lineares e angulares finais.

2.6. Correções de Posição Pós-Colisão

Após calcularmos a física de uma colisão (caso ocorra), é necessário corrigir problemas relacionados à penetração que ocorreu entre os objetos. Existem vários métodos descritos na literatura sobre como resolver esses problemas (Projeção linear, Resolução baseada em velocidade e Projeção não-linear).

Das resoluções, a mais realista é a "Resolução baseada em velocidade", onde após calcularmos as novas velocidades angulares e lineares dos objetos, regredimos o tempo em busca binária até encontrarmos o ponto inicial da colisão e em seguida calculamos a separação entre os objetos a partir daquele ponto. Esse método, no entanto, possui alto custo em tempo de execução para ser implementado.

A resolução que selecionamos foi a menos realista porém mais rápida, a "Projeção linear”, onde cada um dos objetos é recuado na direção da normal da colisão e em sentidos opostos. A fração da penetração que cada objeto irá se deslocar é proporcional ao inverso da massa dos objetos envolvidos na colisão.

2.7. O Ciclo de Resolução da Física

Para terminar a resolução da física, precisamos primeiramente saber quanto tempo o algoritmo demora para executar. O ideal é que se mantenha abaixo de 17ms (60 Hz, *frame-rate* de monitores modernos), porém valores abaixo de 33ms (30 Hz, *frame-rate* de monitores mais antigos) ainda são aceitáveis, visto que o olho humano ainda percebe continuidade para essa faixa de frequências. Desse modo, consideraremos nosso “*passo de tempo*” constante (T).

O ciclo de resolução da física segue os seguintes passos:

- Resolução das forças atuantes nos objetos (gravidade, outras forças aplicadas).
- Resolução da posição e rotação dos objetos baseado nas velocidades angular e linear.
- Resolução das colisões.

- Correções de posição.

Os dois últimos passos foram descritos nas seções anteriores, os outros passos serão tratados nas próximas seções. Ressaltamos que o primeiro passo descrito(resolução das forças atuantes) não foi implementado no projeto.

2.7.1. Resolução de Forças Atuantes

Para cada força atuando nos objetos, calculamos o impulso gerado por ela dado nosso "*passo de tempo*" (no caso de uma aceleração, a transformamos inicialmente em força, utilizando a segunda lei de Newton), e aplicamos esse impulso em nosso objeto.

$$V_f = V_i + F \cdot T \qquad \text{(Equação 2.30)}$$

$$V_f = V_i + \frac{a}{m} \cdot T \qquad \text{(Equação 2.31)}$$

2.7.2. Resolução da Posição e Rotação

Utilizando nosso passo de tempo, podemos definir a nova posição e rotação de nossos objetos, dadas suas velocidades angular e linear. Sendo “P" a posição de um objeto, “R" sua rotação, “v" sua velocidade e “ω” sua velocidade angular, podemos determinar:

$$P_f = P_i + v \cdot T \qquad \text{(Equação 2.32)}$$

$$R_f = R_i + \omega \cdot T \qquad \text{(Equação 2.33)}$$

Capítulo 3

Procedimento Experimental

Discorreremos nesse capítulo sobre os passos que tomamos para implementação dos algoritmos propostos.

Começaremos expondo os principais conceitos do SDK da *Parallella Board*, explicitando suas principais funções.

Em seguida, esclareceremos o modelo de programação adotado comentando as escolhas de implementação feitas.

Por fim, será discorrido sobre as implementações específicas de cada um dos algoritmos propostos.

Todos os códigos implementados estão hospedados publicamente no Github. Seus links podem ser encontrados nos anexos desse trabalho.

3.1. Estrutura de Código para Programação na *Parallella Board*

A *Parallella Board* possui três níveis de organização dos núcleos de seu co-processador (figura 3.1).

O primeiro e mais abrangente é o processador em si, contendo 16 núcleos de processamento, organizados em uma matriz 4x4 começando do processador 0x808.

O segundo nível é um passo intermediário chamado de *Workgroup,* que também é uma matriz de tamanho definido pelo programador. Seu início também é definido pelo programador.

O terceiro nível, e menos abrangente, é o próprio núcleo de processamento. Ao alcançarmos esse nível, podemos enviar para o núcleo um arquivo binário para ser executado.

A fim de se facilitar a programação desse sistema, os núcleos foram re-endereçados de tal maneira em que o primeiro núcleo se encontra na posição (0,0) e o 16º núcleo na posição (3,3).

Em código, precisamos avançar por cada um desses níveis de organização para executarmos nosso código. Os passos que devemos seguir são:

- Declarar uma variável do tipo e_epiphany_t que irá conter nosso *Workgroup*.

- Instanciar nosso Workgroup com uma chamada de e_open. Essa chamada recebe 4 argumentos: O endereço de nossa variável, a linha e coluna referentes ao primeiro núcleo do nosso *Workgroup* e o tamanho horizontal e vertical de nossa matriz. Por exemplo, a chamada e_open(&dev, 1, 0, 2, 3) instanciaria na variável "dev" um *Workgroup* começando no núcleo (1,0) e indo até o núcleo (3,3).
- A partir disso, podemos enviar a um núcleo do nosso *Workgroup* um código binário para ser executado. É importante ressaltar que os núcleos são re-indexados para que o (0,0) represente o primeiro núcleo daquele *Workgroup*. Essa ação pode ser executada através da chamada de e_load. Essa chamada recebe 5 argumentos: A *path* do arquivo binário a ser executado, o endereço da variável “dev”, a linha e a coluna do núcleo que utilizaremos relativas ao *Workgroup* e uma *booleana* indicando se o código deve ser executado assim que carregado. Por exemplo, seguindo o exemplo acima, e_load(“binaryCode.srec”, &dev, 0, 1, E_TRUE), executaria o código “binaryCode" no núcleo (1,1) do processador (o mesmo que o núcleo (0,1) no *Workgroup*).

