Melhoria de desempenho da máquina virtual Java na plataforma Cell B.E.

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Raoni Fassina Firmino e aprovada pela Banca Examinadora.

Campinas, 6 de setembro de 2010.

Rodolfo Jardim de Azevedo (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Firmino, Raoni Fassina
F518m Melhoria de desempenho da máquina virtual Java na plataforma Cell B.E./Raoni Fassina Firmino-- Campinas, [S.P. : s.n.], 2010.

Orientador : Rodolfo Jardim de Azevedo .
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Java (Linguagem de programação de computador). 2.Arquitetura de computadores. 3.Multi core . I. Azevedo, Rodolfo Jardim de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Java virtual machine performance improvement in Cell B.E. architecture

Palavras-chave em inglês (Keywords): 1. Java (Computer program language). 2. Computer architecture. 3. Multi core.

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Rodolfo Jardim de Azevedo (IC – UNICAMP)
Prof. Dr. Sérgio Takeo Kofuji (LSI – USP)
Prof. Dr. Sandro Rigo (IC - UNICAMP)

Data da defesa: 29/06/2010

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 29 de junho de 2010, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Sérgio Takeo Kofuji
LSI / USP

Prof. Dr. Sandro Rigo
IC / UNICAMP

Prof. Dr. Rodolfo Jardim de Azevedo
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Melhoria de desempenho da máquina virtual Java na plataforma Cell B.E.

Raoni Fassina Firmino[1]

Maio de 2010

Banca Examinadora:

- Rodolfo Jardim de Azevedo (Orientador)
- Sandro Rigo
  IC–UNICAMP
- Sérgio Takeo Kofuji
  LSI–USP
- Paulo Cesar Centoducatte (Suplente)
  IC–UNICAMP
- Ivan Luiz Marques Ricarte (Suplente)
  FEEC–UNICAMP

[1]Suporte financeiro de: Bolsa do CNPq (processo 134617/2007-0) 2007, Bolsa FAPESP (processo 2007/02248-9) 2007–2009.

Resumo

Esta dissertação concentra-se no atual momento de transição entre as atuais e as novas arquiteturas de processadores, oferecendo uma alternativa para minimizar o impacto desta mudança. Para tal utiliza-se a plataforma Java, que possibilita que o desenvolvimento de aplicações seja independente da arquitetura em que serão executadas. Considerando a arquitetura Cell B.E. como uma nova plataforma que promete desempenho elevado, este trabalho propõe melhorias na Máquina Virtual Java que propiciem um ganho de desempenho na execução de aplicações Java executadas sobre o processador Cell.

O objetivo proposto é atingido por meio da utilização do ambiente disponível na própria plataforma Java, o *Java Native Interface* (JNI), para a implementação de interfaces entre bibliotecas nativas construídas para a arquitetura Cell – com a intenção de obter o máximo desempenho possível – e as aplicações Java. É proposto um modelo para porte e criação das interfaces para bibliotecas e mostra-se a viabilidade da abordagem proposta através de implementações de bibliotecas selecionadas, consolidando a metodologia utilizada. Duas bibliotecas foram portadas completamente como prova de conceito, uma multiplicação de matrizes grandes e o algoritmo RC5. A multiplicação de matrizes obteve um desempenho e escalablidade comparável ao código original em C e em escala muitas vezes superior ao código JNI para arquitetura x86 a ao código Java executando em arquiteturas x86 e Cell. O RC5 executou apenas aproximadamente 0,3 segundos mais lento que o código C original (perda citada em segundos pois se manteve constante independente do tempo levado para as diferentes configurações de execução).

Abstract

This dissertation focuses on the present moment of transition between the current and new processor architectures, offering an alternative to minimize the impact of this change. For this, we use the Java platform, which enables an architecture-independent application development. Considering the Cell BE architecture as a new platform that promises high performance, this paper proposes improvements in the Java Virtual Machine that provide performance gains in the execution of Java applications running on the Cell processor.

The proposed objective is achieved through the use of the environment available on the Java platform itself, the Java Native Interface (JNI), to implement interfaces between native libraries built for the Cell architecture – with the intention of obtaining the maximum possible performance – and the Java applications. It is proposed a model to port and build interfaces to libraries and it shows the viability of the proposed methodology with the implementation of selected libraries, consolidating the used methodology. Two libraries were completely ported as proof of concept, a multiplication of large matrices and a RC5 algorithm implementation. The matrices multiplication achieved scalability and performance in the same basis as the native implementation and incomparable with JNI implementation targering x86 architecture and Java implementation running in x86 and Cell architectures. The RC5 was just 0.3 seconds slower than the original C code (the loss is put in seconds since it was constant, independent of the execution time taken by different configurations of execution).

Agradecimentos

Eu gostaria de agradecer ao Instituto de Computação e o Laboratório de Computação que ambiente e recursos que tornaram possível a elaboração dessa tese. A Unicamp pelos serviços oferecidos ao seu corpo discente, em especial ao CECOM e ao SAPPE, que me tanto foram úteis nesse período. As agencias de fomento CNPq e FAPESP que financiaram essa pesquisa. A Sílvia e a Tânia do SAPPE pela imensa ajuda e assistência, sem a qual não teria atravessado por todo esse caminho e não teria chegado até aqui. Ao Marcão e sua cantina que foram de vital importância para prover as longas estadias no laboratório. Aos meus colegas de laboratório que tornaram os dias e noites de pesquisa mais divertidos, entre eles, Ecco, João, Auler, Piga, Yang, Nicácio, Faveri, Baldassin, Klein, Cardoso, George e muitos outros que aqui peço desculpa por não incluir os nomes. A turma de grafos, pela companhia nos dias e noites de estudos. Aos meus amigos Rachid, Elaine, Tarcísio, Crístian, Glauber e André, que fiz dentro e fora do instituto de computação e sempre foram prestativos, me apoiaram e estiveram comigo não somente nos momentos de diversão mais também nos momentos difíceis.

Agradeço a minha família, que sempre esteve para mim quando precisei, pacientemente lidando com minha índole e personalidade difíceis de se conviver. Por último, mas não menos importante ao meu orientador, Prof. Dr. Rodolfo Jardim de Azevedo, por seu voto de confiança, otimismo, apoio e paciência constantes, por acreditar em mim quando eu mesmo não acreditei e por se preocupar.

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) que financiaram essa pesquisa durante o mês de agosto de 2007 e no período entre setembro de 2007 e julho de 2009 respectivamente.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Códigos

Capítulo 1

Introdução

Com o limitante do aquecimento dos processadores atuais, inviabilizando a ampliação da frequência de operação, novas arquiteturas têm sido criadas de forma a aumentar o desempenho de execução de códigos sem aumentar a frequência e, consequentemente, o calor dissipado. Diversas abordagens surgiram para tentar contornar esses limites, a maioria esta relacionada com a replicação de núcleos do processador, gerando as arquiteturas *multicores*. Enquanto empresas como a Intel, AMD e IBM desenvolveram versões de seus processadores puramente replicando as unidades de processamento, outras como a SUN tentaram inovar na forma de integração através do processador Niagara e Niagara II, com vários recursos compartilhados entre os diversos processadores. Outro processador que se diferencia da abordagem convencional foi desenvolvido pelo consórcio STI, formado pela IBM, Toshiba e Sony, denominado Cell. O processador Cell implementa uma arquitetura *multicore* heterogênea, formado por um PowerPC e 8 unidades de processamento vetorial, que possuem acesso restrito à memória principal. Esse é o processador utilizado dentro do console Playstation 3 da Sony, além de várias máquinas de alto processamento produzidas pela IBM e suas parceiras.

Se, por um lado, é possível conseguir um grande poder computacional com uma arquitetura como a do processador Cell, por outro lado é difícil programá-lo, como será visto mais a frente. Assim, novas metodologias e níveis de abstração são necessários para se conseguir tirar proveito de todas as características de alto processamento em um espaço mais curto de tempo. Uma opção preferencial para essas situações é utilizar o código legado com as características de processamento das máquinas novas. Essa não é uma tarefa simples, principalmente quando se trata de níveis de abstração mais baixos como no caso da linguagem de programação C. No entanto, subindo um pouco o nível de abstração é possível esconder do programador várias das APIs de programação de uma nova arquitetura e permitir que ele tire proveito das novas funcionalidades seguindo o modelo de programação que ele já está acostumado. Dentro desse escopo

entra a execução de códigos Java em processadores Cell.

A plataforma Java provê, através da máquina virtual Java, *Java Virtual Machine* (JVM), um nível de abstração alto o suficiente para permitir ao desenvolvedor de ferramentas esconder toda a API de programação do Cell, fazendo com que o código legado tire proveito de todo o poder de processamento disponível apenas através da alteração da máquina virtual. Para tal a plataforma dispõe de uma interface, a *Java Native Interface* (JNI), que possibilita a comunicação e interação entre código Java e código nativo e assim criar bibliotecas que proveem funcionalidades não presentes, ou desempenho crítico dentro da aplicação Java, através de código nativo da plataforma Cell para o programador de forma transparente. O uso dessa abordagem para diminuir a distância entre os paradigmas de programação já estabelecidos na industria e os trazidos pela implementação e aplicação de novas arquiteturas, em especial a arquitetura Cell, são os objetivos buscados por essa dissertação.

Neste trabalho é proposto o reuso de bibliotecas nativas já existentes para a plataforma Cell para uso em aplicações para a plataforma Java através da criação de uma interface JNI para cada biblioteca implementando uma camada de abstração que permite a integração entre as duas plataformas.

Inicialmente a pesquisa dessa dissertação, buscando o aumento do desempenho da máquina virtual Java sobre a arquitetura Cell, focou-se no porte da mesma para a plataforma. O porte o Compilador *Just-In-Time* para o SPE foi contemplado de forma a deslocar parte da carga de trabalho da máquina virtual. Dadas as características do processador Cell, adaptar uma implementação já existente, que não prevê o porte para arquiteturas com sistema de memória distribuída ou arquiteturas heterogêneas, se mostrou pouco viável. A melhor forma de aumentar o desempenho da JVM através de seu porte seria a escrita completa de uma JVM específica para o processador Cell. Assim, novas abordagens foram buscadas resultando na abordagem contemplada nessa dissertação.

Este texto é organizado da seguinte forma: O capítulo 2 é explanado sobre os tópicos ou tecnologias fundamentais de base para este projeto, a Maquina Virtual Java (JVM), o seu compilador Java *Just-In-Time*, a *Java Native Interface* (JNI), a arquitetura Cell Broadband Engine, a primeira implementação do processador Cell e os trabalhos já desenvolvidos na área desse projeto; O capítulo 3 apresenta um detalhamento da abordagem proposta; O capítulo 4 destaca o desenvolvimento prático do projeto tema desta tese, os pontos a serem considerados, as dificuldades encontradas e discute os resultados atingidos; Por fim, o capítulo 5 finaliza a discussão desse projeto retomando os objetivos propostos e os resultados atingidos.

Capítulo 2

Trabalhos relacionados

Sendo esse projeto focado nas plataformas Java e Cell, uma introdução as duas arquiteturas é feita nesse capítulo, apresentando suas principais características. Por último, uma revisão das pesquisas sobre Java para a arquitetura Cell e análise do estado da arte são apresentadas.

2.1 Java

Desde seu surgimento, antes de 1995, a linguagem Java vem ganhando importância como um padrão para linguagem de programação orientada a objetos [24]. O código fonte escrito em Java é primeiramente traduzido para o Java *bytecode*, um conjunto de instruções neutro quanto à arquitetura. A execução do programa é realizada pela máquina virtual Java (JVM), implementada para cada plataforma, que executa as sequências de *bytecode* (o conjunto de instruções de baixo nível). Esse *design* torna programas em Java *bytecode* independentes de plataforma e traz flexibilidade e reusabilidade para a linguagem.

2.1.1 *Java Virtual Machine*

A *Java Virtual Machine* é a plataforma Java, o alvo para o qual os programas Java são compilados. A JVM é um misto de arquitetura e sistema operacional, que executa as sequências de *bytecode* e realiza a interface entre a aplicação Java e a plataforma real onde ela é executada. Ela também é responsável por realizar tarefas como escalonamento e gerenciamento de memória das aplicações Java, daí as características de sistema operacional presentes nela (o esquema da arquitetura da JVM pode ser visto na figura 2.1). Duas partes fundamentais da JVM são o *Garbage Collector*, responsável por liberar toda a memória não mais necessária as aplicações Java em execução na JVM e

o compilador *Just-In-Time* (JIT), que apesar de não ser fundamental para e nem parte da especificação básica da JVM, é o módulo fundamental para aumentar o desempenho das aplicações Java.

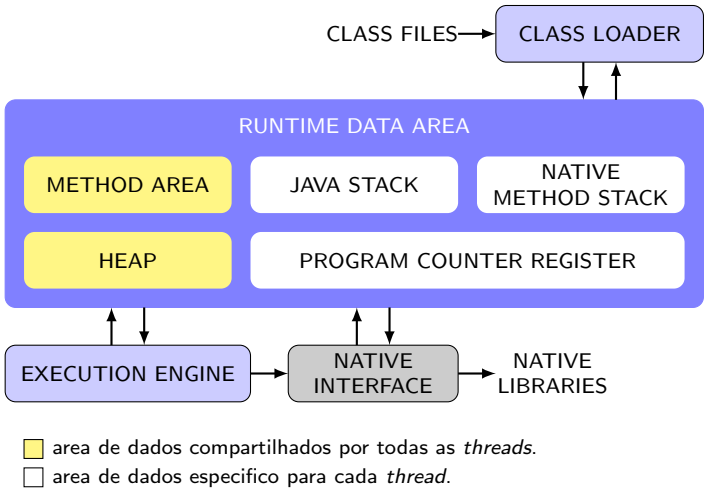

Figura 2.1: Diagrama da máquina virtual Java [27].

2.1.2 Compilador Java *just-in-time*

Embora a neutralidade de plataforma, flexibilidade e reusabilidade sejam vantagens expressivas para uma linguagem de programação, a execução por interpretação impõe uma queda de desempenho considerável, principalmente no tempo de busca e decodificação do *bytecode*. Uma maneira encontrada de melhorar o desempenho é usar um compilador *just-in-time* que converte, em tempo de execução, as sequências de *bytecode* em sequências equivalentes de código nativo [14, 24]. Um compilador Java *just-in-time* (JIT) gera código nativo a partir das instruções *bytecode* durante a execução do programa, sendo assim, a velocidade de compilação é mais importante em um compilador JIT do que em um compilador tradicional, requerendo assim que os algoritmos de otimização sejam leves e eficientes [2, 24, 25].

O Compilador JIT deve ser leve e produzir um código nativo eficiente [2, 6]. Porém, as técnicas de otimização de código utilizada em compiladores tradicionais são pesadas demais para serem usadas no compilador JIT, assim novas técnicas são exploradas [2, 6, 14, 17, 21, 28]. Uma dessas tecnologias é a divisão do JIT em vários níveis, um primeiro nível gera um código nativo com velocidade, não criando um *overhead* no tempo de execução, enquanto um JIT de segundo nível utiliza de técnicas de otimização gerando um código mais eficiente, ou ainda fazendo a compilação focando somente partes do código que são mais utilizadas e não em toda sequência de *bytecode* [22].

2.1.3 *Java Native Interface*

Java Native Interface (JNI) é um *framework*, parte da plataforma Java, que permite a interação entre código Java e código nativo. Através do JNI, uma aplicação Java pode realizar a execução de um método ou código nativo escrito em várias linguagens, como C ou C++, ou ainda o caminho inverso, uma aplicação nativa pode iniciar a JVM para realizar a execução de um método em Java. Essa facilidade permite a utilização de bibliotecas nativas dentro de uma aplicação Java, reutilizando código, tanto nativo quanto em bytecode, ou para implementar partes críticas para o desempenho de uma aplicação Java nativamente a fim de otimizar a execução da mesma.

Do ponto de vista do código nativo, o JNI é uma biblioteca que provê uma estrutura que serve de interface de acesso ao ambiente Java. Para a versão do JNI para a linguagem C a estrutura é um `struct` composto primariamente por ponteiros para funções que realizam a interface com o ambiente Java e possibilitam a conversão entre os tipos de dados básicos do Java e os tipos de dados nativos para que possam ser utilizados no ambiente nativo (a correspondência entre os tipos de dados Java e e os tipos de dados nativo no JNI para a linguagem C pode ser vista na tabela 2.1). Tipos de dados primitivos e vetores em Java são transformados diretamente para os tipos de dados e vetores nativos da linguagem utilizada para o JNI. Porém o acesso a um objeto Java se dá sempre por meio de funções que dão acesso a seus atributos e de funções que invocam a execução de seus métodos.

Java Language Type	**Native Type**	**Description**
`boolean`	`jboolean`	unsigned 8 bits
`byte`	`jbyte`	signed 8 bits
`char`	`jchar`	unsigned 16 bits
`short`	`jshort`	signed 16 bits
`int`	`jint`	signed 32 bits
`long`	`jlong`	signed 64 bits
`float`	`jfloat`	signed 32 bits
`double`	`jdouble`	signed 64 bits

Tabela 2.1: Correspondência entre tipos de dados primitivos em Java e os tipos de dados nativos no JNI para a linguagem C [18].

Do ponto de vista do código Java, as funções nativas são métodos de uma classe com somente seus protótipos declarados dentro do código Java e possuindo o tributo `native`.

A implementação de tais métodos são fornecidas para a JVM e, consequentemente para a aplicação Java através de uma *shared library* gerada a partir do código em C ou outra linguagem.

Para detalhar o funcionamento do JNI, a partir desse ponto será utilizada a linguagem C como referência para o código nativo e serão utilizadas frações de uma aplicação Java de exemplo um pouco mais complexa do que o "*Hello World!*". A aplicação é separada em três arquivos, `WriteBuffer.java`, que pode ser visto no código 2.1, esta escrita a classe java, `WriteBuffer.h`, que pode ser visto no código 2.2, se encontra as assinaturas das funções JNI geradas pelo utilitário `javah` e por último `WriteBuffer.c`, que pode ser visto nos códigos 2.3 e 2.4, implementa os métodos JNI.

```
1  import java.io.File;
2  import java.lang.String;
3
4  class WriteBuffer {
5          private native void write(File file, byte[] in);
6          public static void main (String[] args) {
7                  File f = new File ("hello.txt");
8                  String s = new String("Hello World!!!");
9                  new WriteBuffer().write(f, s.getBytes());
10         }
11         static {
12                 System.loadLibrary("WriteBuffer");
13         }
14 }
```

Código 2.1: Implementação da classe Java utilizando um método nativo através de JNI.

O funcionamento da aplicação exemplo, chamada **WriteBuffer**, é a que se segue: a classe Java faz um uso simples e único do método nativo `write()` cuja a tarefa é escrever um vetor de *bytes* em um arquivo referenciado por um objeto da classe `File`. A função nativa implementa o método `write()` e deve extrair o nome do arquivo do objeto `File` utilizando a interface JNI para poder executar a tarefa designada. A figura 2.2 mostra as etapas necessárias para criar uma aplicação JNI, os nomes dos arquivos são os utilizados no exemplo **WriteBuffer**.

A assinatura da função JNI nativa em C gerada a partir da assinatura na classe Java difere essencialmente da original pois toda função JNI em C possui dois argumentos extras em relação aos argumentos da assinatura original do método na classe Java. O primeiro deles é um ponteiro para a estrutura que serve de interface de acesso ao ambiente Java e o segundo é uma referência para o próprio objeto da classe a qual o método nativo pertence (algo como a *keyword* `self` em Java). O código 2.5 mostra o exemplo de assinatura do método nativo em `write()` Java e o código 2.6 mostra a

```
1  /* DO NOT EDIT THIS FILE - it is machine generated */
2  #include <jni.h>
3  /* Header for class WriteBuffer */
4  #ifndef _Included_WriteBuffer
5  #define _Included_WriteBuffer
6  #ifdef __cplusplus
7  extern "C" {
8  #endif
9
10 /* Class:     WriteBuffer
11  * Method:    write
12  * Signature: (Ljava/io/File;[B)V */
13 JNIEXPORT void JNICALL Java_WriteBuffer_write
14                   (JNIEnv *, jobject, jobject, jbyteArray);
15
16 #ifdef __cplusplus
17 }
18 #endif
19 #endif
```

Código 2.2: Declaração do método nativo gerado através do utilitário `javah`.

assinatura da correspondente função em C.

Como já dito, o acesso aos tipos básicos em Java são feitos através de conversão direta para tipos básicos em C, o mesmo não acontece com os objetos. Todo objeto é representado por uma referência do mesmo tipo (`jobject`) e, para ter acesso às particularidades de cada objeto, é necessário antes identificar a classe do mesmo e que é feita através da função `GetObjectClass()` (figura 2.7) parte da interface JNI. O Acesso a atributos e métodos de um objeto são disponíveis na interface como famílias de funções genéricas, independente da classe do objeto, se diferenciando somente pelo tipo e qualificadores do atributo a ser acessado ou o tipo de retorno e qualificadores do método a ser invocado. Exemplos de funções de acesso aos atributos e métodos de um objeto podem ser vistos nos códigos 2.8, 2.9 e 2.10.

Assim como é necessário a identificação da classe do objeto para poder acessá-lo, para acessar seus métodos e atributos é necessário ter a identificação do método ou do atributo que se deseja acessar para poder utilizar as funções da interface JNI comentadas anteriormente. As funções `GetFieldID()` e `GetMethodID()` mostradas nos códigos 2.11 e 2.12 são usadas para receber a identificação dos membros de uma classe. Para identificar o membro, as funções recebem, além do identificador da classe, duas *strings* sendo uma o nome do membro da classe e a outra a assinatura desse membro, que segue uma sintaxe estabelecida na especificação da plataforma Java.

Para ilustrar o funcionamento do JNI juntamente com algumas funcionalidades, o

```
1  #include "WriteBuffer.h"
2  #include <stdio.h>
3
4  JNIEXPORT void JNICALL Java_WriteBuffer_write
5                  (JNIEnv *env, jobject obj, jobject file, jbyteArray in) {
6
7          /* getting the object class */
8          jclass class = (*env)->GetObjectClass(env, file);
9          /* getting the class's method id. */
10         jmethodID methodid = (*env)->GetMethodID(env, class, "getPath",
11                         "()Ljava/lang/String;");
12         if (methodid == NULL) {
13                 fprintf(stderr, "error: method not found.\n");
14                 return;
15         }
16         /* calling the object method, the return is a Strig object. */
17         jstring str = (*env)->CallObjectMethod(env, file, methodid);
18         /* getting the native representation of the string. */
19         const char *fname = (*env)->GetStringUTFChars(env, str, NULL);
20         if (fname == NULL) {
21                 fprintf(stderr, "error: unable to convert the string.\n");
22                 return;
23         }
24         /* Getting the array of bytes in a native representation. */
25         jbyte *bytes = (*env)->GetByteArrayElements(env, in, NULL);
26         if (bytes == NULL) {
27                 fprintf(stderr, "error: unable to convert the buffer.\n");
28                 /* releasing the String object. */
29                 (*env)->ReleaseStringUTFChars(env, str, fname);
30                 return;
31         }
32
33 /* ... */
```

Código 2.3: Implementação do método nativo utilizando JNI (parte 1).

código 2.13 mostra um fragmento da utilização do método nativo `write()` por sua classe Java. Ilustrando o uso de variáveis Java de tipos de dado primitivos pelo código nativo, o código 2.14 contêm fragmentos que demonstram a manipulação de um vetor de `bytes`. Por último, o código 2.15 ilustra o uso de objetos Java pelo código nativo manipulando um objeto da classe `File`.

```
33 /* ... */
34
35         /* getting the number of elements in the array. */
36         jsize size = (*env)->GetArrayLength(env, in);
37         if (size == 0) {
38                 fprintf(stderr, "error: getting buffer length.\n");
39                 /* releasing the String object. */
40                 (*env)->ReleaseStringUTFChars(env, str, fname);
41                 /* releasing the array of bytes . */
42                 (*env)->ReleaseByteArrayElements(env, in, bytes, JNI_ABORT);
43                 return;
44         }
45         FILE* f = fopen(fname, "wb");
46         if (f == NULL) {
47                 fprintf(stderr, "error: unable to open the file.\n");
48                 /* releasing the String object. */
49                 (*env)->ReleaseStringUTFChars(env, str, fname);
50                 /* releasing the array of bytes . */
51                 (*env)->ReleaseByteArrayElements(env, in, bytes, JNI_ABORT);
52                 return;
53         }
54         size_t count = fwrite(bytes, sizeof(jbyte), size, f);
55         if (count < size) {
56                 fprintf(stderr, "error: buffer was not entirely wrote.\n");
57                 /* close the file stream, no error check if done because we
58                  * are exiting the function anyway. */
59                 fclose(f);
60                 /* releasing the String object. */
61                 (*env)->ReleaseStringUTFChars(env, str, fname);
62                 /* releasing the array of bytes . */
63                 (*env)->ReleaseByteArrayElements(env, in, bytes, JNI_ABORT);
64                 return;
65         }
66         /* close the file stream, no error check if done because we are
67          * exiting the function anyway. */
68         fclose(f);
69         /* releasing the String object. */
70         (*env)->ReleaseStringUTFChars(env, str, fname);
71         /* releasing the array of bytes . */
72         (*env)->ReleaseByteArrayElements(env, in, bytes, JNI_ABORT);
73         return;
74 }
```

Código 2.4: Implementação do método nativo utilizando JNI (parte 2).

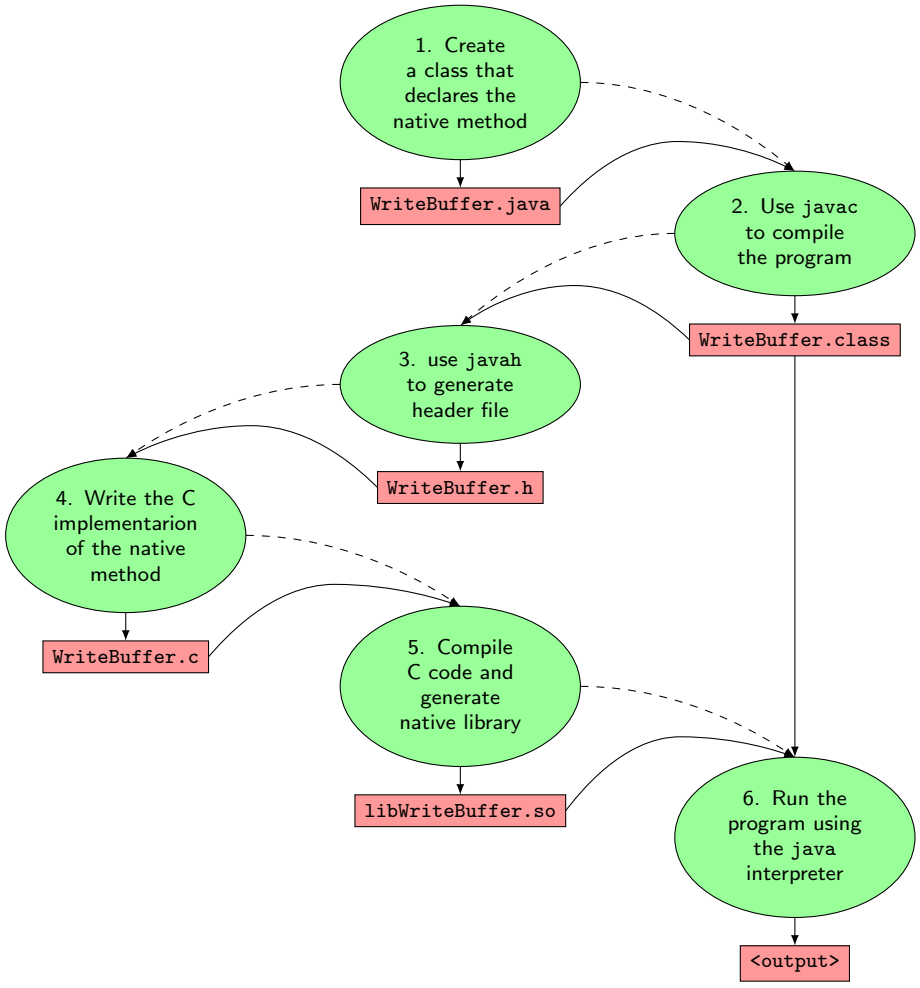

Figura 2.2: Etapas para criação de um método JNI [18].

```
1 private native void write (File file, byte[] in);
```

Código 2.5: Protótipo do método `write()`, um exemplo da declaração de um método Java para utilização do JNI, onde **file** é um objeto da classe `File` e **in** é um vetor do tipo primitivo `byte`.

```
1 JNIEXPORT void JNICALL Java_WriteBuffer_write (JNIEnv *env,
2                                                jobject obj,
3                                                jobject file,
4                                                jbyteArray in);
```

Código 2.6: Protótipo da função `write()`, um exemplo da assinatura de uma função em C correspondente a assinatura do método nativo `write()`(código 2.5) em Java, onde **env** é o ponteiro para a estrutura de interface JNI, **obj** é a variável referente ao próprio objeto da classe do qual o método nativo pertence. **file** é a variável referente ao objeto que é o primeiro parâmetro da assinatura em Java do método nativo e analogamente **in** refere ao objeto que é o segundo parâmetro.

```
1 jclass *GetObjectClass (JNIEnv *env, jobject obj);
```

Código 2.7: Protótipo da função `GetObjectClass()`, onde **env** é o ponteiro para a estrutura JNI, **obj** a referência para o objeto que se deseja identificar a sua classe e o retorno é a referência da classe do objeto em questão.

```
1 <NativeType> Get<Type>Field (JNIEnv *env, jobject obj, jfieldID fieldID);
```

Código 2.8: Protótipo da família de funções `Get<Type>Field()`, onde **env** é o ponteiro para a estrutura JNI, **obj** a referência para o objeto de onde se quer ler o valor do atributo, **fieldID** o identificador do atributo em questão e o retorno da função é o valor do atributo no tipo de dado nativo correspondente. As *tags* `<NativeType>` e `<Type>` referem aos pares da tabela 2.1 ou correspondem respectivamente ao tipo nativo `jobject` e a classe `Object`.

```
1 <void Set<Type>Field (JNIEnv *env, jobject obj, jfieldID fieldID,
2                       <NativeType> value);
```

Código 2.9: Protótipo da família de funções `Set<Type>Field()`, onde **env** é o ponteiro para a estrutura JNI, **obj** a referência para o objeto de onde se quer atribuir o valor do atributo, **fieldID** o identificador do atributo em questão e **value** é o valor a ser atribuído ao atributo. As *tags* `<NativeType>` e `<Type>` referem aos pares da tabela 2.1 ou correspondem respectivamente ao tipo nativo `jobject` e a classe `Object`.

```
1 <NativeType> Call<Type>Method (JNIEnv *env, jobject obj,
2                                jmethodID methodID, ...);
```

Código 2.10: Protótipo da família de funções `Call<Type>Method()`, onde **env** é o ponteiro para a estrutura JNI, **obj** a referência para o objeto que o método a ser invocado pertence, **MethodID** o identificador do método em questão, o retorno é o valor de retorno do método em um tipo de dado nativo e os demais parâmetros são variáveis e referentes aos parâmetros passados ao método invocado. As *tags* `<NativeType>` e `<Type>` referem aos pares da tabela 2.1 ou correspondem respectivamente ao tipo nativo `jobject` e a classe `Object` ou o tipo de dado nativo `void` e a classe `Void`. As famílias de funções `Call<Type>MethodA()` e `Call<Type>MethodV()` são variações onde se diferencia somente a forma de passagem dos parâmetros referente os parâmetros do método a ser invocado.

```
1 jfieldID GetFieldID (JNIEnv *env, jclass clazz, const char *name,
2                      const char *sig);
```

Código 2.11: Protótipo da função `GetFieldID()`, onde **env** é o ponteiro para a estrutura JNI, **clazz** a referência da classe da qual o atributo desejado pertence, **name** o nome do atributo em questão, **sig** a assinatura do atributo, que segue a sintaxe para assinaturas de membros de classe estabelecida na especificação da plataforma Java e o retorno da função é o identificador do atributo.

```
1 jmethodID *GetMethodID (JNIEnv *env, jclass clazz, const char *name,
2                         const char *sig);
```

Código 2.12: Protótipo da função `GetMethodID()`, onde **env** é o ponteiro para a estrutura JNI, **clazz** a referência da classe da qual o método desejado pertence, **name** o nome do método em questão, **sig** a assinatura do método, que segue a sintaxe para assinaturas de membros de classe estabelecida na especificação da plataforma Java e o retorno da função é o identificador do método.

```
1  class WriteBuffer {
2          private native void write (File file, byte[] in);
3          public static void main (String[] args) {
4                  File f = new File ("hello.txt");
5                  String s = new String("Hello World!!!");
6                  new WriteBuffer().write(f, s.getBytes());
7          }
8          static {
9                  System.loadLibrary("WriteBuffer");
10         }
11 }
```

Código 2.13: Declaração e implementação da classe Java `WriteBuffer` utilizando um método nativo através de JNI.

```
1  JNIEXPORT void JNICALL Java_WriteBuffer_write (JNIEnv *env,
2                                                 jobject obj,
3                                                 jobject file,
4                                                 jbyteArray in) {
5
6          /* Getting the array of bytes in a native representation. */
7          jbyte *nbuffer = (*env)->GetByteArrayElements(env, in, NULL);
8
9          /* getting the number of elements in the array. */
10         jsize nsize = (*env)->GetArrayLength(env, in);
11
12         /* ... */
13
14         /* releasing the array of bytes . */
15         (*env)->ReleaseByteArrayElements(env, in, nbuffer, JNI_ABORT);
16 }
```

Código 2.14: Manipulação de um vetor do tipo básico `byte` em Java dentro de um método em C através de JNI. O exemplo, ilustra a conversão do vetor de um tipo Java para um tipo nativo que pode ser livremente utilizado dentro do código C, a leitura de informações do vetor, no exemplo sendo a leitura da quantidade de elementos e, por fim, liberando o vetor nativo resultado da conversão do vetor em Java.

```
1  JNIEXPORT void JNICALL Java_WriteBuffer_write (JNIEnv *env,
2                                                 jobject obj,
3                                                 jobject file,
4                                                 jbyteArray in) {
5
6          /* getting the object class */
7          jclass class = (*env)->GetObjectClass(env, file);
8
9          /* getting the class's method id. */
10         jmethodID methodid = (*env)->GetMethodID(env, class, "getPath",
11                         "()Ljava/lang/String;");
12
13         /* calling the object method, the return is a Strig object. */
14         jstring str = (*env)->CallObjectMethod(env, file, methodid);
15
16         /* getting the native representation of the string. */
17         const char *fname = (*env)->GetStringUTFChars(env, str, NULL);
18
19         /* ... */
20
21         /* releasing the String object. */
22         (*env)->ReleaseStringUTFChars(env, str, fname);
23 }
```

Código 2.15: Manipulação de um objeto da classe `File` em Java dentro de um método em C através de JNI. A utilização de um objeto é feita através do uso de funções genéricas independe da classe do objeto. Para utilizar um objeto, primeiro é buscado o identificador da classe do objeto. Para se utilizar um método do objeto, necessita-se primeiro buscar o identificador do método utilizando o identificador da classe e a assinatura do método em questão. O método é chamado e o seu retorno pode ser utilizado. No caso, o retorno é um objeto da classe `string` que é tratada especialmente no JNI, existindo funções especificas para essa classe.