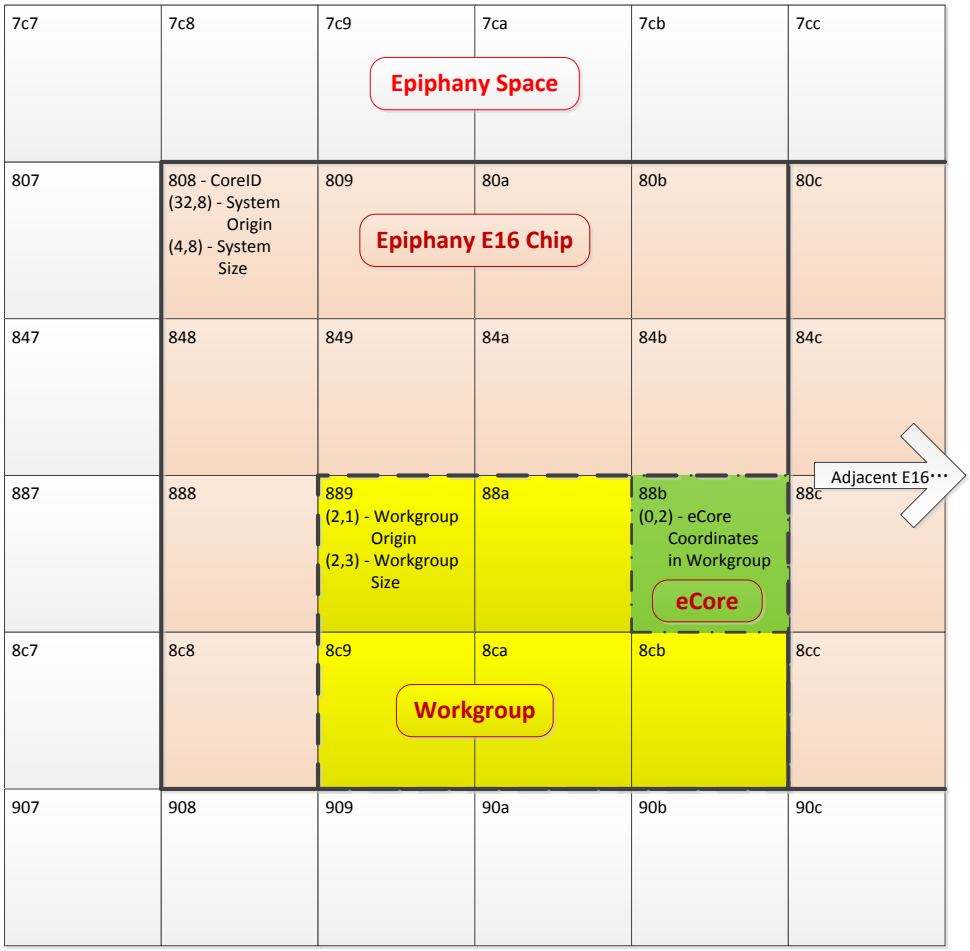

Figura 3.1 - Coordenadas do core na Plataforma e Workgroup (retirado de [3])

Após esses passos, caso queiramos trocar nosso *Workgroup*, precisamos utilizar a chamada e_close em nosso “dev" a fim de liberá-lo para nova utilização (é importante que não haja intersecção entre os *Workgroups* enquanto os núcleos de um deles não terminarem de fato suas tarefas).

Caso um *Workgroup* tenha terminado sua tarefa por completo, é importante chamar a função e_reset_group para permitir aos núcleos desse grupo receberem novos códigos binários.

3.1.1. Acesso à memória dos núcleos do Epiphany

Realizar um acesso à memória dos núcleos do Epiphany (pelos próprios núcleos) apenas requer que um ponteiro aponte para o endereço da memória que queremos acessar.

Esse endereço pode ser dado de duas maneiras: como um endereço de 2 bytes ou como um endereço de 4 bytes.

No caso de um endereço de 2 bytes, o núcleo irá acessar uma memória local no endereço dado. Essa memória deve ser um valor entre 0x0000 e 0x7FFF como já citado anteriormente.

No caso de um endereço de 4 bytes, utilizamos os 2 primeiros bytes para identificar qual o núcleo em que a memória se encontra. Ou seja, acessamos memórias externas ao núcleo. Por exemplo, para se acessar o endereço 0x2015 do núcleo 0x809, utilizamos o endereço 0x80902015.

É importante ressaltar que o acesso a memórias externas ao seu núcleo demora 1 *clock* a mais para cada 1 de distância entre os núcleos envolvidos (Distância de Manhattan).

Explicitamos também que essa memória não é exclusiva dos dados com que trabalharemos, mas também é a em que se encontrará o programa e sua pilha, limitando (na maior parte das vezes) seu espaço útil para os endereços compreendidos entre 0x2000 e 0x7500.

3.1.2. Utilização de estruturas de dados nos núcleos do Epiphany

Muitas vezes, para facilitar a comunicação entre o processador ARM e o processador Epiphany, utilizamos estruturas de dados do tipo "*struct*", porém o alinhamento de memória utilizado por cada processador é diferente. Desse modo, precisamos forçar um mesmo alinhamento (no caso, de 8 bytes) para a estrutura que será compartilhada. Para fazer isso, utilizamos a diretriz "*__attribute__((aligned(8)))*" após a definição da "*struct*".