2.2 Cell Broadband Engine Architecture

A *Cell Broadband Engine Architecture* (CBEA), é uma arquitetura *multi-core* heterogênea, RISC e com suporte a vetorização, com unidades SIMD (*Single Instruction, Multiple Data*), que em sua primeira implementação oferece um poder computacional teórico de até 200 GFlops em um chip de 3.2GHz [8]. O processador Cell carrega um grande potencial para aplicações de computação intensiva, mas requer também resolver os desafios causados pela arquitetura não convencional deste processador [3].

A CBEA foi definida de forma a implementar processadores voltados para o processamento distribuído. A intenção é permitir a implementação de uma grande variedade de simples e múltiplos processadores e configurações de memória, de forma a satisfazer diferentes sistemas e necessidades [12].

A CBEA define quatro componentes funcionais, sendo duas unidades de processamento distintas, o *PowerPC Processor Element* (PPE) e o *Synergistic Processor Element* (SPE), um controlador de interrupções, o *Internal Interrupt Controller* (IIC), e um barramento que interliga o sistema, o *Element Interconnect Bus* (EIB). A arquitetura especifica que o sistema deve conter um IIC, um EIB, um ou mais PPEs e um ou mais SPEs, sendo que a primeira geração de processadores Cell possui um PPE e oito SPEs. O esquema básico da primeira implementação da arquitetura Cell pode ser visto na figura 2.3.

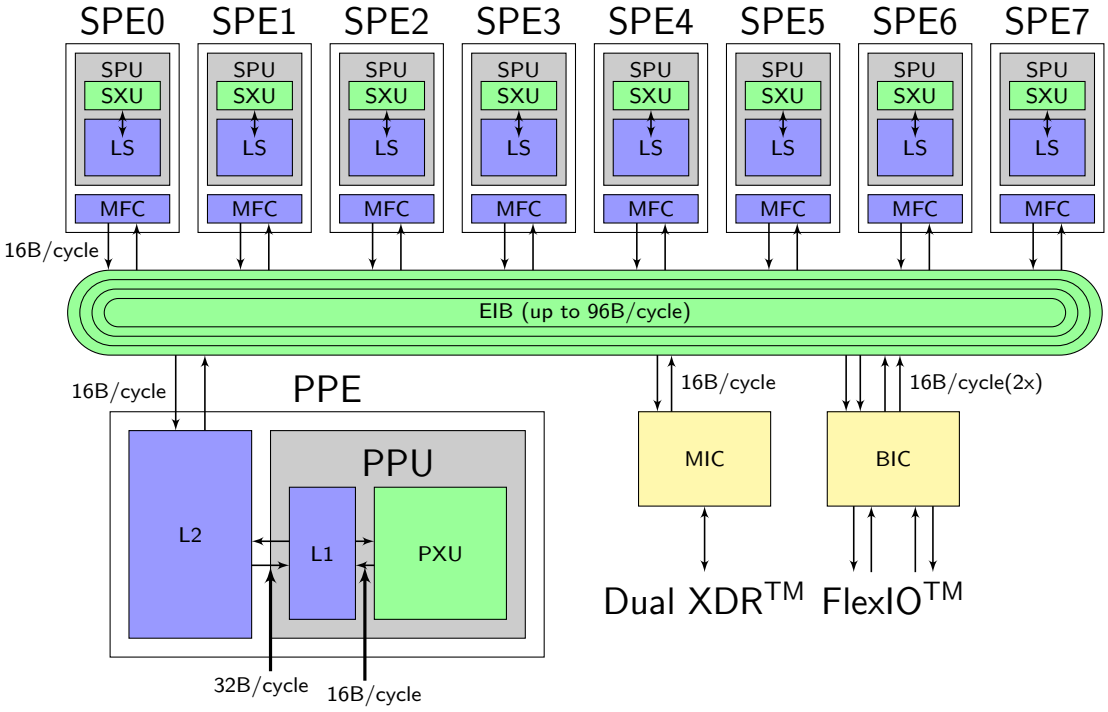

Figura 2.3: Diagrama do processador Cell. Imagem adaptação do original [13].

2.2.1 *Power Processor Element*

PowerPC Processor Element é a arquitetura Power de 64 bits, a mesma usada na linha de processadores PowerPC da IBM. No processador Cell, o PPE é formado pela unidade de execução *Power Processor Unit* (PPU) (incluindo a memória cache L1) juntamente com sua a memória cache L2. A unidade de execução PPU, na primeira implementação da arquitetura Cell, tem as seguintes características [10, 15]:

- Processador RISC escalar;
- Execução *in-order*;
- Possui suporte ao conjunto de instruções do PowerPC 970;
- Capacidade de execução de duas *threads* simultaneamente;
- Inclui suporte ao conjunto de instruções VMX para SIMD.

Pelas restrições de projeto do Cell, esse núcleo não possui, relativamente, um grande poder de processamento e foi concebido na arquitetura com a intenção de ser usado principalmente como um controlador, ou gerenciador, dos demais *cores*.

2.2.2 *Synergistic Processor Element*

O Processador, ou núcleo *Synergistic Processor Element* é formado por um conjunto de unidades funcionais, o *Synergistic Processor Unit*, que é a unidade de execução, incluindo nela a memória local dedicada *Local Store* (LS), e o controlador de memória *Memory Flow Controller* (MFC). O MFC por sua vez contêm uma *Memory Management Unit* (MMU) e uma *Replacement Management Table* (RMT) [12]. As características da unidade de execução SPU na primeira implementação da arquitetura Cell são:

- Processador RISC vetorial (SIMD);
- Execução *in-order*;
- *Dual issue* de instruções;
- Banco de registradores contendo 128 registradores de 128 bits;
- Execução de uma única *thread*.

Todos os registradores do banco de registradores são de propósito geral, cada um sendo um vetor de quatro palavras de 32 bits. O *dual issue* é muito específico no SPE, ele possui duas unidades aritméticas para propósitos diferentes e, para que funcione o despacho de instruções em pares, esses pares devem ser formados por duas instruções que utilizam cada um uma unidade aritmética diferente, sendo a primeira instrução relativa à unidade número um e a segunda relativa a segunda unidade aritmética. O grande número de registradores desse núcleo se deve à característica das arquiteturas RISC, que costumam ter muitos registradores para permitir melhores otimizações no código em tempo de compilação e assim otimizar o desempenho evitando muitas leituras e escritas na memória para reutilizar um registrador quando todos já estão ocupados.

O *Synergistic Processor Element* é o núcleo onde se concentra o grande poder de processamento do processador Cell. Cada processador Cell da primeira e atual geração possui 8 unidades SPE, interligadas através de um barramentos de alta velocidade de 3,2Gbps. Essas unidades possuem um conjunto de instruções diferente do conjunto de instruções do núcleo PowerPC que integra o processador Cell exigindo, por essa razão, um ambiente de desenvolvimento separado. A maior restrição do SPE é a sua pequena quantidade de memória local na atual implementação, apenas 256KBytes [10,15]. Todo acesso à memória principal do Cell deve ser feito através de transferências explícitas de DMA de/para a memória local. Assim, tanto o código quanto os dados necessários devem estar presentes na memória local do SPE para poderem ser utilizados.

As restrições descritas acima para o acesso a memória principal fazem com que os modelos de programação para o Cell sejam focados em transferências de dados intercaladas com processamento. Para aumentar o desempenho, o uso de diversos SPEs pode permitir uma transferência praticamente contínua de dados de/para a memória principal juntamente com o processamento dos mesmos.

Embora o SPE seja um núcleo focado e otimizado para aplicações com características específicas [7, 16], ele é um processador de propósito geral, uma vez que possui o conjunto de instruções completo para tal propósito e, devido a sua grande quantidade no processador Cell (8 SPEs na versão atual), é possível realizar todos os tipos de tarefas com esse núcleo.

2.2.3 *Memory Flow Controller*

O *Memory Flow Controller* é uma unidade fundamental para a arquitetura Cell. Podendo ser considerado, dado as devidas proporções, um núcleo de processamento, o MFC é responsável por gerenciar todas as transferências de dados entre o SPE e qualquer outro elemento externo, incluindo outro SPE.

As transferências de dados são feitas explicitamente através do envio de comandos

específicos para o MFC, que os interpreta e realiza o DMA. O MFC possui duas filas de comandos, uma para as requisições externas e outra para os comandos a partir do SPU. Uma característica fundamental do MFC e da LS é que é permitido realizar operações na LS paralelamente tanto a partir do SPU quanto a partir do MFC.

O MFC possui uma limitação que exige que todas as transferências de DMA sejam alinhadas a 16 bytes caso sejam maiores ou iguais a 16 bytes. Caso contrário, ela deve ser de 2,4 ou 8 bytes, e alinhada ao byte correspondente, ou seja, para dois bytes, o endereço deve ter o último bit zero, para quatro, os dois últimos bits e assim por diante até dezesseis bytes. Ainda, os endereços de fonte e destino devem possuir o mesmo alinhamento.

2.3 Pesquisas sobre Java para a arquitetura Cell B.E.

Desde o lançamento da arquitetura Cell, surgiram vários esforços para explorar seu potencial computacional e ao mesmo tempo manter uma interface amigável para o desenvolvimento de aplicações novas e portabilidade de aplicações já existentes. Um foco nas pesquisas foi linguagem e plataforma Java e sua integração com a plataforma Cell. Nessa área foram contempladas várias abordagens diferentes, sempre buscando uma melhora no desempenho para aplicações Java.

O acompanhamento e estudo das pesquisas recentes realizadas na mesma área do projeto é importante para definir as prioridades das etapas e passos tomados do projeto. Dentre os trabalhos recentes, destacam-se: **Java on Cell** [23], **CellGC** [5], **CellVM** [20] e **Hera-VM** [19].

Podem ser divididos em dois grupos os trabalhos de porte e integração da plataforma Java para o Cell. O primeiro é dos trabalhos em que o foco principal é aumentar o desempenho despachando a execução da aplicação Java para os SPEs e o segundo grupo formado pelos trabalhos em que o foco são deslocar para os SPEs partes ou toda a própria JVM com o objetivo de diminuir o *overhead*.

Dentre dos trabalhos desenvolvidos que se enquadram no primeiro grupo, destacam-se: **Java on Cell** [23], um compilador JIT para o SPE, **CellVM** [20], **Hera-VM** [19] e **JAM_PPU** [11] replicam a máquina virtual para os diversos núcleos e executam diferentes threads Java em cada um. A primeira e terceira abordagens implementaram uma JVM executando código Java interpretado. Já a segunda não é interpretada, utiliza um JIT para o núcleo destino. Ainda, das três últimas abordagens, a última expõe para o programador os diversos núcleos e cabe a esse criar threads usando um *framework* específico. As duas primeiras abordagens são transparentes para o programador, bastando as aplicações utilizarem os recursos habituais de threads Java. No segundo grupo, está **CellGC** [5], um porte do *Garbage Collector* para o SPE.

2.3.1 CellGC

CellGC é um trabalho de desenvolvimento de um *garbage collector* automático e dinâmico para o SPE, implementando e otimizando o **BDW garbage collector** (Bohem-Demers-Weiser garbage collector) para a arquitetura Cell.

No porte do *garbage collector* a fase de Marcação (*mark phase*) do BDW *garbage collector* foi transferido para execução no SPE, dando enfase na exploração da LS com semânticas de *copy-in/copy-out* [5].

Na otimização do *gargabe collector*, foi analisado o desempenho de diversas abordagens de gerenciamento de memória, escolhendo um gerenciamento explícito da memória local do SPE, usando ambas as técnicas de *cache* e *prefetching* com a funcionalidade de DMA provida pelo MFC [5].

2.3.2 CellVM

CellVM tem como objetivo distribuir *threads* Java para execução nos SPEs criando um ambiente homogêneo sobre a plataforma heterogênea da arquitetura Cell (o *design* básico pode ser visto na figura 2.4). O projeto se baseia na execução interpretada do código Java em máquinas virtuais replicada entre SPEs trabalhando em conjunto. O trabalho usa a **JamVM** como base para o de uma JVM interpretada para o SPE, criando uma máquina virtual onde as *threads* Java são interpretadas no SPE, o PPE integra o conjunto de JVMs, realiza o controle sobre o acesso aos dados e trata o código que não pode ser gerenciado nos SPEs, como a invocação de métodos nativos. A CellVM utiliza *software cache*, tanto para dados quanto para código, para transpor a restrição de memória dos SPEs e gerar uma memória lógica unificada.

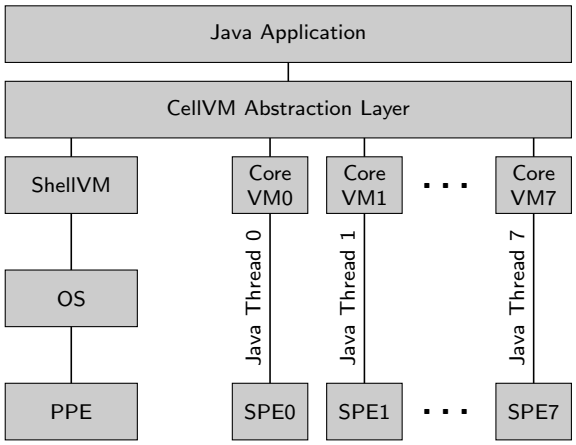

Figura 2.4: CellVM [20].

2.3.3 JAM_PPU

O trabalho **JAM_PPU**, diferentemente dos demais trabalhos aqui apresentados, não busca tornar as características da arquitetura Cell completamente transparentes para o programador Java. O trabalho apresenta um modelo para execução de programas Java na arquitetura Cell utilizando um *framework* que permite a distribuição explícita da aplicação entre os SPEs, tal qual um modelo de memória explicitamente distribuída [11].

O Modelo de execução da **JAM_PPU** é muito simples, consiste de uma JVM completa executando no PPE e de uma JVM otimizada para execução no SPE e integrada para se comunicar com a JVM principal no PPE. A JVM para o SPE é replicada para cada SPE utilizado e a JVM do PPE cumpre o papel de carregar e gerenciar as diversas JVMs especializadas (o esquema de execução pode ser visto na figura 2.5).

Como dito anteriormente, a **JAM_PPU** expõe ao programador a característica *multi-core* do Cell B.E. e disponibiliza as ferramentas necessárias para se utilizar tais recursos. Isso é feito de uma forma simples e elegante, o programador deve, para executar um código em um SPE, criar um objeto de uma classe que implementa uma interface especifica chamada `SPETask`. A `SPETask` possui um único método chamado `execute`, que é o método principal executado no SPE, similarmente ao método `run` da classe `Thread` da API padrão do Java. A classe `SPETask` define o ambiente de execução para um SPE, e para o objeto criado efetivamente ser carregado e executado em um SPE, é provida uma segunda classe, a `SPETaskDispatcher`, que possui um método estático responsável por despachar o objeto criado para um SPE. Esse método é o `executeTask` e é um método síncrono e que bloqueia a execução da *thread* que o chama até o término da tarefa no SPE, sendo assim comumente executado a partir de uma *thread* adicional para cada objeto a ser executado em um SPE.