3.2. Estruturação da FFT *Multicore*

Nosso objetivo é aumentar a eficiência de execução do algoritmo da FFT quando aplicados em uma grande quantidade de dados. Como cada um dos núcleos possui apenas 24576 bytes disponíveis (endereços de memória entre 0x2000 e 0x7FFF), precisamos primeiro avaliar quantos elementos da FFT caberão na memória.

Sabemos que a quantidade de elementos de entrada na FFT é uma potência de 2. Sabemos também que esses elementos são pertencentes ao conjunto dos números complexos. Precisamos então propor um modelo de dados para se representar os números complexos.

Dada a precisão do tipo float (ponto flutuante), modelamos o tipo Complex como uma estrutura contendo 2 elementos do tipo float (um para representar sua parte real e um para representar sua parte imaginária).

```
typedef struct{
    float real;
    float imag;
}Complex;
```

Quadro 3.1 - Estrutura Utilizada na Representação do Conjunto dos Complexos

Sabendo que o tamanho ocupado por um ponto flutuante é de 4 bytes e que o tamanho de uma estrutura é a soma do tamanho de seus elementos, podemos calcular quantos elementos cabem em um único núcleo de processamento.

$$N \leq \frac{24576}{2 \cdot 8}, N = 2^k \qquad \text{(Equação 3.1)}$$

Implicando em:

$$N = 2048 \qquad \text{(Equação 3.2)}$$

Logo, cada núcleo pode processar até 2048 números complexos.

Desse modo, precisamos inicialmente reduzir nosso problema a diversas FFTs de 2048 elementos antes de começarmos a paralelizar nosso processamento.

Observando o pseudo-código apresentado no Quadro 2.1, podemos separar cada chamada de nossa FFT em duas etapas. A primeira consiste em separar o nosso vetor em elementos de índices pares e índices ímpares e a segunda consiste em juntar essas partes recursivamente realizando as operações necessárias.

Abaixo, propomos o método de separação de nosso vetor inicial, para então propor o método de redução de nosso problema.

Dado um vetor de tamanho N (N é uma potência de 2) com índices de 0 a N-1, queremos colocar em sua primeira metade todos os elementos de índice par (ordenados

pelo índice), e na segunda metade todos os elementos de índice ímpar (ordenados pelo índice).

0	1	2	3	4	5	…	N-6	N-5	N-4	N-3	N-2	N-1

Podemos trocar todos os elementos de índice ímpar da primeira metade do vetor com todos os elementos de índice par da segunda metade do vetor, obtendo então:

0	N/2	2	N/2+2	…	N-2	1	…	N/2-3	N-3	N/2-1	N-1

Observando apenas uma das metades de nosso vetor, percebemos que nosso problema agora é passar os elementos que se encontram nos novos índices ímpares dessa metade para o final dela, enquanto os elementos de índice par deve ficar em seu começo. Ou seja, caso apliquemos esse algoritmo recursivamente, conseguiremos obter nossa saída na forma:

0	2	4	6	…	N-2	1	…	N-7	N-5	N-3	N-1

Em código, podemos representar esse algoritmo como:

```
void separateEvenUneven(unsigned* vector,int size){

    if(size==1){
      return;
    }

    int halfSize = size/2;

    for(int i=0; i<halfSize/2; i++){
      unsigned a = vector[2*i+1];
      vector[2*i+1] = vector[2*i+halfSize];
      vector[2*i+halfSize] = a;
    }

    separateEvenUneven(vector, halfSize);
    separateEvenUneven(&vector[halfSize], halfSize);
}
```

Quadro 3.2 - Implementação do Algoritmo de Separação dos Índices Pares e Ímpares

Podemos agora propor nosso algoritmo completo para execução da FFT paralelizada.

Consideraremos, para facilitar nossos cálculos, que estamos utilizando o processador Epiphany 16-core CPU, contando com 16 núcleos de processamento.

Podemos separar em 3 casos a execução de nossa FFT:

- Quando nossa entrada possui menos de 16 elementos.
- Quando nossa entrada possui entre 16 elementos e 2048 elementos para cada core (totalizando 32768 elementos).
- Quando nossa entrada possui mais de 32768 elementos.

Para o primeiro caso, utilizaremos nosso algoritmo de separação em nossa entrada, sem paralelismo, até separar completamente nosso vetor.

Em seguida, enviaremos para N/2 cores cada par de elementos para que seja realizado um passo da segunda etapa da recursão da FFT. Repetiremos esse passo diversas vezes, sempre duplicando o número de elementos enviados e consequentemente dividindo na metade o número de núcleos utilizados, até terminarmos nosso algoritmo por completo.

Para o segundo caso, faremos apenas as primeiras 4 chamadas recursivas do algoritmo de separação, a fim de obtermos 16 seções de nossa entrada.

Em seguida, enviaremos cada uma dessas seções para cada um dos núcleos de processamento e executaremos uma FFT completa nessas seções.

Após isso, iremos realizar os últimos passos da FFT paralelamente até que cada uma das seções de nosso vetor não caiba mais nos núcleos de processamento.

Feito isso, realizaremos os últimos passos da FFT no processador ARM, finalizando a execução de nossa FFT.

Para o terceiro caso, faremos chamadas do nosso algoritmo de separação até cada seção de nosso vetor possuir apenas 2048 elementos.