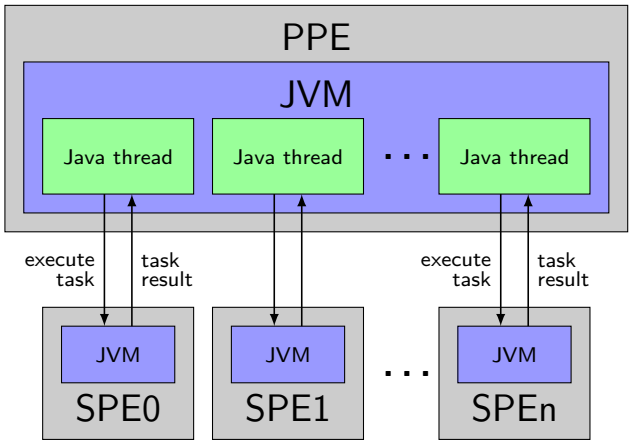

Figura 2.5: Modelo de execução da máquina virtual JAM_PPU [11].

2.3.4 Hera-VM

Hera-VM tem como objetivo distribuir *threads* Java para execução nos SPEs, assim como a CellVM, e também se baseia na replicação de JVMs entre os núcleos SPEs. Porém a máquina virtual base para o trabalho é a **JikesRVM**, uma JVM baseada em JIT que não executa código interpretada e utiliza o modelo "Java em Java". Ela traz uma diferença fundamental para o projeto pois quase todo o seu sistema é escrito em Java e uma pequena parte de código em *assembly* (o esquema básico pode ser visto na figura 2.6). Assim, o foco da Hera-VM é o porte da porção de código nativo para o SPE, o suporte do JIT para compilação para a arquitetura SPE e um escalonador dando a JVM a capacidade de migrar *threads* de um núcleo para outro núcleo, mesmo de arquiteturas diferentes. A Hera-VM, assim como a CellVM, utiliza *software cache*, tanto para dados quanto para código, para transpor a restrição de memória dos SPEs e gerar uma memória lógica unificada.

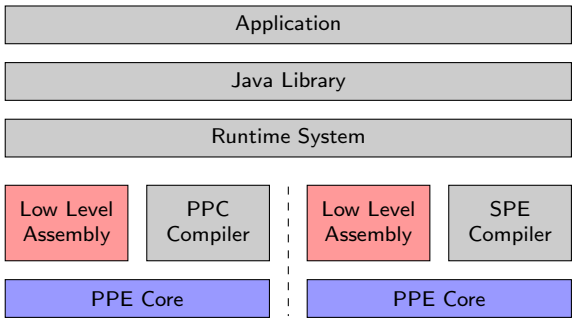

Figura 2.6: Hera-VM [19].

2.3.5 Java On Cell

Java on Cell tem como foco o suporte do JIT da **CacaoVM** para a arquitetura SPE. O objetivo é usar o JIT para despachar métodos de computação intensiva para serem executados no SPE, gerando código *just-in-time* para a arquitetura SPE (o esquema básico pode ser visto na figura 2.7). O JIT roda no PPE e age como um provedor de serviços e recursos para os métodos executando nos SPEs. Na abordagem em questão, os objetos no *heap* devem ser compartilhados entre todos os núcleos, tanto o PPE quanto os SPEs, sendo também o JIT um ponto de controle para o acesso sincronizado das diversas *threads* ao *heap* e para manter a sua consistência uma vez que, nessa abordagem, um objeto no *heap* pode ser movido temporariamente para a LS de um SPE.

O JIT do **Java on Cell** possibilita a seleção dinâmica dos métodos que serão executados no SPE otimizando sua utilização para os métodos mais apropriados para a sua

arquitetura. Com o JIT sendo o controle para a execução de métodos nos SPEs, ele se aproveita do acesso a memória principal para prover cache de métodos aos códigos compilados e prover os mesmos aos SPEs quando necessário [23].

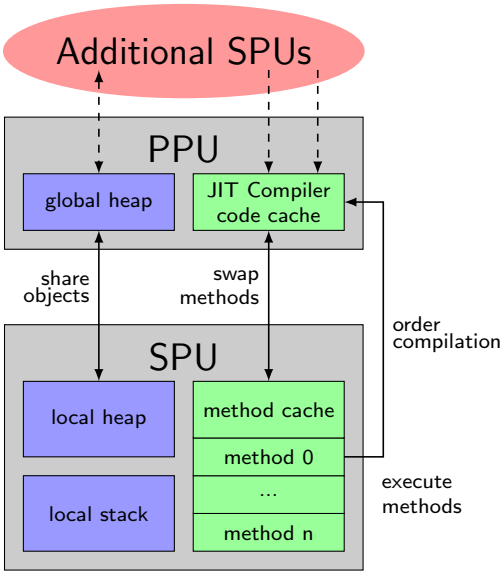

Figura 2.7: Modelo de execução da JVM Java On Cell [23].

2.4 Análise do estado da arte

Analisando os trabalhos, pode-se apontar as principais características onde estão se concentrando as pesquisas analisadas. Tanto **Java on Cell** quanto **Hera-VM** focam-se no suporte para o JIT da arquitetura SPE, a **JAM_PPU**, a **Hera-VM** e a **CellVM** focam na distribuição e gerenciamento da execução de várias *threads* Java em um conjunto de SPEs, seja interpretado ou compilado, e na replicação de toda a JVM para os SPEs. Somente o CellGC, tem como foco mover parte da JVM para os SPEs com o intuito de diminuir o *overhead* na JVM, mais precisamente, o CellGC foca em mover o trabalho do *garbage collector* para um SPE.

Capítulo 3

Integração da plataforma Java com a arquitetura Cell B.E.

Essa dissertação tem como objetivo a integração da plataforma Java com a arquitetura Cell B.E. de forma a combinar o desenvolvimento de aplicações, utilizando os atuais paradigmas de programação com o poder de processamento do processador Cell com a menor interferência possível.

A integração da plataforma Java será feita utilizando a abordagem de se portar bibliotecas nativas da plataforma Cell para a plataforma Java através do mecanismo de interface com código nativo. JNI. Esse mecanismo é parte da especificação da plataforma Java e torna possível a utilização explícita de código nativo, que utiliza o poder da arquitetura Cell, dentro de uma aplicação Java. Podendo ainda ser exposto ou não para a aplicação Java as características peculiares da arquitetura base.

3.1 *Java Native Interface* para a arquitetura Cell B.E.

A utilização do JNI para explorar os recursos da arquitetura Cell é um método menos genérico do que o de se trabalhar somente no *backend* da plataforma Java, alterando-se a JVM, pois é necessário prover à aplicação Java versões escritas usando JNI das bibliotecas utilizadas. Escolher quais bibliotecas codificar com JNI envolve análise de quais delas são mais utilizadas para cada aplicação e quais são mais compatíveis com as características do processador a serem exploradas.

Podemos separar as bibliotecas em dois conjuntos: as que fazem parte do conjunto de bibliotecas padrões da linguagem Java e as externas para propósitos específicos. O primeiro conjunto oferece um escopo de aplicação maior tendo em vista que são bibliotecas mais genéricas e de grande aplicação. Porém, essa mesma característica expõe

menores chances de otimizações se portadas para versões nativas do que bibliotecas com propósitos mais específicos. As do segundo grupo, que são desenvolvidas com propósitos específicos, contam com algoritmos que podem melhor explorar as possibilidades de desempenho da plataforma Cell dada a natureza específica das funcionalidades que visam implementar.

Em vista das características apresentadas acima, o foco, para aplicação de JNI nesse projeto, serão bibliotecas externas que ofereçam as características e algoritmos que melhor se adaptam a arquitetura do processador Cell. O uso de JNI pode ser feito entre Java e código nativo em várias linguagens, nesse trabalho optou-se por utilizar o JNI entre Java e C dado que C é a linguagem padrão e mais amplamente utilizada para desenvolvimento na plataforma Cell. A busca por bibliotecas e aplicações factíveis de serem adaptadas com o uso de JNI para serem usadas na plataforma Java será concentrada no site do centro de desenvolvimento da arquitetura Cell [1] que possui uma seção com cadastros de projetos desenvolvidos para a mesma.

3.1.1 Conflito de paradigmas

Realizar a conexão entre o código Java e uma biblioteca nativa em C requer resolver o problema de integração entre dois paradigmas de programação diferentes, dado que Java é uma linguagem orientada a objeto e C uma linguagem procedural e relativamente de baixo nível.

Para integrar uma biblioteca em C a um código orientado a objeto deve-se encapsular o seu conjunto de funções em uma ou mais classes. Se a biblioteca não necessita manter dados ou informações em alguma estrutura própria, pode-se utilizar uma classe abstrata realizando uma interface direta entre as funções em C com métodos estáticos em Java somente sendo necessário a conversão entre os dados de entrada e saída. Para ilustrar, uma biblioteca que se encaixaria nas características descritas seria a biblioteca padrão do C `string.h` que é formada por funções que realizam operações atômicas em vetores de caracteres.

Bibliotecas como a Posix `pthread.h` dependem de dados mantidos em estruturas próprias que persistem ao escopo de execução das funções de sua biblioteca. Para essa situação, além de se encapsular as funções de uma biblioteca, os seus dados também devem ser encapsulados em classes. Esse encapsulamento pode ser feito em classes dedicadas ou na mesma classe das funções, sendo que, nesse último caso, a vantagem é evitar passá-los por parâmetro para os métodos se esses não forem definidos como estáticos. Em ambas as situações, deve-se usar os tipos de dados existentes em Java para representar os dados da biblioteca. Se a estrutura e formato dos dados são conhecidos, pode-se usar tipos de dados equivalentes em Java. Essa abordagem permite

que os dados sejam utilizados também pelo código Java, além do código nativo. Porém, quando a estrutura ou formato não são conhecidos, uma abordagem mais genérica deve ser utilizada, como a serialização dos dados ou manter os dados no *heap* e se guardar a referência para os mesmos, sendo que ambas as técnicas não possibilitam o acesso direto aos dados pela aplicação Java.

Nesse trabalho são estudadas bibliotecas que se encaixam nos dois casos descritos acima. Para as bibliotecas que se encaixam no segundo caso, uma abordagem mista é utilizada, os dados são mantidos no *heap* e a sua referência será serializada em um vetor de bytes, tornando o código mais portável entre ambiente de execução de 32 e 64 bits, ambos suportados pela plataforma Cell.

3.1.2 Otimizações

Ainda que o JNI permita acessar código nativo, otimizações na interface entre o código Java e o código nativo podem ser necessárias para se obter um bom desempenho na aplicação que se utiliza de uma biblioteca JNI.

No processo de acesso aos dados Java pelo código JNI, um passo fundamental é a aquisição dos identificadores de Classe, método e atributos, assim como mostrado na seção 2.1.3. O processo de aquisição de tais identificadores é dispendioso e um ponto de otimização a ser explorado. Este objetivo pode ser alcançado, guardando os identificadores para futuro reuso, ou no momento da primeira utilização ou utilizando uma função específica para esse fim que seja chamada na construtora estática da classe que encapsula a biblioteca. Para o caso de identificadores de classe, ao se escolher guardar um identificador no momento da primeira utilização, deve-se estar atento para o fato de que o identificador de uma classe é variável, mudando cada vez que uma classe é desalocada da memória, o que torna inválido o valor previamente calculado. A saída para essa situação é manter a classe em memória, mantendo uma referência para a mesma e assim impedindo que o *Garbage Collector* desaloque a classe. Esse problema não acontece quando se usa uma construtora estática da classe, pois essa é chamada toda vez que a classe é carregada para a memória.

Na tradução dos dados em Java para dados que podem ser utilizados pelo código nativo, uma cópia dos mesmos deve ser feita, o que gera um processamento extra, não somente da cópia de dados em si, mas como também da alocação e desalocação de memória, principalmente quando manipulando vetores. A Interface JNI permite, em algumas situações, utilizar espaços de memória pré-alocados para tais conversões. Seja pelas limitações da interface JNI ou das características do algoritmo, nem sempre é possível ter controle sobre a alocação de recursos, sendo assim necessário a aplicação de políticas que minimizem constantes alocações de memória e cópias de dados, seja

no código Java ou seja no código nativo.

3.2 Porte de bibliotecas nativas Cell para a plataforma Java

O projeto de integração da plataforma Java com a arquitetura Cell B.E. consiste na análise de bibliotecas nativas existentes para a plataforma Cell e no porte das mesmas para a plataforma Java usando JNI.

A Seleção das bibliotecas a serem analisadas foi baseada principalmente na seção de projetos do site do centro de desenvolvimento da arquitetura Cell [1] que mantém uma lista de projetos externos desenvolvidos para a arquitetura. Dado que a lista de projetos inclui os mais variados tipos, incluindo *frameworks*, aplicações completas, bibliotecas, *benchmarks* e ferramentas, nem todos os projetos cadastrados são aptos a serem objetos de estudo para essa dissertação. Assim sendo, cerca de 50% dos projetos foram descartados por não serem bibliotecas ou aplicações de que não se possa extrair ou converter em bibliotecas e bibliotecas que não são voltadas para exploração das características de processamento da arquitetura Cell.

3.2.1 Análise de bibliotecas para a plataforma Cell

As bibliotecas selecionadas foram analisadas detalhadamente, sob o ponto de vista das características necessárias para viabilidade de implementação de uma interface JNI. Muitos projetos foram descartados pois não foram encontrados os códigos fontes nos seus sites, código incompleto, com erros, sem documentação mínima ou clareza suficiente no código, entre outros motivos. Para o caso de projetos que eram aplicações ou implementações de algoritmos, outros fatores de descarte foram: implementações pouco modularizadas que impossibilitavam ou dificultavam extrair um conjunto de funções para adaptar em uma biblioteca, dados de entrada e saída usados escritos diretamente no código (*hardcoded*), parâmetros de configuração/execução pouco flexíveis ou inexistentes e para aplicações que operam sobre dados de entrada e a falta de dados de exemplo ou documentação especificando os formatos dos mesmos. Para as bibliotecas, fatores de descarte incluíram falta de exemplo ou documentação de uso da mesma, incluindo explicações sobre os parâmetros das funções disponibilizadas ou o modo de operação das mesmas.

Aspectos técnicos

Além dos aspectos já citados encontrados ao se analisar a base de projetos desenvolvidos, outros aspectos mais técnicos foram observados. Os aspectos técnicos que impedem ou dificultam o porte de uma biblioteca nativa para JNI são diretamente ligados as decisões de implementações e podem requerer pequenos ajustes de códigos ou mesmo a reescrita completa dos mesmos.

Ambiente de execução: o Núcleo PPE da arquitetura Cell possui dois ambientes de execução: 32 bits ou 64 bits. Os ambientes são escolhidos dinamicamente para cada aplicação no momento de execução de acordo com o método utilizado para compilação dos binários a serem executados, se 32 bits, a arquitetura executa em modo de compatibilidade de 32 bits, se em 64 bits, ela executa em seu modo nativo. Como uma biblioteca JNI é ligada a partir da JVM, ela deve ser executada no mesmo ambiente que a JVM e, portanto, ser compilada para o mesmo ambiente de execução. Uma aplicação nativa para a plataforma Cell pode ser codificada de forma portátil para compilar tanto em 32 bits ou 64 bits, ou ser escrita de forma a só executar corretamente se compilada para o modo no qual foi planejada ser compilada. Dependendo da JVM utilizada, essa situação pode ser um problema, pois algumas JVM possuem a opção de se escolher o ambiente de execução das mesmas, e outras não. Caso a JVM e o código nativo sejam incompatíveis, alterações no último são necessárias.

Versão da `libspe`: A biblioteca primária de gerenciamento do SPE é a `libspe` que possui duas versões principais, a 1 e a 2. A interface e o paradigma de programação adotados na biblioteca mudou completamente da versão 1 para a 2, tornando as duas incompatíveis e a versão 1 obsoleta e não mais presente nas versões atuais do Cell SDK. Assim, projetos que usam a `libspe` versão 1 não são mais possíveis de compilar, a não ser com uma versão antiga do SDK. A adaptação de código usando a versão 1 para a versão 2 da biblioteca depende da complexidade de interação entre o código do PPE e o código do SPE, pois essa é a biblioteca usada pelo PPE para todo o controle, gerenciamento e comunicação com o SPE.

Modo de ligação: Algumas bibliotecas, como a `libspe` precisam ser ligadas explicitamente usando a biblioteca de ligação dinâmica, a `libdlfcn`, de forma a funcionarem corretamente, caso contrário a JVM reportará erro em tempo de execução quando tentar ligar as bibliotecas extras com a biblioteca JNI que estiver carregando. Assim, a ligação convencional não pode ser utilizada, como em uma aplicação nativa e o código escrito dessa forma deve ser alterado de forma a realizar a ligação explicitamente.

Uso de bibliotecas externas: O uso de bibliotecas externas diminui a portabilidade da biblioteca gerada, pois podem não estar disponíveis na plataforma de execução, além de poder gerar os problemas do modo de ligação e do ambiente de execução durante a adaptação para a biblioteca JNI ou mesmo não estarem disponíveis as versões de desenvolvimento, para se poder gerar o código binário.

Modo de execução do SPE: Essa situação mais comum em projetos de aplicações e algoritmos para o Cell, se caracteriza por utilizar uma implementação em que o código do SPE não se comunica com o código no PPE, ou mesmo não existindo o segundo código. Como a biblioteca JNI depende do código executado no PPE, a existência e comunicação do mesmo com SPE é fundamental, caso não exista, uma mudança completa no código pode ser necessária de forma a prover a comunicação necessária entre a biblioteca JNI e o código nativo no SPE.

Dentre os problemas técnicos que podem ser encontrados, quando se necessita mudanças no código, essas podem ser no código do PPE, no do SPE, ou nos dois. É importante observar que qualquer alteração no código para o SPE é muito mais custoso do que no código do PPE, pois as bibliotecas para o SPE são de mais baixo nível, o código do PPE geralmente é um código de controle, enquanto que no SPE fica o algoritmo que realiza a tarefa em si, assim mudanças em seu código podem fazer necessárias mudanças na lógica do algoritmo.

Projetos selecionados para o porte utilizando JNI

Dentre o conjunto de projetos analisados para porte para uma biblioteca JNI, três projetos foram selecionados. Através destes foi demonstrada a metodologia de porte do código e sua viabilidade. Os critérios para o sucesso como escolha de um projeto para porte em uma biblioteca JNI dependem dos aspectos gerais e técnicos discutidos acima. As características técnicas necessárias impostas pela plataforma de desenvolvimento são, compatibilidade com compilação para 32 bits, pois a JVM utilizada só suporta esse modo de execução, suporte a `libspe` versão 2, pois é a versão suportada pela atual versão do Cell SDK.

GeorgiaTech RC5: A GeorgiaTech possui um projeto de implementação de vários algoritmos otimizados para a plataforma Cell, cadastrado no centro de desenvolvimento da arquitetura Cell [1]. Dentre os algoritmos implementados, o algoritmo de criptografia RC5 se destaca por ser uma boa opção para a criação de uma interface JNI. Sua implementação para o Cell utiliza explicitamente compilação em 32 bits e a versão 2 da `libspe`. É implementado em forma de biblioteca, possuindo uma simples interface e código de exemplo.

SPU MD5: O projeto **SPU MD5** não faz parte dos projetos cadastrados no centro de desenvolvimento da arquitetura Cell, esse projeto é um *benchmark* implementado por Nick Breese [4], posteriormente otimizado por Jonathan Taylor [26]. A implementação consiste em uma aplicação SPE independente que implementa o algoritmo de *hash* MD5, usando a capacidade de computação vetorial do SPE para executar o algoritmo paralelamente sobre quatro blocos de dados independentes (em outras palavras, o cálculo de quatro *hashs* diferentes simultaneamente) que são inseridos diretamente no código da aplicação. Sendo uma aplicação SPE, a questão do modo de compilação não se aplica, em vista que o SPE sempre executa em 32 bits. Também não se aplica a questão da versão da `libspe` utilizada, já que essa é uma biblioteca para o PPE. Os problemas mais evidentes desse projeto são o fato de não aceitar entradas dinamicamente, não possuir uma interface de comunicação com o PPE ou mesmo qualquer tipo de interface. Ainda assim, esse projeto foi focado para implementação devido a grande otimização do algoritmo de MD5 para execução no SPE, incluindo o paralelismo usado através de instruções para cálculo vetorial.

Fast Matrix Multiplication on Cell Systems (MatMul): O Projeto **MatMul** desenvolvido por Hackenberg [9] é uma aplicação focada em realizar multiplicação de grandes matrizes quadradas com alto desempenho como parte de análise de desempenho da arquitetura Cell. Ela é uma aplicação estável e bem desenvolvida, possui uma versão utilizando a mais nova versão da `libspe`. Ela não depende de bibliotecas externas, e após análise concluiu-se que apesar de inicialmente utilizar explicitamente o modo 64 bits para compilação, a aplicação poderia funcionar em modo 32 bits também. Apesar de ser uma aplicação, ela somente realiza uma única tarefa e o formato de sua função principal torna simples sua adaptação para uma biblioteca. A aplicação também apresenta a comunicação e controle dos SPEs, sendo assim uma aplicação Cell completa, com código para o PPE e o SPE, o que facilita o porte.

Projetos não selecionados

A seguir será detalhado um pouco sobre alguns projetos interessantes para o porte para uma biblioteca JNI que não foram selecionados por não satisfazem em menor ou maior grau os requisitos citados anteriormente. Os projetos são em sua maioria parte do grupo de projetos cadastrados no centro de desenvolvimento da arquitetura Cell [1] e seus nomes seguem os mesmo encontrados no site em questão.

Cell based GA for JSSP: Esse projeto implementa um algoritmo genético para resolver o problema de escalonamento de trabalho ou sequenciamento de operações (*Job*

Shop Scheduling Problem). Esse projeto implementa uma aplicação completa com parâmetros flexíveis de configurações, tais como o número de SPEs usados no algoritmo. Esse projeto foi descartado pois a versão disponível no site utiliza a versão obsoleta da `libspe` e, sendo um algoritmo complexo, a adaptação da aplicação para o uso da versão atual da biblioteca foi desconsiderada.

Echo Cancelation: É um algoritmo parte do projeto **CIV Toolkit**. Ele utiliza um arquivo de áudio do ruído ambiente como base para cancelar o mesmo de um segundo arquivo de áudio, gerando uma stream de áudio sem o ruído. Sendo parte do mesmo projeto do **Min-Max Ant System**, ele compartilha as mesmas características técnicas do anterior, incluindo o fato da versão disponível não estar completa, o algoritmo gera somente um arquivo de áudio vazio, por isso foi descartado.

GeorgiaTech JPEG2000: Parte do projeto da GeorgiaTech de implementação de vários algoritmos otimizados para a plataforma Cell, o JPEG2000 é um algoritmo de compressão de imagens, um porte de uma implementação já existente já implementado em forma de biblioteca. Ele utiliza complexas estruturas para representação de imagem sendo assim, como descrito na seção 3.1.1, necessário o uso de estruturas para preservar os dados entre os ambientes Java e JNI.

Min-Max Ant System: É uma implementação para o Cell do algoritmo das formigas para achar o caminho mais curto em um grafo. Essa é a implementação parte da biblioteca do projeto **CIV Toolkit**. Ela possui as características que mais se adaptam para o porte para uma biblioteca JNI, utiliza explicitamente a compilação para execução em 32 bits e utiliza a versão 2 da `libspe`. Esse algoritmo foi descartado pois não estava correto, sendo a versão disponível apresentando erro em tempo de execução.

Radix Sort: O projeto implementa o algoritmo de ordenação de mesmo nome, otimizado para a plataforma Cell. Esse algoritmo é uma escolha interessante, pois não é um algoritmo de ordenação que se baseia em comparações e os núcleos SPEs não possuem predição de desvio, sendo a comparação uma operação muito custosa nesses núcleos. Esse projeto utiliza a versão obsoleta da `libspe` e, após uma tentativa sem sucesso de conversão para usar a versão mais recente da biblioteca, ele foi descartado.

Ray tracing on Cell processor: O projeto implementa o algoritmo de *ray tracing* para renderização de cenas tridimensionais em tempo real. Ele foi descartado pois utiliza a biblioteca `libglut` para renderizar o cena usando o servidor X, caindo no caso de uso de bibliotecas externas. O algoritmo também foi implementado como

uma aplicação monolítica, sendo assim não trivial a tarefa de conversão para uma interface de biblioteca.

Um resumo das características não satisfeitas dos projetos não selecionados pode ser visto na tabela 3.1.

		Projetos rejeitados					
		GA for JSSP	Echo Cancel	JPEG2000	Ant System	Radix Sort	Ray tracing
Aspecto técnico	Ambiente 64bits						
	`libspe` obsoleta	•				•	
	bibliotecas externas						•
	execução isolada do SPE						
	Código incompleto		•		•		
	Complexidade			•			

Tabela 3.1: Projetos não selecionados e as características não satisfeitas para o porte JNI.

Capítulo 4

Desenvolvimento de bibliotecas usando Java Native Interface

O processo de integração das plataformas Java e Cell foi realizado através do desenvolvimento de uma biblioteca JNI a partir de uma biblioteca nativa Cell. Primeiramente foi feita a análise e seleção das bibliotecas a serem portadas, como descrito no capítulo anterior. Em seguida foi realizada a implementação da camada entre as suas aplicações utilizando JNI e realizada as alterações necessárias nas bibliotecas originais. Por último foram realizados os testes de desempenho e analisados os resultados obtidos.

4.1 Implementações

As implementações da camada de comunicação entre as duas plataformas para uma biblioteca JNI, apesar de similar, foi realizada individualmente para cada biblioteca portada, seguindo as características originais das bibliotecas nativas para alcançar uma melhor integração. Dependendo da biblioteca a ser portada, maiores ou menores alterações no código original da mesma podem ser necessários para possibilitar a criação da interface JNI. A seguir serão discutidos os detalhes de implementação de cada uma das bibliotecas selecionadas. A primeira implementação a ser discutida é referente a biblioteca **MatMul**.

4.1.1 Implementação da biblioteca MatMul

A aplicação **MatMul** realiza multiplicação de matrizes quadradas grandes, com elementos de ponto flutuante de precisão simples e utilizando os SPEs do processador Cell. Será usada essa biblioteca para substituir uma rotina de multiplicação de matrizes em Java por uma nativa mais eficiente. Temos inicialmente uma classe chamada `Matrix`,

mostrada no código 4.1, que implementa um método estático de multiplicação de matrizes quadradas e devolve uma nova matriz com os resultados.

```
1  public class Matrix {
2          public static float[][] multiply(float[][] A, float[][] B) {
3                  int n = A.length;
4                  float[][] R = new float[n][n];
5                  for (int i = 0; i < n; i++)
6                          for (int j = 0; j < n; j++) {
7                                  R[i][j] = 0;
8                                  for (int k = 0; k < n; k++)
9                                          R[i][j] += A[i][k] * B[k][j];
10                         }
11                 return R;
12         }
13 }
```

Código 4.1: Declaração da classe Java `Matrix`.

A aplicação **MatMul** apresenta uma modelagem modularizada, sua função principal, interpreta os parâmetros de entrada do programa e inicia a execução do algoritmo. No ponto após a interpretação dos parâmetros está a posição ideal para separar o código da rotina principal, descartando o código inicial, e transforma-la em uma função que possa ser utilizada no código de uma outra aplicação. Os parâmetros interpretados pela função são o tamanho da matriz e a quantidade de SPEs a ser utilizada. Existe uma restrição nesse ponto, o algoritmo deve ser executado em uma quantidade de SPEs que seja potência de dois, a dimensão da matriz quadrada devem ser múltipla de 128 elementos, sendo que o tamanho da dimensão da matriz dividido pelo número de SPEs utilizados deve também ser múltiplo de 128 elementos (pois os SPEs trabalham com blocos quadrados de 128 elementos em cada dimensão). A aplicação faz multiplicação entre duas matrizes quadradas constituídas por elementos de ponto flutuante de precisão simples.

Tradicionalmente aplicações Cell possuem alguns elementos comuns e a **MatMul** não é diferente. Uma ou mais estruturas são definidas para representar um contexto contendo um SPE com informações e dados a ele transmitidos e recebidos, nesse caso as estruturas `control_block` (código 4.2), `addr64` (código 4.3) e `pthread_arg` (código 4.4). A primeira é uma estrutura de controle passada por referência para a função `main()` do SPE e contêm as informações relevantes para o SPE executar sua tarefa. Tal estrutura tem sua definição compartilhada por ambos os códigos, para o PPE e para o SPE, e deve ser consistente para ser representada igualmente nas duas arquiteturas. A segunda estrutura é somente para ajudar o SPE na manipulação de endereços da memória principal que são de 64 bits e não 32 bits como na LS do SPE. A terceira estrutura relaciona um contexto SPE com a estrutura de controle passada para o mesmo.

Outro elemento comum a uma aplicação Cell que está presente é a implementação de uma função *thread* responsável por iniciar a execução de um SPE. Isso é feito com o uso de *thread* extra pois a chamada de execução do SPE bloqueia a *thread* corrente. A implementação da função `spe_thread_function()` responsável por essa tarefa pode ser vista no código 4.5. Essa aplicação utiliza o objeto SPE encapsulado do objeto PPE, assim o mesmo é acessível através da referência esterna `matmul_spu`. A declaração da referência pode ser vista no código 4.6 juntamente com as demais variáveis globais utilizadas na aplicação.

```
1  typedef struct _control_block {
2          addr64 matrix_A;
3          addr64 matrix_B;
4          addr64 spe_result;
5          int size;
6          int spe_num;
7          int num_spes;
8          int m_blocks;
9          int n_blocks;
10         int p_blocks;
11         unsigned char pad[74];   /* pad to 128 bytes */
12 } control_block;
```

Código 4.2: Definição da estrutura `control_block`. Ela é usada para representar a informações necessária a serem passadas a um SPE. Usualmente uma referência a uma estrutura desse tipo é passada para o SPE no momento de sua execução.

```
1  typedef union {
2          unsigned long long ull;
3          unsigned int ui[2];
4  } addr64;
```

Código 4.3: Definição da estrutura `addr64`. Ela é usada para facilitar a manipulação de endereços da EA que muitas vezes devem ser tratados em duas partes.

```
1  typedef struct {
2          control_block *cb;
3          spe_context_ptr_t ctx;
4  } pthread_arg;
```

Código 4.4: Definição da estrutura `pthread_arg`. Essa estrutura relaciona um contexto SPE com um bloco de controle, ela é usada como parâmetro para a *thread* que chama a execução de um SPE.

```
1  void *spe_thread_function(void *argp) {
2          int status;
3          unsigned int entry_point = SPE_DEFAULT_ENTRY;
4          struct pthread_arg *parg = (struct pthread_arg *) argp;
5          status = _spe_context_run(parg->ctx, &entry_point, 0, parg->cb,
6                          NULL, NULL);
7          if (status < 0) {
8                  perror("Failed spe_context_run");
9                  exit(1);
10         }
11         _spe_context_destroy(parg->ctx);
12         return (NULL);
13 }
```

Código 4.5: Definição da função `spe_thread_function()`.

```
1  extern spe_program_handle_t matmul_spu;
2
3  void *dlib;
4  spe_context_ptr_t (*_spe_context_create) (unsigned int flags,
5                                            spe_gang_context_ptr_t gang);
6  int (*_spe_context_destroy) (spe_context_ptr_t spe);
7  int (*_spe_context_run) (spe_context_ptr_t spe, unsigned int *entry,
8                           unsigned int runflags, void *argp, void *envp,
9                           spe_stop_info_t *stopinfo);
10 int (*_spe_program_load) (spe_context_ptr_t spe,
11                           spe_program_handle_t *program);
```

Código 4.6: Declaração das variáveis globais da biblioteca JNI `Matrix`.

Originalmente, a aplicação **MatMul** não apresentava uma função para uso como uma biblioteca. Assim, a função principal foi alterada transformando essa aplicação em uma biblioteca. A função criada, chamada `matrix_mult()`, que realiza a multiplicação de duas matrizes, recebe duas matrizes em forma de dois vetores, a matriz de resultado também tem forma de um vetor, a dimensão das matrizes (sendo matrizes quadradas, as dimensões das três matrizes devem ser as mesmas) e o número de SPEs a ser utilizado. O algoritmo do PPE é simples, ele primeiro aloca a estrutura de controle para cada SPE a ser utilizado e preenche com os dados necessários, após ele inicia o contexto para cada SPE, como pode ser visto no código 4.7. Quando os todos os SPEs forem iniciados, um protocolo de sincronização é realizado entre os SPEs e o PPE através de troca de mensagens utilizando *mailbox*. Os SPEs escrevem nos seus *mailboxes* de saída indicando que estão prontos para iniciar o processamento da matriz e leem os seus *mailboxes* de entrada esperando a liberação do PPE. O PPE realiza o processo inverso, lê o *mailbox* de saída dos SPEs esperando estarem prontos e em seguida escreve nos seus

respectivos *mailboxes* de entrada a liberação para começarem os cálculos. O processo de sincronização é feito logo após a inicialização dos SPEs e pode ser visto no código 4.8, que é a segunda parte do código da função `matrix_mult()`. Por último, os SPEs, antes de finalizarem, escrevem em seus *mailboxes* de saída informações de tempo de execução e encerram. Do lado do PPE, após enviar a mensagem para iniciar os cálculos pelos SPEs, ele aguarda lendo essas informações de execução, espera o término das *threads* criadas para execução dos SPEs e retorna.