Em seguida, realizaremos os mesmos passos que executamos para o segundo caso, com excessão de executar os últimos passos da FFT paralelamente.

Com esse algoritmo implementado, calcularemos seu tempo de execução para diversos tamanhos de entrada e compararemos com uma implementação similar do mesmo algoritmo, porém com execução apenas no processador ARM.

3.3. Estruturação da Decodificação de Códigos LDPC *Multicore*

O maior desafio na implementação desse algoritmo na *Parallella Board,* é o gerenciamento de memória e compartilhamento da mesma a fim de ser possível se paralelizar o máximo possível do código.

A matriz H que utilizamos possui tamanho de 9720x16200 bits, ou seja 18,77MiB. Cada um dos 16 núcleos de processamento possui apenas 24KiB úteis de memória, totalizando 384KiB de memória. Porém, a matriz H é esparsa, possuindo apenas 0,037% de seus elementos iguais a 1 (58319 elementos), e o restante igual a 0.

Para reduzir o tamanho dessa matriz, a percorremos linha a linha (em software externo à *Parallella Board)* guardando em uma lista o índice da coluna com elemento igual a 1 e, a cada nova linha da matriz, guardamos na mesma lista um elemento igual a -1.

Por exemplo, para a matriz:

linhas\colunas	**1**	**2**	**3**	**4**
1	1	0	1	0
2	0	0	0	1
3	0	1	0	0
4	0	0	1	0
5	1	1	0	0

Obteríamos a seguinte lista:

1	3	-1	4	-1	2	-1	3	-1	1	2	-1

Cada um dos elementos dessa lista ocupa 2 bytes, que é o menor tamanho de estrutura capaz de guardar números até 16200. Dessa maneira, nossa matriz passa a ocupar apenas 132,88 KiB de memória (aproximadamente 7% do tamanho original).

Dividimos essa matriz entre todos os núcleos em locais de memória compartilhada a fim de ocupar o mínimo de espaço em cada um deles com essa matriz, além de homogeneizar o tempo de acesso à matriz de cada núcleo do Epiphany. Dessa forma, o núcleo com mais elementos terá 8,31 KiB ocupados de memória por essa matriz.

Nossa segunda preocupação em relação à memória, é tornar possível cada núcleo conter os seguintes elementos: nossa entrada (o vetor r), um pedaço de nossa matriz, e um contador para cada elemento de nossa matriz.

No caso limite, o maior pedaço de nossa matriz ocupa 8,31KiB de memória. Nossa entrada ocupa 16200 bits, ou seja, 1,98KiB.

Como cada coluna de nossa matriz possui no máximo apenas 12 elementos iguais a 1, nosso contador precisa ser capaz de armazenar valores de -12 a 12, logo, são necessários 5 bits para cada elemento de nosso contador. Separamos esse contador em dois vetores: um vetor de bits para representar o sinal e um vetor de bytes, onde cada byte representa dois elementos do nosso contador. Dessa forma, o contador irá ocupar 10125 bytes, ou seja 9,88 KiB.

No total ocupamos aproximadamente 20,17KiB de memória em cada núcleo (dos 24KiB disponíveis).

3.4. Estruturação da *Framework* de simulação de física para a *Parallella Board*

Abordaremos toda a implementação dos algoritmos relacionados à geometria e física da *Framework* além da renderização dos objetos em equipamento externo à *Paralela Board*.

3.4.1. Representação dos objetos físicos

Primeiramente, citaremos como representamos os objetos de física que serão tratados em nossos algoritmos.

Os objetos foram modelados geometricamente a partir de dois referenciais, sendo um deles um referencial local e o outro um referencial global (inercial). Além disso, cada objeto possui um ponto (no referencial local) que representa o centro de rotação do objeto. Outras duas informações estão presentes também, sendo elas o centro do menor círculo envolvente(no referencial local) e seu raio.

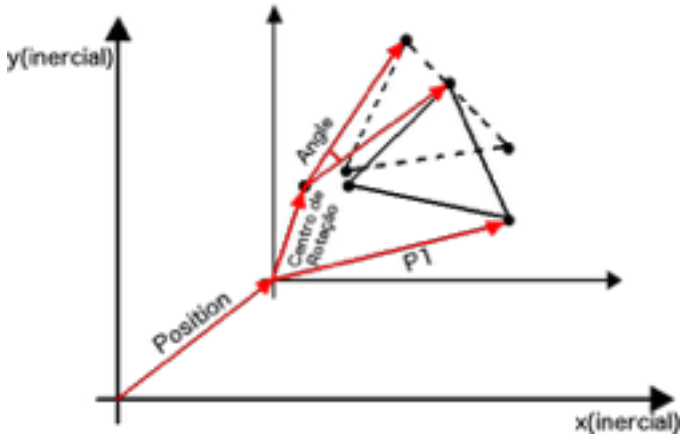

Figura 3.2 - Relações das propriedades do objeto com os referenciais local e global

Desse modo, podemos executar os seguintes passos para converter um ponto do referencial local para o referencial inercial:

$$P_{aux} = P_{local} - RC \qquad \text{(Equação 3.3)}$$

$$P_{rot} = \begin{bmatrix} \cos(Angle) & -sen(Angle) \\ sen(Angle) & \cos(Angle) \end{bmatrix} \cdot P_{aux} \qquad \text{(Equação 3.4)}$$

$$P_{inerc} = P_{rot} + RC + Position \qquad \text{(Equação 3.5)}$$

Onde RC é o centro de rotação do objeto.

Do ponto de vista da física, cada objeto possui como parâmetros sua velocidade, sua massa e seu momento de inércia (consideramos o seu centro de massa coincidente de seu centro de rotação, por simplicidade).

Em *Frameworks* comerciais de física bidimensional, é comum se guardar o inverso do momento de inércia e o inverso da massa dos objetos ao invés dos valores normais. Isso se deve ao fato de podermos representar objetos de massa infinita e momento de inércia infinita. Em nosso projeto, também representaremos dessa maneira, pois, além das vantagens já citadas, em cálculo de colisões, só nos importa o inverso dessas variáveis, e, mantê-las dessa maneira, aumentaria a otimização de nosso programa, visto que os núcleos da *Epiphany* não possuem suporte para divisão em hardware.

3.4.2. Renderização dos resultados

A fim de se obter um *Feedback* visual da aplicação, enviamos através da rede os estados dos objetos simulados para um *Ipad*, onde eles serão renderizados.

Utilizamos um par de *sockets* TCP (*server-client*) unidirecionais, onde a *Parallella Board* (como servidor) envia constantemente dados para o *Ipad* (como cliente).

Os dados são enviados em forma de duas mensagens distintas: uma para se criar um objeto novo e uma para se alterar um objeto já existente.

As mensagens possuem as seguintes informações:

Mensagem de objetos novos:

- ID: Inteiro - Identificador do objeto.
- Position: Par de pontos flutuante - A posição do objeto.
- Points: Vetor com pares de pontos flutuantes - As coordenadas dos vértices.
- Rotation: Ponto flutuante - A rotação do objeto.
- RotationCenter: Par de pontos flutuante - O centro de rotação do objeto.
- Type: Inteiro - 0 para círculo, 1 para polígono.
- Radius: Ponto flutuante - O raio do objeto, caso seja um círculo.

Mensagem de atualização de objetos:

- ID: Inteiro - Identificador do objeto.
- Position: Par de pontos flutuante - A nova posição do objeto.
- Rotation: Ponto flutuante - A nova rotação do objeto.

Todos os dados são calculados com base nos referenciais descritos em **3.3.1**.

Para o envio dos dados primeiramente os serializamos para uma *string* no padrão JSON (*JavaScript Object Notation*). Em seguida, calculamos o tamanho da mensagem (em bytes) e enviamos primeiramente o tamanho calculado e em seguida a mensagem em si.

Apesar de o lado cliente da aplicação estar implementado, ele não foi integrado com o projeto em si. Seus códigos podem ser encontrados nos anexos.

3.4.3. Paralelização do Cálculo de Colisões

Antes de executar o código paralelizado, executaremos algumas etapas de *setup* de nosso sistema nos núcleos ARM da *Parallella Board*.

Inicialmente, criamos uma estrutura do tipo *PhysicsEngine*, ela serve unicamente para guardar um vetor de objetos físicos (*PhysicsObject*) e um contador para o número de elementos desse vetor, a fim de facilitar ao programador o gerenciamento da quantidade de objetos possíveis de se calcular paralelamente.

Após adicionarmos todos os objetos que queremos observar em nossa *PhysicsEngine*, comandaremos-a a calcular o menor círculo envolvente de cada um dos objetos e a dividi-los em triângulos.

Feito isso, distribuímos os objetos nos núcleos da *Epiphany*. Cada núcleo possui um vetor de objetos contendo um máximo de 4 objetos (definido no projeto, dadas estimativas dos tamanhos das estruturas utilizadas). Desse modo, o objeto de índice “Ie” localizado no n-ésimo núcleo da *Epiphany* representa o objeto de índice “Ia” localizado no core ARM de acordo com a seguinte relação:

$$I_a = I_e \cdot 16 + N \qquad \text{(Equação 3.6)}$$

A partir disso, precisamos garantir que o ciclo de execução da física (descrito na seção 2.7.) seja executado de maneira síncrona pelos núcleos *Epiphany*. Para permitir isso, criamos 4 estados de execução de cada núcleo (*OnVelocity* - atualização de todos os dados antes das colisões; *EndVelocity* - após atualização dos dados antes das colisões; *OnCollision* - Cálculo das colisões; *EndCollision* - após cálculo das colisões).

Quando um núcleo chegar ao estado “*EndVelocity*” ou “*EndCollision*”, ele iniciará um ciclo que verificará se todos os núcleos se encontram no mesmo estado que ele. Em caso positivo, ele avançará para o próximo estado e forçará um núcleo subsequente a também mudar, executando uma reação em cadeia e avançando de maneira síncrona os estados dos núcleos.

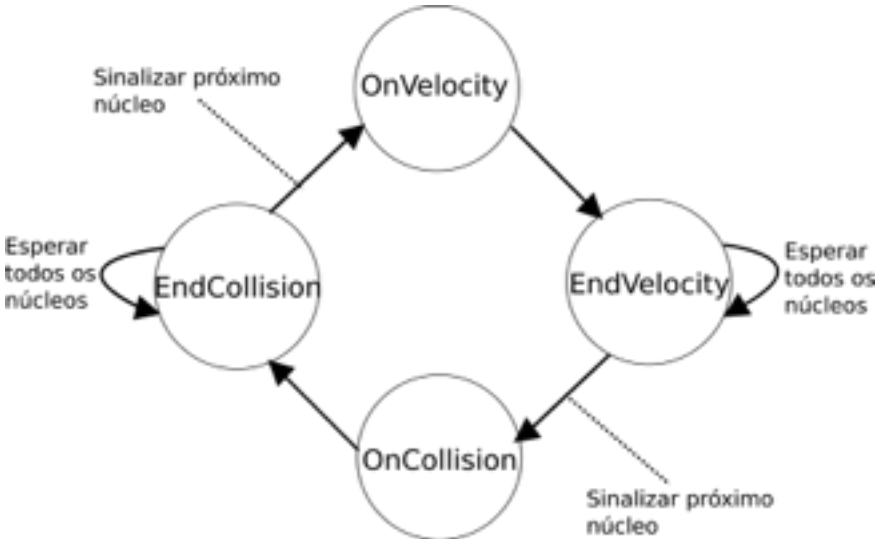

Figura 3.3 - máquina de estados para sincronização dos núcleos do *Epiphany*.