Tendo a classe Java que realiza a multiplicação de matrizes e a biblioteca nativa com o mesmo propósito, para integrar as duas, uma interface JNI é necessária. Assim, a classe `Matrix` original é alterada para se construir a sua versão JNI, como é mostrado no código 4.9. A classe JNI possui, além do método de multiplicação de matrizes, que permanece com a mesma assinatura com a inclusão do atributo `native`, um método nativo extra, um atributo estático e uma construtora estática. O método nativo `init()` é necessário por uma limitação da arquitetura JNI, é preciso realizar a ligação dinâmica da biblioteca `libspe` do Cell que é utilizada no código nativo. A construtora estática é necessária para carregar a biblioteca nativa antes que a classe possa ser usada e também chamar o método nativo `init()` para realizar a ligação dinâmica com as demais bibliotecas nativas necessárias. Como visto na biblioteca nativa, ela aceita a configuração do número de SPEs a ser utilizado na função de multiplicação, o atributo estático `numSPEs` serve para manter a compatibilidade entre os métodos `multiply()` entre a versão Java e JNI, retirando a necessidade de se passar essa informação para o método da versão JNI. O número de SPEs foi fixado em um, para ficar equiparado com a versão Java que não divide o trabalho de multiplicação de matriz em mais de uma *thread*.

O código 4.10 mostra a assinatura em C dos dois métodos nativos da classe `Matrix`. As assinaturas das versões nativas dos métodos JNI sempre possuem ao menos dois parâmetros, o primeiro é um ponteiro para a estrutura JNI que mantêm referência para todas as funções providas pelo ambiente JNI e o segundo é uma referência para a classe do método nativo, se este for estático, ou para uma instância da classe, se este não for nativo. A implementação nativa no método `init()` é simplesmente o uso da biblioteca de ligação dinâmica, a `dlfcn` (ou *libdl*). A função `dlopen()` carrega a biblioteca desejada, no nosso caso a versão 2 da `libspe` e, em seguida, pega referências para as funções desejadas. Como as funções são utilizadas em vários pontos do código, elas são armazenadas em variáveis globais. A declaração das variáveis pode ser vista no código 4.6.

A implementação da função que realiza a interface direta entre o código Java e a biblioteca nativa é o ponto principal do processo de porte. A implementação do método nativo `multiply()` é o responsável por essa interface encapsulando a chamada para a função nativa `matrix_mult()`. A função da biblioteca nativa necessita de cinco parâ-

```
1  int matrix_mult(float *A, float *B, float *R, int size, int num_spes) {
2          struct pthread_arg parg[num_spes];
3          pthread_t thread[num_spes];
4          spe_spu_control_area_t *ps_area[num_spes];
5          unsigned int spe_time[num_spes], spe_count[num_spes];
6          int status, i r;
7
8          /* Create SPE Threads */
9          for (i = 0; i < num_spes; i++) {
10                 posix_memalign((void **)&(parg[i].cb), 16,
11                                 sizeof(control_block));
12                 parg[i].cb->matrix_A   = (addr64) (unsigned long long)A;
13                 parg[i].cb->matrix_B   = (addr64) (unsigned long long)B;
14                 parg[i].cb->spe_result = (addr64) (unsigned long long)R;
15                 parg[i].cb->size       = (int) size;
16                 parg[i].cb->spe_num    = (int) i;
17                 parg[i].cb->num_spes   = (int) num_spes;
18                 parg[i].cb->m_blocks   = (int) size / MATRIX_DIM;
19                 parg[i].cb->n_blocks   = (int) size / MATRIX_DIM;
20                 parg[i].cb->p_blocks   = (int) size / MATRIX_DIM;
21                 parg[i].ctx  = _spe_context_create(SPE_MAP_PS |
22                                 SPE_EVENTS_ENABLE, NULL);
23                 if (parg[i].ctx == NULL) {
24                         fprintf(stderr, "Failed spe_context_create");
25                         exit(1);
26                 }
27                 if (_spe_program_load(parg[i].ctx, &matmul_spu)) {
28                         fprintf(stderr, "Failed spe_program_load");
29                         exit(1);
30                 }
31                 status = pthread_create(&thread[i], NULL,
32                                 &spe_thread_function, &(parg[i]));
33                 if (status) {
34                         fprintf(stderr, "Erro exit code: %d\n", status);
35                         exit(1);
36                 }
37         }
38
39 /* ... */
```

Código 4.7: Definição da função nativa `matrix_mult()` (parte 1).

metros, três são as matrizes no formato de vetores contínuos (assim como a declaração estática de matrizes em C e não a dinâmica), o tamanho de cada dimensão das matrizes e o número de SPEs a serem utilizados. Um ponto importante a ressaltar que não foi descrito anteriormente é que dadas as limitações da arquitetura Cell as matrizes devem estar alocadas com alinhamento de dezesseis *bytes*. Poderia-se aceitar matrizes com

```
39 /* ... */
40
41         /* wait for a synchronisation signal of each SPE */
42         for (i = 0; i < num_spes; i++)
43                 spe_out_intr_mbox_read(parg[i].ctx, &r, 1,
44                                        SPE_MBOX_ALL_BLOCKING);
45         /* send a start signal to each SPE */
46         r = 0;
47         for (i = 0; i < num_spes; i++)
48                 spe_in_mbox_write(parg[i].ctx, &r, 1,
49                                        SPE_MBOX_ALL_BLOCKING);
50         /* get the performance data of each SPE */
51         for (i = 0; i < num_spes; i++) {
52                 spe_out_intr_mbox_read(parg[i].ctx, &(spe_time[i]), 1,
53                                        SPE_MBOX_ALL_BLOCKING);
54                 spe_out_intr_mbox_read(parg[i].ctx, &(spe_count[i]), 1,
55                                        SPE_MBOX_ALL_BLOCKING);
56         }
57         /* wait until all SPE threads have finished */
58         for (i = 0; i < num_spes; i++) {
59                 if (pthread_join(thread[i], NULL)) {
60                         fprintf(stderr, "FAILED: SPE terminated\n");
61                         exit(1);
62                 }
63         }
64         for (i = 0; i < num_spes; i++)
65                 free(parg[i].cb);
66         return 0;
67 }
```

Código 4.8: Definição da função nativa `matrix_mult()` (parte 2).

```
1  public class Matrix {
2          static int numSPEs;
3          public static native float[][] multiply(float[][] A, float[][]B);
4          public static native void init();
5          static {
6                  System.loadLibrary("Matrix");
7                  numSPESs = 1;
8                  init();
9          }
10 }
```

Código 4.9: Declaração da versão JNI da classe `Matrix`.

qualquer alinhamento e realocá-las para o algoritmo, porém seria um gasto computacional extra e desnecessário já que a alocação das matrizes será feita dentro do código JNI para uso nessa biblioteca, podendo já serem alocadas com o alinhamento correto.

```
1  JNIEXPORT void JNICALL Java_Matrix_init (JNIEnv *env, jclass cls);
2  JNIEXPORT jobjectArray JNICALL Java_Matrix_multiply (JNIEnv *env,
3                                                       jclass cls,
4                                                       jobjectArray mA,
5                                                       jobjectArray mB);
```

Código 4.10: Protótipo do conjunto de funções JNI da classe `Matrix`.

```
1  JNIEXPORT void JNICALL Java_Matrix_init (JNIEnv *env, jclass cls) {
2          dlib = dlopen("/usr/lib/libspe2.so", RTLD_LAZY);
3          if (dlib == NULL) {
4                  fprintf(stderr, "error(1): library can not be opened\n");
5                  fflush(stderr);
6                  return;
7          }
8          _spe_context_create = dlsym(dlib, "spe_context_create");
9          _spe_program_load = dlsym(dlib, "spe_program_load");
10         _spe_context_destroy = dlsym(dlib, "spe_context_destroy");
11         _spe_context_run = dlsym(dlib, "spe_context_run");
12 }
```

Código 4.11: Definição da função JNI `init()`.

O primeiro passo é converter os dados necessários que estão em estruturas Java. O valor do tamanho das matrizes é pego através do uso função JNI `GetArrayLength()` sobre uma das matrizes de entrada. As matrizes em si são transformadas para uma versão nativa com o auxílio da função `getFloatMatrixArray()` vista no código 4.12. Em Java, uma matriz é um vetor de objetos, sendo cada objeto um vetor. No código, após alocar o tamanho necessário em memória, as linhas da matriz são acessadas uma por uma e copiadas para o espaço recém alocado com o uso das funções `GetObjectArrayElement()`, que pega a referência para uma linha da matriz e `GetFloatArrayRegion()` que copia a linha da matriz em Java ao local de memória especificado. O número de SPEs a ser utilizado é guardado em um atributo estático na classe `Matriz`, esse valor é recuperado através do uso conjunto das funções `GetStaticFieldID()` `GetStaticIntField()`. Após alocar a matriz de resultado e com todos os parâmetros necessários convertidos para uma versão que pode ser utilizada pela função `matrix_mult()`, ela então é chamada. Uma vez com o resultado obtido, ele deve ser convertido para uma matriz em Java para poder ser retornado. Sabendo como é composta uma matriz em Java, o primeiro passo é se criar um vetor de objetos com a função `NewObjectArray()`. Em seguida, cada linha da matriz é criada com a função `NewFloatArray`, preenchida com os resultados usando `SetFloatArrayRegion`, atribuída ao vetor de objetos com a função `SetObjectArrayElement` e, por último, a referência a linha pode ser descartada com a função

`DeleteLocalRef`. Com a matriz de resultados preenchida, a memória alocada para os dados nativos pode ser liberada e a referência para a matriz é retornada.

```
1  jfloat* getFloatMatrixArray(JNIEnv *env, jobjectArray obj, jsize size) {
2          int i;
3          jfloat *m;
4          posix_memalign((void **)&m, 16, sizeof(jfloat) * size * size);
5          for (i = 0; i < size; i++) {
6                  jfloatArray line = (*env)->GetObjectArrayElement(env,obj,i);
7                  (*env)->GetFloatArrayRegion(env, line, 0, size, m + i*size);
8          }
9          return m;
10 }
```

Código 4.12: Definição da função JNI `getFloatMatrixArray()`.

4.1.2 Implementação da biblioteca MD5Hash

Sendo o projeto **SPE MD5** a implementação de um *benchmark* sem qualquer interface, o primeiro passo foi definir a interface a ser utilizada para a classe Java que seria a biblioteca. Foi escolhido usar a mesma interface da classe de Hash `MD5`, parte da biblioteca Java. Essa interface é herdada da classe `BaseHash`. Foi decidido manter o código de controle na parte Java da biblioteca, mantendo a similaridade com as classes que serviram de base (`MD5` e `BaseHash`) e reutilizando o código presente nas mesmas. Como o algoritmo de MD não é paralelizável, cada instância da classe se utiliza de um único SPE. E como o algoritmo original do projeto se utiliza da aritmética vetorial da arquitetura SPE para calcular até quatro *hashs* simultâneos, a classe criada teve a sua interface alterada das classes guia para refletir essas características. A classe criada se chama `MD5Hash` e um trecho de sua definição contendo os atributos e métodos extras, assim como as os métodos que tiveram sua interface alterada pode ser vista no código 4.14. A diferença na interface dos métodos das classes de referência é que os métodos que recebem os blocos de dados para processamento e que devolvem os resultados tiveram seus parâmetros transformados em vetores para, simultaneamente, poderem passar dados para o cálculo de mais de um `hash`.

O Algoritmo de MD5 é um algoritmo iterativo, podendo ser executado em mais de um passo. Assim, uma estrutura de controle é necessária para guardar informações sobre o andamento do processo. O código Java mantém os dados relativos ao algoritmo. Porém, são também necessários manter dados relativos ao controle do contexto do SPE e ao *buffer* utilizando no lado JNI do código. Assim, um atributo no código Java foi necessário para dar persistência as informações geradas no lado JNI. Esse atributo é o

```
1  JNIEXPORT jobjectArray JNICALL Java_Matrix_multiply (JNIEnv *env,
2                                                       jclass cls,
3                                                       jobjectArray objA,
4                                                       jobjectArray objB) {
5          int i;
6          jsize size = (*env)->GetArrayLength(env, objA);
7          jfloat *A = getFloatMatrixArray(env, objA, size);
8          jfloat *B = getFloatMatrixArray(env, objB, size);
9          jfloat *R;
10         posix_memalign((void**)&R, 16, sizeof(jfloat) * size * size);
11
12         jfieldID fid = (*env)->GetStaticFieldID(env, cls, "numSPEs", "I");
13         jint num_spes = (*env)->GetStaticIntField(env, cls, fid);
14
15         matrix_mult(A, B, R, size, num_spes);
16
17         jobjectArray result;
18         jclass floatArrCls = (*env)->FindClass(env, "[F");
19         result = (*env)->NewObjectArray(env, size, floatArrCls, NULL);
20
21         for (i = 0; i < size; i++) {
22                 jfloatArray farr = (*env)->NewFloatArray(env, size);
23                 (*env)->SetFloatArrayRegion(env, farr, 0, size, R+i*size);
24                 (*env)->SetObjectArrayElement(env, result, i, farr);
25                 (*env)->DeleteLocalRef(env, farr);
26         }
27
28         free(A);
29         free(B);
30         free(R);
31         return result;
32 }
```

Código 4.13: Definição da função JNI `multiply()`.

`nativeContext` e segue o modelo descrito na seção 3.1.1.

O Código 4.15 mostra os protótipos da funções JNI referentes aos métodos nativos da classe `MD5Hash`. Seguindo o modelo de se manter o código de controle em Java, na parte JNI só se mantém a parte de comunicação com o SPE e com o sistema. Tal comunicação é necessária em quatro momentos, descritos pelas quatro funções JNI. No início da biblioteca é necessário realizar a ligação dinâmica com a biblioteca `libspe`. Isso é feito através da função `staticInit()`, como mostrado no código 4.17, que contém a implementação da função de inicialização JNI, sendo as variáveis globais referenciadas dentro da função mostradas suas declarações no código 4.16. A função `resetBackEnd()` que inicia e reinicia os dados de controle de contexto do objeto JNI e inicia o SPE para poder executar o algoritmo de MD5. A função `fetchResult()` se comunica com SPE

```
1  public class MD5Hash {
2          private static final int BANDWIDTH = 4;
3          private int bandwidth;
4
5          /** Temporary input buffer. */
6          private byte[][] buffer;
7
8          private MD5Hash(String name, int hashSize, int blockSize,
9                          int bandwidth);
10         public void update(byte[][] b);
11         public void update(byte[][] b, int[] offset, int[] len);
12         public byte[][] digest();
13         private byte[][] padBuffer();
14         private byte[][] getResult();
15
16         private byte[] nativeContext;
17
18         private native static int staticInit();
19         private native void resetBackEnd();
20         private native byte[][] fetchResult();
21         private native void transform(byte[][] b, int[] offset);
22
23         static {
24                 System.loadLibrary("MD5Hash");
25                 staticInit();
26         }
27 }
```

Código 4.14: Declaração parcial da classe `MD5Hash` que toma como base a estrutura e algoritmos utilizados nas classes `BaseHash` `MD5`, que são parte do *classpath*. Os métodos nativos são a interface com o código em C, os demais tiveram suas assinaturas modificadas das versões originais e o atributo `nativeContext` é o atributo responsável por manter a persistência dos dados privados da parte JNI da biblioteca e que são únicos para cada instância da classe.

para recuperar a *hash* que é resultado do algoritmo MD5 executado. A função `transform` manda para o SPE os dados e comando para continuar a execução do algoritmo sobre o novo bloco de dados.

O projeto **SPE MD5** possuía grandes limitações que foram contornadas com a implementação de uma casca para contorna-las. Dentre as limitações e as medidas tomadas se destacam:

Conversão entre *big endian* e *little endian*: O MD5 funciona tratando o bloco de dados a ser codificado como um vetor de inteiros de 32 bits em formato *little endian*. Contudo, a arquitetura SPE somente suporta dados em formato *big endian*, assim o bloco de dados deve ser reorganizado para se obter o resultado esperado na

```
1 JNIEXPORT jint JNICALL Java_MD5Hash_staticInit (JNIEnv *env, jclass class);
2 JNIEXPORT void JNICALL Java_MD5Hash_resetBackEnd (JNIEnv *env, jobject obj);
3 JNIEXPORT jobjectArray JNICALL Java_MD5Hash_fetchResult (JNIEnv *env,
4                                                         jobject obj);
5 JNIEXPORT void JNICALL Java_MD5Hash_transform (JNIEnv *env, jobject,
6                                                jobjectArray b,
7                                                jintArray offset);
```

Código 4.15: Protótipo do conjunto de funções JNI da classe `MD5Hash`, onde a função `StaticInit()` inicializa a biblioteca JNI, `resetBackEnd()` reinicia o estancia da classe para futuro reuso, `fetchResult()` recupera do SPE o resultado do cálculo do algoritmo de MD5 e `transform()` envia para o SPE um bloco de dados para ser calculado.

```
1 void *dlib;
2 spe_context_ptr_t (*_spe_context_create) (unsigned int flags,
3                                           spe_gang_context_ptr_t gang);
4 int (*_spe_context_run) (spe_context_ptr_t spe, unsigned int *entry,
5                          unsigned int runflags, void *argp, void *envp,
6                          spe_stop_info_t *stopinfo);
7 int (*_spe_program_load) (spe_context_ptr_t spe,
8                           spe_program_handle_t *program);
9 int (*_spe_context_destroy) (spe_context_ptr_t spe);
```

Código 4.16: Declaração das variáveis globais da biblioteca JNI `MD5Hash`, elas são ponteiros para as funções da biblioteca `libspe` que é ligada dinamicamente na inicialização da biblioteca.

```
1  JNIEXPORT jint JNICALL Java_MD5Hash_staticInit(JNIEnv *env, jobject this) {
2          dlib = dlopen("/usr/lib/libspe2.so", RTLD_LAZY);
3          if (dlib == NULL) {
4                  fprintf(stderr, "error(1): library can't be opened\n");
5                  fflush(stderr);
6                  return -1;
7          }
8          _spe_context_create = dlsym(dlib, "spe_context_create");
9          _spe_program_load = dlsym(dlib, "spe_program_load");
10         _spe_context_destroy = dlsym(dlib, "spe_context_destroy");
11         _spe_context_run = dlsym(dlib, "spe_context_run");
12         return 0;
13 }
```

Código 4.17: Definição da função `staticInit()` responsável por inicializar as variáveis globais utilizadas pela biblioteca JNI, onde a declaração das variáveis globais utilizadas dentro da função podem ser vistas no código 4.16.

aplicação do algoritmo.