Após isso, ativamos os núcleos da Epiphany para realizarem os cálculos da física. Ao terminar o cálculo, os núcleos criam estruturas do tipo "*Frame*". Essas estruturas representam o estado (posição e rotação) dos objetos físicos e possuem como parâmetros adicionais o índice do objeto que representam no núcleo ARM e a numeração da frame que aquele estado pertence.

Nosso núcleo ARM irá executar (em ciclo infinito) uma rotina de ler as "*Frames*" geradas pelos núcleos *Epiphany*. Essa rotina é completamente assíncrona com os cálculos de física e pode, porventura, coletar frames calculadas em momentos distintos do algoritmo. Nesse caso, apesar de levar a uma inconsistência visual, é possível manter uma velocidade de processamento maior nos núcleos *Epiphany*, visto sua independência de sincronização com o processador ARM.

Capítulo 4

Resultados e Discussões

Neste capítulo mostraremos os resultados obtidos com a implementação de nossos algoritmos e discorreremos sobre as principais dificuldades encontradas para implementação dos mesmos.

4.1. Comparação dos Resultados da FFT em Suas Implementações *Singlecore* e *Multicore*

Durante a implementação do algoritmo da FFT alguns problemas foram encontrados na utilização dos núcleos do Epiphany.

Primeiramente, é importante citar que quando os erros que serão discutidos ocorriam, era gerada uma inconsistência na execução do código, impedindo o gerenciamento de recursos dos núcleos e consequentemente travando o algoritmo em estados inconsistentes.

O primeiro problema encontrado foi na atribuição de estruturas. Dado um ponteiro para estrutura (Complex* a) e uma outra estrutura qualquer (Complex b), quando tentávamos atribuir diretamente o conteúdo do ponteiro na variável, a fim de copiá-la, um estado inconsistente do código era gerado (por exemplo: b=a[0]). Para resolver isso fizemos uma cópia valor a valor dos elementos da estrutura. Esse erro também impedia chamadas de funções que recebia estruturas como argumento. Para contornar esse segundo problema, precisamos passar os argumentos como referências (ponteiros).

O segundo problema encontrado foi na utilização de funções da biblioteca *math.h*. Em qualquer chamada de funções dessa biblioteca, a mesma inconsistência era gerada. A fim de contornar esses problemas, como utilizamos apenas as funções seno e cosseno dessa biblioteca, utilizamos uma série de Taylor de graus 5 e 8 respectivamente.

O terceiro problema foi durante a execução de divisões, caso as utilizássemos, a mesma inconsistência era gerada. Para evitar isso fizemos uso de uma estrutura switch-case para determinar as divisões que necessitaríamos fazer e substituímos nossas divisões por multiplicações.

É importante citar também que após a ocorrência desses erros, os núcleos de processamento se mantinham presos em seus estados, mesmo quando o programa principal já havia sido terminado. Para se resetar os núcleos, era necessária a execução de e-reset no terminal. Desse modo, acrescentamos essa chamada em nosso script de execução do programa.

Resolvidos os erros citados, pudemos mensurar o desempenho da execução da FFT em múltiplos núcleos da Epiphany e apenas no ARM.

Os resultados estão apresentados abaixo na figura 4.1.

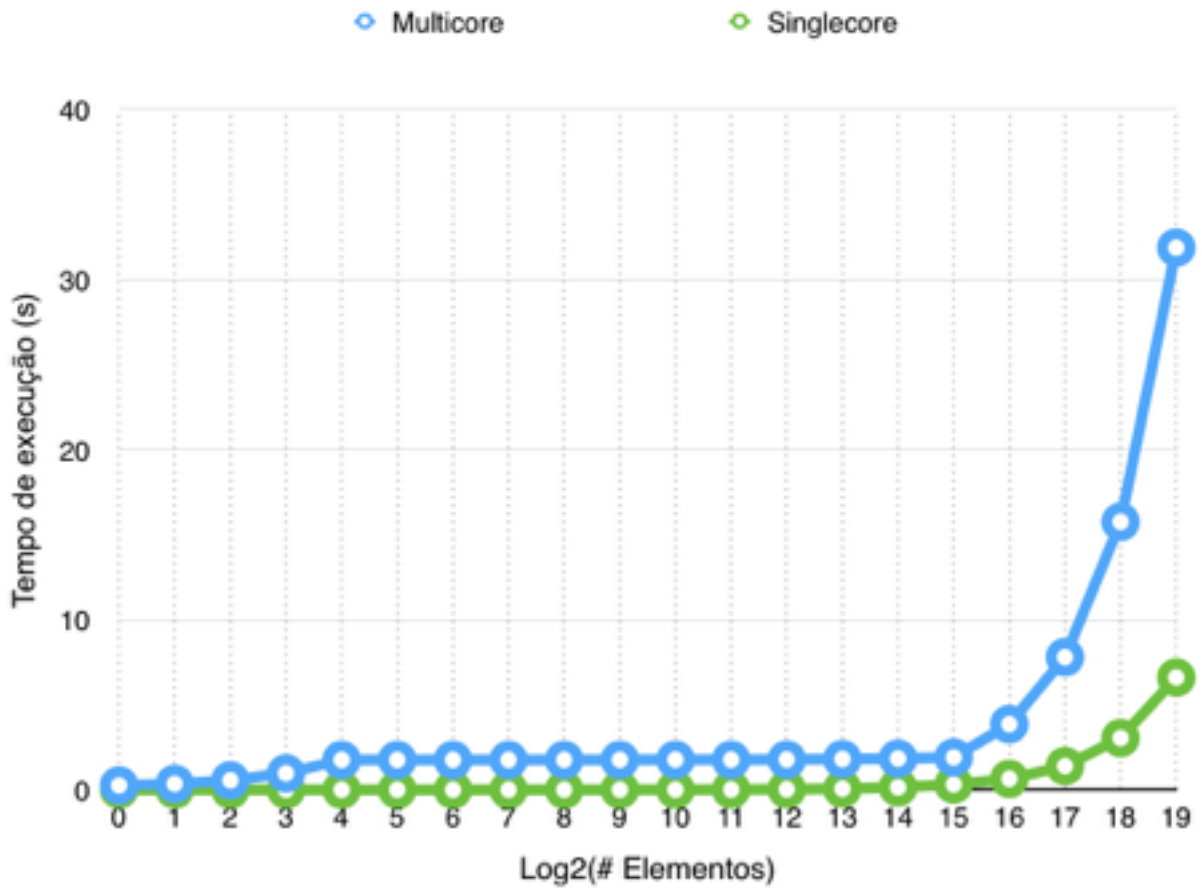

Figura 4.1 - Comparação dos Tempos de Execução da FFT.

Percebemos que nossa implementação da FFT obteve um desempenho muito inferior ao esperado. Analisando nosso código, podemos atribuir esse baixo desempenho a dois fatores. O primeiro fator é a grande movimentação de memória entre a RAM do ARM e a CACHE da Epiphany, visto que movimentação de memória é um processo demorado. O segundo fator é: após toda interação resetarmos o núcleo que terminou sua execução e reiniciarmos todo o processo de escrita do programa na memória do núcleo.

Percebemos também, durante a execução do programa que o primeiro núcleo de processamento consegue finalizar a execução de sua parte da FFT antes mesmo de todos os outros núcleos terem os dados necessários copiados para suas respectivas memórias.

Futuramente podemos mensurar os pontos específicos do código onde ocorrem os maiores gastos de tempo e tentar otimizá-los para obter um melhor resultado do algoritmo.

4.2. Resultados da Implementação da Decodificação LDPC *Multicore*

Durante a implementação do algoritmo tentamos corrigir alguns dos problemas que foram encontrados durante a implementação da FFT *multicore.* Inicialmente, ao invés de alocarmos um *Workgroup* para cada núcleo do Epiphany, alocamos apenas um *Workgroup* para todos os núcleos e continuamos tratando cada núcleo individualmente.

Além disso reduzimos a quantidade de sinais de controle usados para sincronização do processador ARM e do co-processador do Epiphany de 5 para 1. Desse modo reduzimos o *overhead* de múltiplos acessos à memória.

O código utilizado pelos núcleos do Epiphany foi colocado dentro de um *loop* infinito (*while(1)*) de modo a não ser necessário carregá-lo novamente na memória a cada vez que ele termina de ser executado reduzindo ainda mais o *overhead*.

Com essas pequenas alterações conseguimos observar que, diferentemente da implementação da FFT, os núcleos do Epiphany não terminam suas execuções tão rápido quanto terminamos de escrever os dados nos outros núcleos.

Contudo, alguns erros ainda persistem na implementação e precisam ser melhor estudados a fim de encontrarmos a melhor estrutura para implementação de códigos na *Parallella Board*.

Notamos que para uma mesma entrada em cada um dos núcleos do Epiphany, com o mesmo código em todos eles, alguns dos núcleos nunca terminavam sua execução. Isso se manteve até resetarmos a *Parallella Board* e colocarmos uma refrigeração mais potente nela.

Realizamos 3 medições de tempo de execução com os 16 núcleos em funcionamento e uma mesma entrada. A média de tempo obtida nessa execução foi de 25.88 segundos. Assim, obtivemos 1.22KiB/s de transmissão de dados, ou 78B/s/núcleo.

4.3. Resultados da Implementação da *Framework* de Colisões

Inicialmente reforçaremos o que já foi citado na seção de não ser possível se passar uma "*struct*" a uma função a não ser por referência (ponteiro).

Além disso, nessa implementação, o programa (após compilação) ficou muito extenso, tomando todos os endereços de 0x0000 a 0x6100 de cada núcleo, nos deixando pouco espaço útil para os "*PhysicsObjects*" e "*Frames*". Porém, como os únicos métodos recursivos utilizados não são executados nos núcleos do "*Epiphany*", a pilha utilizada pelo programa é pequena e de tamanho máximo fixo, não ocupando muito de nosso espaço útil.