Transferência de dados: O Algoritmo MD5 executa em blocos de dados de 64 bytes, sendo que a implementação original aqui utilizada executava sobre um bloco de dados de 64 bytes diretamente incluído no código do programa. Essa limitação teve que ser superada, fazendo com que o SPE recebesse quantidades arbitrárias de blocos de dados.

Intercalação de dados: A implementação original desse projeto, usando as instruções vetoriais do processador SPE, executava o algoritmo de MD5 simultaneamente sobre 4 blocos de dados diferentes. Sendo os vetores de entradas compostos por quatro elementos de 4 bytes, cada operação executava sobre uma palavra de cada um dos blocos de dados ao mesmo tempo, tendo em vista o algoritmo MD5 tratando um bloco de dados como um vetor de inteiros de 4 bytes. Cada operando vetorial é organizado em memória como dezesseis bytes consecutivos, sendo necessário que cada conjunto de quatro bytes venha de um bloco de dados diferente de forma a calcular os quatro `hashs` diferentes prometidos pelo algoritmo. Assim, usando dados arbitrários, é necessário primeiro combinar os blocos de dados de forma que o algoritmo possa ler de uma vez os quatro blocos em um único operando vetorial.

O problema da transferência de dados para alimentar o algoritmo sendo executado no SPE foi abordado implementando-se uma comunicação entre o PPE e o SPE para prover o segundo com as informações necessárias para que ele possa iniciar DMAs de conjuntos compostos por quatro blocos de dados individuais para alimentar o algoritmo MD5. Foi utilizada a técnica de *double buffer* para prover melhor desempenho ao sistema.

Os problemas do formato dos dados e de se combinar quatro blocos em um único bloco de dados intercalado foram resolvidos simultaneamente pelo uso conjunto de duas instruções do SPE. Foi utilizada a biblioteca para o SPE que provê uma extensão do conjunto de instruções da arquitetura para uso direto na linguagem C. As instruções utilizadas foram as instruções `or` e `shuffle`, suas assinaturas podem ser vistas nos códigos 4.18 e 4.19 respectivamente. A instrução `or` é o conhecido ou lógico bit a bit e a instrução `shuffle` possibilita montar um vetor byte a byte escolhendo-se através de um vetor de índices, bytes arbitrários provindos de dois vetores de entrada. Isso possibilita ao mesmo tempo inverter a ordem dos bytes em uma palavra de 4 bytes e intercalar palavras de 4 bytes de dois vetores diferentes. Assim, é possível montar uma variável vetorial de quatro posição com cada uma de suas posições vindas de um bloco de dados diferente e com cada palavra com a posição de seus bytes invertidas em apenas 3

instruções em C que são mapeadas na proporção de um para um para instruções de máquina.

```
1 d = spu_or(a, b);
```

Código 4.18: Protótipo da função `spu_or()`, onde **a**, **b** e **c** são variáveis vetoriais dos tipos básicos em C, sendo que todas devem ser do mesmo tipo. Essa instrução realiza a operação de **ou lógico** bit a bit entre os vetores **a** e **b** e guarda o resultado em **d**.

```
1 d = spu_shuffle(a, b, pattern);
```

Código 4.19: Protótipo da função `spu_shuffle()`, onde **a**, **b** e **c** são variáveis vetoriais dos tipos básicos em C, sendo que todas devem ser do mesmo tipo e **pattern** uma variavel vetorial do tipo `char`. Essa instrução monta um vetor de resultado **d**, baseando-se no vetor **pattern** para escolher cada byte do vetor resultado se é uma cópia de um byte arbitrário dos vetores **a** ou **b** ou ainda se é um dos valores fixos 0x00, 0xff ou 0x80.

4.1.3 Implementação da biblioteca RC5

O projeto do RC5 original possui uma interface simples e não possuia possibilidades de configuração ou controle, trazendo algumas limitações. O número de SPEs utilizados é fixado em tempo de compilação. O tamanho do bloco de dados deve ser uma potência de dois elementos (sendo cada elemento de 4 bytes) segundo a especificação. A chave de criptografia é gerada aleatoriamente internamente e mantida durante uma sessão de execução da biblioteca, impedindo que se possa criptografar dados em uma execução e descriptografar em outra. O algoritmo funciona dividindo o bloco de dados a ser criptografado em partes iguais para todos os SPEs utilizados, sem tratar os casos em que a divisão possui resto. Assim, como a entrada é uma potência de dois, o algoritmo divide imprecisamente quando o número de SPEs não é uma potência de dois (três, cinco, seis e assim por diante).

A biblioteca original possui muitas limitações, como apontado acima e, buscando manter o menor número de modificações na estrutura interna da biblioteca, somente uma alteração foi realizada. A forma como é definida a quantidade de SPEs utilizados pelo algoritmo, inicialmente feito em tempo de compilação, foi alteranda para que se possa definir, em tempo de execução, durante a função de inicialização do algoritmo. A interface de funções da biblioteca é mostrada no código 4.20, com a devida alteração na assinatura da função de inicialização `rc5_Init()` alterada para receber como parâmetro o número de SPEs a ser utilizado. Internamente, o código que envia uma estrutura de

informações para os SPEs foi modificado para refletir a nova característica, assim como o código no lado do SPE que lê as informações. Tal informação é a de que, originalmente, cada SPE recebia o endereço do pedaço do bloco de dados que lhe cabia tratar. Porém, o tamanho do pedaço do bloco não era passado, e sim, em seu lugar, o tamanho total do bloco de dados. Assim, cada SPE dividia o tamanho total pelo número de SPEs utilizados, que também era definido explicitamente no código fonte do SPE. Com a alteração, o tamanho repassado para cada SPE é somente o do pedaço do bloco de dados que cada SPE compete tratar, sendo então retirada a parte da divisão no código do mesmo e a necessidade de se saber a quantidade de SPEs sendo utilizada.

```
1 int rc5_Init (int spusnum);
2 void rc5_Encrypt (unsigned int *input_data, unsigned int *encrypted_data,
3                   unsigned int DATASIZE);
4 void rc5_Decrypt (unsigned int *encrypted_data, unsigned int *decrypted_data,
5                   unsigned int DATASIZE);
6 int rc5_Check (unsigned int *input_data, unsigned int *decrypted_data,
7                 unsigned int DATASIZE);
```

Código 4.20: Protótipo do conjunto de funções da biblioteca `RC5`, `rc5_Init()` é a função de inicialização da biblioteca e deve ser chamada antes de se utilizar as funções de criptografia, onde **spusnum** se refere ao número de SPEs que se deseja utilizar para se realizar as operações de criptografia. `rc5_Encrypt()` e `rc5_Decrypt()` são as funções para criptografar e descriptografar os dados respectivamente, sendo na primeira os parâmetros **input_data** o ponteiro para o bloco de dados não criptografados e **encrypted_data** o ponteiro para o bloco de memória onde os dados criptografados serão armazenados, na segunda os parâmetros **encrypted_data** o ponteiro pra os dados já criptografados e **decrypted_data** o ponteiro para onde os dados descriptografados serão armazenados e em ambas **DATASIZE** o tamanho do bloco de dados a ser tratado. A última função, `rc5_Check()`, é uma função que compara o bloco de dados original com o bloco de dados descriptografados e retorna se são consistentes um com o outro, onde **input_data** é o ponteiro para o bloco de dados original, **decrypted_data** o ponteiro para o bloco de dados criptografado, **DATASIZE** o tamanho dos blocos que devem ser os mesmos.

Da parte das alterações conceituais, o código original usava ligação padrão, implícita da biblioteca `libspe` com a aplicação, e assim, como descrito anteriormente nessa seção, foi necessário alterar o código para usar uma ligação dinâmica explícita, incluindo no código de inicialização da biblioteca as chamadas de função que realizam a ligação dinâmica.

A biblioteca JNI foi modelada usando uma classe com métodos estáticos para representar as funções da biblioteca RC5 original, possuindo os métodos os mesmos parâmetros que a original como pode ser visto no código 4.21. Caso não houvesse a limitação

da chave de criptografia ser interna à biblioteca, a mesma deveria ter uma estrutura em Java para representa-la e ser incluída como parâmetro nos métodos de criptografar e descriptografar dados, ou então deixando os métodos da biblioteca de serem estáticos, ela poderia ser incluída na classe através de um atributo, sendo setada na função de inicialização e assim podendo ser utilizada sem a necessidade de passá-la como um parâmetro.

```
1  public class RC5 {
2          static boolean initialized;
3
4          private static native int Init(int spesnum);
5          public static native byte[] Encrypt(byte[] data);
6          public static native byte[] Decrypt(byte[] data);
7          public static native int Check(byte[] data1, byte[] data2);
8
9          RC5 (int spesnum) {
10                 if (initialized == false) {
11                         initialized = true;
12                         Init(spesnum);
13                 }
14         }
15
16         static {
17                 System.loadLibrary("RC5");
18         }
19 }
```

Código 4.21: Declaração da classe `RC5`

A representação JNI dos métodos nativos são mostrados no código 4.22. Assim como todas as funções JNI, possui dois parâmetros extras, o primeiro sendo o ponteiro para a estrutura JNI e o segundo sendo uma referência para a classe da qual a função JNI é parte, caso essa seja estática, ou uma referência para o objeto da classe, caso não seja estática.

```
1 JNIEXPORT jint JNICALL Java_RC5_Init (JNIEnv *env, jclass class,
2                                       jint spesnum);
3 JNIEXPORT jbyteArray JNICALL Java_RC5_Encrypt (JNIEnv *env, jclass class,
4                                                jbyteArray data);
5 JNIEXPORT jbyteArray JNICALL Java_RC5_Decrypt (JNIEnv *env, jclass class,
6                                                jbyteArray data);
7 JNIEXPORT jint JNICALL Java_RC5_Check (JNIEnv *env, jclass class,
8                                        jbyteArray data1, jbyteArray data2);
```

Código 4.22: Protótipo do conjunto de funções JNI dos métodos nativos da classe `RC5`.

A representação entre as funções e métodos da biblioteca JNI é de um para um com as funções da biblioteca original. O mesmo segue para os parâmetros das funções, sendo a única diferença, o parâmetro que identifica o tamanho dos blocos de dados retirados dos métodos nativos, pois como os vetores em Java possuem internamente informação sobre o seus tamanhos, essa informação não é mais necessária se o bloco de dados preenche totalmente o vetor que o mantêm, como na modelagem utilizada. Nas funções JNI, o tamanho de um vetor é adquirido através do uso da função `GetArrayLength()` parte da biblioteca JNI.

4.2 Resultados

Os resultados das implementações das bibliotecas selecionadas são apresentados a seguir. As principais contribuições são mostrar a viabilidade de se construir uma interface entre um código Java e bibliotecas nativas pré-existentes e apresentar um modelo factível para a execução desta tarefa, aumentando o desempenho do código Java. A Plataforma de teste utilizada foi um Playstation 3, que possui um processador Cell com um PPE com *dual thread* e seis SPEs disponíveis (O Processador Cell fabricado possui oito SPEs, porém a versão para o Playstation 3 possui um desativado por controle de qualidade e um é travado para uso exclusivo do *firmware* do *videogame*) executando a 3.2GHz e 256MB de memória principal, sendo cerca de 220MB disponível para uso com o sistema operacional. O sistema operacional é uma versão do Linux versão 2.6.24 adaptada para suporte ao processador Cell.

Para medir o tempo de execução dos testes realizados foi usado o programa **gnu time**. O programa retorna varias informações sobre a execução de um processo, dentre elas as utilizadas foram o tempo de execução do processo em modo usuário (t_{user}), tempo de execução do processo em modo *kernel* (t_{system}), tempo de execução real do processo ($t_{elapsed}$) e a porcentagem de uso do processador pelo processo (cpu_{usage}). A porcentagem de uso do processador é dado pela seguinte equação: $cpu_{usage} = \frac{t_{user}+t_{system}}{t_{elapsed}}$. Os tempos dados pelo t_{user} e t_{system} são a soma do tempo de uso, nos seus respectivos modos, de todos os núcleos presente no sistema. Assim, um processo executando em uma plataforma com dois núcleos pode ter o t_{user} somado ao t_{system} igual ao dobro do $t_{elapsed}$, o tempo real de execução do processo. Por essa razão, em arquiteturas *multicore* (ou com tecnologias que emulam mais de um núcleo para o sistema operacional, como a *hyper-threading* ou o *dual thread* do PPE na arquitetura Cell) é possível que a porcentagem de uso da CPU seja maior do que 100%, sendo que o máximo possível o número de núcleos da arquitetura vezes 100%. Um último ponto de observação sobre o **gnu time** na plataforma Cell é que ele não leva em conta o uso dos SPEs para o cálculo de uso do processador tanto em modo usuário quanto em modo *kernel*. Assim um processo

que execute exclusivamente em SPE terá um t_{user} e t_{system} iguais ou próximos de zero, consequentemente o cpu_{usage} tende a zero. Como o $t_{elapsed}$ é o tempo real de execução de um processo ele não é afetado por essa peculiaridade.

Medir o tempo de execução de um processo utilizando a soma de t_{user} e t_{system} seria mais preciso pois não conta o tempo em que um processo aguarda para ser escalonado para uso do processador e diminui a influencia na avaliação de desempenho que possa ser ocasionado por outros processos que estejam simultaneamente executando no sistema. Porém, dado as características citadas do comportamento do **gnu time** na plataforma Cell, não é possível utilizar os t_{user} e t_{system} para uma analise razoável de desempenho e assim sendo, nas avaliações que seguem é utilizado sempre o valor de $t_{elapsed}$.

Todos os testes para as bibliotecas implementadas foram executados dez vezes e os dados apresentados são a média aritmética simples entre todas as execuções.

4.2.1 MatMul

Para realizar a comparação de desempenho da biblioteca **MatMul** foi realizado testes sobre a plataforma Cell e a plataforma x86. Para a platadorma Cell foi implementada a aplicação Java mostrada no código 4.23. Essa aplicação pode executar a biblioteca JNI MatMul ou realizar a multiplicação de matrizes em ambiente Java de forma análoga a classe Java `Matrix` apresentada (Código 4.1). Ainda para a plataforma Java foi utilizada a aplicação nativa matmul original (a mesma que foi utilizada de base para a biblioteca JNI). Para a plataforma x86 foi utilizado a mesma aplicação Java. Entretanto, a biblioteca JNI utilizada no Cell foi substituída por uma adequada à plataforma x86, que realiza a multiplicação de matriz de forma análoga ao algoritmo apresentado na classe java `Matrix` (Código 4.1).

Foram realizados 2 testes comparativos, o primeiro comparando a biblioteca JNI para o Cell com a execução puramente em Java para o Cell e x86 e a biblioteca JNI também para a arquitetura x86. O segundo teste realiza a comparação entre a configuração da biblioteca JNI para uso de um, dois e quatro SPEs e em relação a aplicação original da qual a biblioteca foi retirada. O objetivo do primeiro teste foi determinar a eficiência tanto entre a aplicação Java e a JNI quanto entre as plataformas Cell e x86. Foi utilizada a configuração da biblioteca JNI para o Cell que utiliza somente um SPE. Essa escolha foi feita para tornar homogênea a comparação com as demais aplicações que utilizam uma única *thread* e não paralelizam a computação do resultado, o que ocorreria com o uso de mais de um SPE na biblioteca JNI para o Cell. O segundo teste teve por objetivo observar a escalabilidade da aplicação através do número de SPEs utilizados e observar o *overhead* na biblioteca, causado pela interface entre o código JNI e Java, comparando

o desempenho desta com a aplicação nativa original.

A plataforma x86 utilizada foi um processador Intel Pentium 4 executando a 2.80GHz com 2GB de memória principal e sistema operacional Linux versão 2.6.9.