A fim de se estimar de maneira confiável o *Frame-Rate* que conseguimos alcançar, deixamos os núcleos do *Epiphany* calcular livremente as colisões dados apenas dois objetos em duas situações distintas - uma onde eles colidiriam e outra onde não. Em paralelo a isso, temos nosso ARM calculando o tempo de cada ciclo completo de coleta de *Frames* e dividindo a quantidade de frames já calculadas pelo tempo em que demoramos para calculá-las. Desse modo obtivemos duas medidas distintas de frames por segundo, uma quando pelo menos dois objetos estão próximos o suficiente para se detectar uma "colisão grosseira" e outra quando os objetos estão mais afastados. No primeiro caso, obtivemos uma medida de aproximadamente 1.85 fps (0.54 segundos por frame), que é um valor muito abaixo do esperado, no segundo caso, no entanto, obtivemos um valor próximo de 80 fps (0.0125 segundos por frame), ou seja, um valor acima da frequência de um monitor moderno.

A partir desses resultados, pudemos estabelecer um passo de tempo de 0.54 segundos entre duas de nossas frames.

4.4. Discussões Gerais

Apesar dos resultados terem sido abaixo do esperado, alguns pontos importantes podem ser notados quanto a utilização da *Parallella Board*.

É necessário uma familiaridade maior com a arquitetura do sistema a fim de se conseguir utilizar da melhor maneira o possível os recursos oferecidos pelo *hardware*. Um exemplo disso é a utilização do DMA entre os núcleos do Epiphany, que, nesse trabalho, não foram utilizados. Sua utilização provavelmente reduziria o acesso a memórias externas a cada um dos núcleos, possibilitando reduzir o overhead do acesso através de cópias de blocos maiores de memória.

Atém disso, em se tratando dos resultados obtidos com a FFT, uma abordagem com menos sinais de controle e mais independência dos núcleos Epiphany em relação aos núcleos ARM (como a estrutura utilizada nas outras duas aplicações) poderia trazer resultados melhores.

Atém disso, é importante lembrar que devemos tomar precauções para que os dados transmitidos entre os núcleos ARM e Epiphany sejam compatíveis (como pudemos observar com os dados de precisão dupla e alinhamento de estruturas).

Capítulo 5

Conclusão

Ao longo do desenvolvimento desse trabalho, notamos que é possível obter tempos de execução razoáveis para as aplicações propostas, porém é necessária uma maior familiaridade com a arquitetura trabalhada.

Durante as pesquisas encontramos outras implementações de FFT para essa arquitetura, porém grande parte dos códigos foram implementados em nível mais baixo do que C (Assembly). Essas implementações se mostraram mais eficazes, provavelmente por maior otimização de algumas etapas dos cálculos utilizados.

Além disso, é importante citar que nenhuma *flag* de otimização foi utilizada em nenhum dos compiladores utilizados (*gcc* e *e-gcc*).

Por fim, podemos dizer que o uso de paralelismo computacional pode ser uma opção ao uso de FPGAs para alguns algoritmos, porém o desenvolvimento desse tipo de código ainda precisa de uma curva de aprendizagem grande, que é a familiarização com a arquitetura utilizada.

Referências Bibliográficas

[1] Weisstein, Eric W.; "Fast Fourier Transform." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/FastFourierTransform.html

[2] Adapteva, Lexington; “Epiphany Architecture Reference”. http://www.adapteva.com/docs/epiphany_arch_ref.pdf

[3] Adapteva, Lexigton; “Epiphany SDK Reference”. http://www.adapteva.com/docs/epiphany_sdk_ref.pdf

[4] Ta, Tuan; “A Tutorial on Low Density Parity-Check Codes”. http://www.ece.umd.edu/~tta/resources/LDPC.pdf

[5] Taylor, Adam; "Parallella Chronicles Part Six: Moving Multiple Bytes Simply”. https://www.parallella.org/2015/04/05/parallella-chronicles-part-six-moving-multiple-bytes-simply/

[6] ECMA International; “The JSON Data Interchange Format”. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[7] Fernandes, Eraldo R.; “Algoritmos Para o Menor Círculo Envolvente”. http://www.eraldoluis.pro.br/download/mec/algoritmos.htm

[8] Heng, Edison; "Network and Socket programming tutorial in C". http://www.codeproject.com/Articles/586000/Networking-and-Socket-programming-tutorial-in-C

[9] Millington, Ian; “Game Physics Engine Development”. San Francisco: Morgan Kaufmann Publishers, 2007. 456 p.

[10] Neumann, Erik; “Rigid Body Collisions Physics Simulation”. http://www.myphysicslab.com/collision.html

Anexos

[1] Todos os códigos utilizados na FFT desse trabalho se encontram em https://github.com/otavionettozani/ES951-TG1-FFT. O identificador do último *commit* utilizado até a finalização deste documento é "1f1c9fe6ecf1f3006636c7708f66e8c03e461d5f" na *branch* “master”.

[2] Todos os códigos utilizados na Decodificação de Códigos LDPC desse trabalho se encontram em https://github.com/otavionettozani/ES951-TG1-LDPC. O identificador do último *commit* utilizado até a finalização deste documento é "b27a198ca7e5592d49c2e957fb3a6c3398c3dadf" na *branch* “master".

[3] Todos os códigos utilizados na Framework de Simulação de Física desse trabalho se encontram em https://github.com/otavionettozani/E-PhysicsCollider-ES952. O identificador do último *commit* utilizado até a finalização deste documento é "e04fec6cf96e0e10a6bc232ab51edd8c19b8f7b1" na *branch* “master".

[4] Todos os códigos utilizados no auxiliar de renderização para Ipad desse trabalho se encontram em https://github.com/otavionettozani/E-PhysicsRenderer-Ipad. O identificador do último *commit* utilizado até a finalização deste documento é "bc96234e879dd1425885364f38035d34f804f9db" na *branch* “master".