O gráfico na figura 4.1 mostra o uso da CPU no primeiro teste. É esperado que o uso da CPU seja mais próximo do máximo quanto maior o tamanho do conjunto de dados pois o tempo de preparo do processo, de interação com o ambiente e de espera pela CPU se torna mínimo em relação ao tempo de computação da matriz (que é uma tarefa isolada e intensiva). Para matriz de dimensão 128 é clara a interferência, em todas as versões testadas, das situações descritas anteriormente. Para matrizes maiores, as versões Java e JNI para a plataforma x86 seguem o comportamento esperado atingindo valores muito similares. A versão JNI pra Cell segue o inverso, com o processador menos ocupado para matrizes de tamanho cada vez maiores. Isso se deve ao fato do uso da CPU só levar em consideração o uso do PPE e como o trabalho intensivo é deslocado para o SPE mais ocioso o PPE fica pois só realiza funções de auxilo e coordenação dos SPEs.

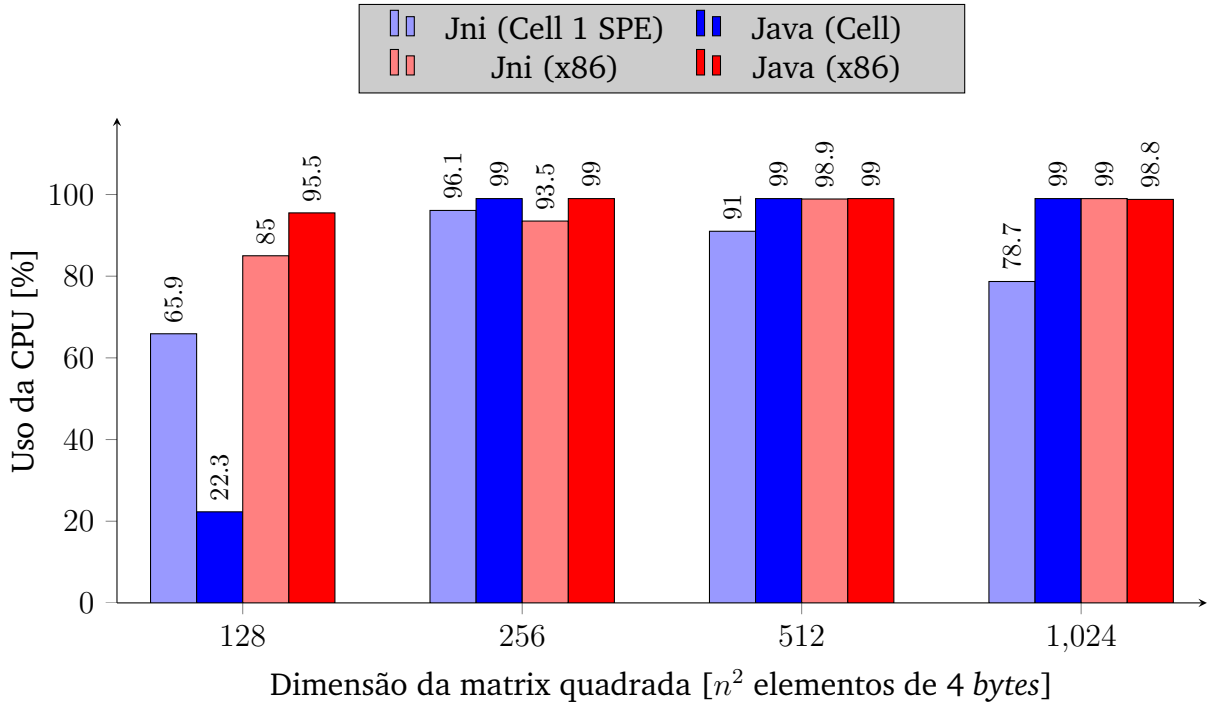

Figura 4.1: Gráfico de uso da CPU pela biblioteca `MatMul` nas versões JNI e Java para as plataformas x86 e Cell. A versão da biblioteca JNI para o Cell utiliza um SPE em sua execução.

O gráfico na figura 4.2 mostra o tempo de execução para cada uma das implementações. Todas as versões menos a JNI para o Cell crescem exponencialmente com o

aumento do tamanho das matrizes de entrada. Neste gráfico fica clara a vantagem da arquitetura Cell para processamento de cálculos intensivos. O ponto interessante do gráfico é o desempenho da versão JNI do Cell para matrizes de dimensão 128 que é inferior a versão JNI para a plataforma x86 e inferior até mesmo a mesma versão para matrizes maiores, de dimensão 256 e 512. Isso se deve possivelmente ao fato da necessidade de transferir dados entre a memória principal e a memória local do SPE, pois o barramento de grande largura possui melhor desempenho transferindo grandes quantidades de dados.

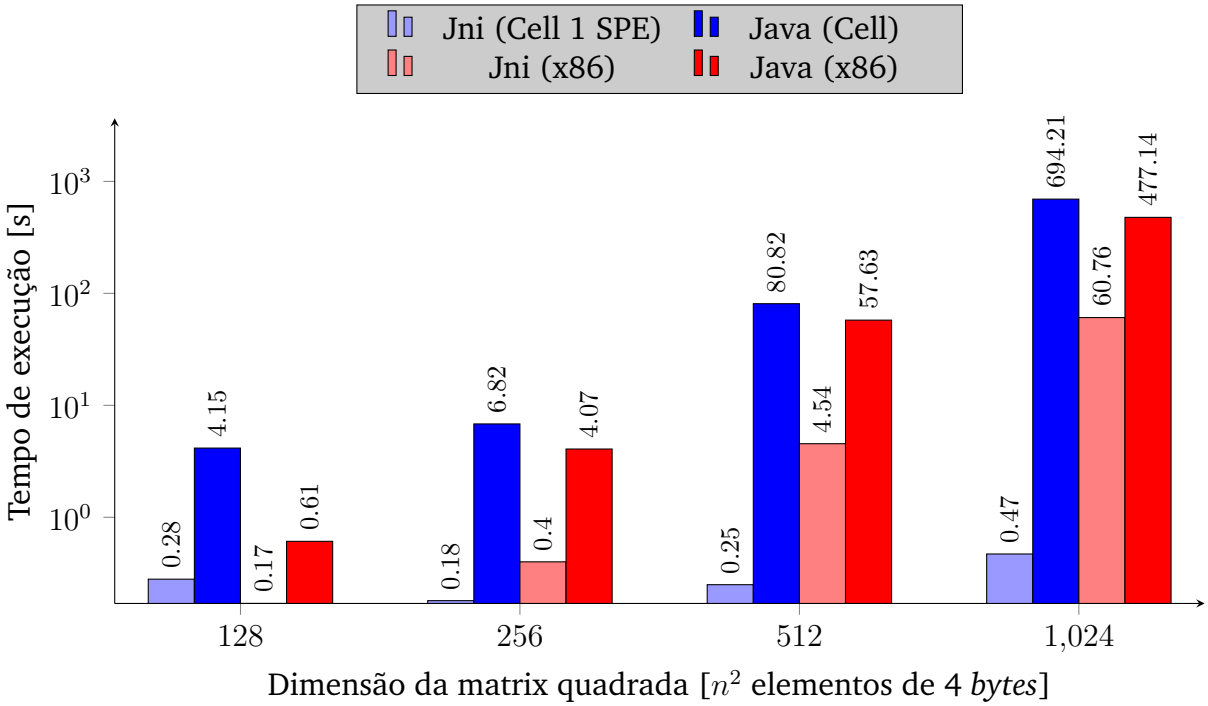

Figura 4.2: Gráfico do tempo de execução da biblioteca `MaiMul` nas versões JNI e Java para as plataformas x86 e Cell. O eixo do tempo mostra o tempo real de execução do processo e esta em escala logarítmica. A versão da biblioteca JNI para o Cell utiliza um SPE em sua execução.

O gráfico da figura 4.3 mostra o uso da CPU no segundo teste. Seguindo o raciocínio anterior, é esperado que o tempo de uso da cpu diminua em relação ao tamanho das matrizes processadas e esse comportamento pode ser observado no gráfico. O Uso da CPU aumenta em relação a quantidade de SPEs usados, isso se deve a maior atividade do PPE para inicializar e gerenciar a maior quantidade de SPEs ativos. É interessante notar que o uso da CPU pela aplicação JNI cai drasticamente para matrizes de dimensão 4096 e se torna praticamente a mesma independente do número de SPEs utilizado. Isso se deve ao uso da memória virtual pela aplicação aumentando drasticamente o tempo

de execução e o tempo de ociosidade da CPU.

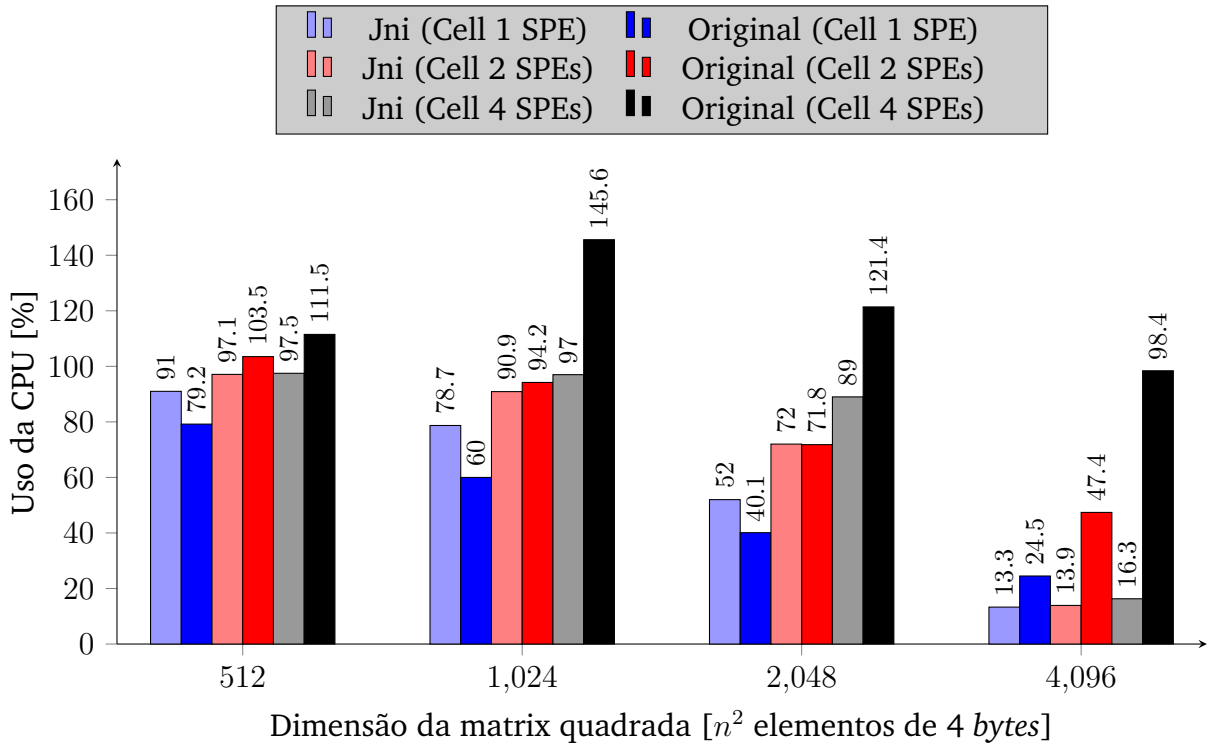

Figura 4.3: Gráfico de uso da CPU pela biblioteca `MatMul` JNI e pela aplicação original de que ela foi extraída. As aplicações foram executadas utilizando-se um, dois e quatro SPEs.

O gráfico da figura 4.4 mostra o tempo de execução da aplicação original e da biblioteca JNI, usando um, dois e quatro SPEs. É possível observar o aumento de tempo destoante na biblioteca JNI para a matrizes de dimensão 4096, evidenciando, como dito anteriormente, o uso da memória virtual pela aplicação. Tanto a aplicação original quanto a JNI começam a apresentar uma escalabilidade em relação a quantidade de SPEs utilizados para entradas de dimensão 2048 e maiores. Essa escalabilidade só impedida pela capacidade de memória do sistema, pois o uso da memória virtual aumenta o tempo de execução de forma a tornar desprezível o ganho de desempenho devido a um maior número de SPEs. O gráfico também mostra o *overhead* gerado pela interface Java e JNI em relação a aplicação original. Sendo tal *overhead* aumenta em menor proporção relativo ao tempo de execução e o tamanho dos dados de entrada, tornando o desempenho da biblioteca JNI mais próxima da original quanto maior o tamanho da entrada. A tabela 4.1 mostra mais detalhadamente a diferença do tempo de execução entre o uso de um, dois e quatro SPEs para a aplicação nativa original na plataforma Cell e a biblioteca JNI também na plataforma Cell.

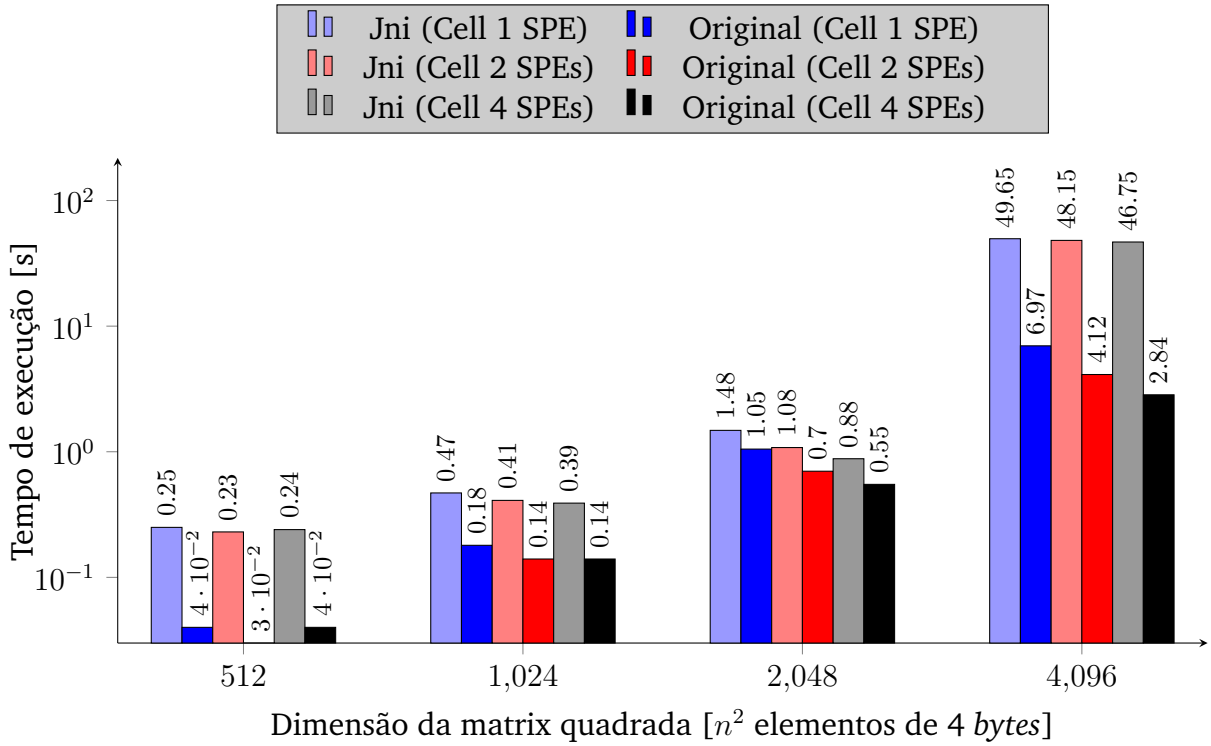

Figura 4.4: Gráfico do tempo de execução da biblioteca `MatMul` JNI e da aplicação original de que ela foi extraída. O eixo do tempo mostra o tempo real de execução do processo e esta em escala logarítmica. As aplicações foram executadas utilizando-se um, dois e quatro SPEs.

Os resultados dos testes executados demostram a vantagem da biblioteca *MatMul* JNI para o Cell em relação a versões Java ou JNI para arquitetura Cell ou x86, demonstrando a capacidade de escalabilidade para processamento intensivo de grandes quantidades de dados tirando proveito das características que a arquitetura Cell tem a oferecer. Ainda mostra a proximidade de desempenho entre uma implementação JNI e uma implementação totalmente nativa, sendo sua desvantagem a utilização de uma maior quantidade de memória.

4.2.2 MD5Hash

O algoritmo original do MD5 não possuía as características mais propícias para o porte simples em uma biblioteca JNI. Ele porém é um algoritmo otimizado para o Cell e apresenta a possibilidade de mostrar os diferentes cenários a se enfrentar para o porte de um código para o Cell para uma versão JNI. Foi necessário construir uma interface de controle com o PPE. O algoritmo MD5 é um algoritmo interativo, que pode ser alimen-

		Tempo de execução pelo número de SPEs utilizados[s]				
datasize[MB]	**versão**	**1 SPE**	**2 SPEs**	**4 SPEs**	**2 SPEs - 1 SPE**	**4SPEs - 1 SPE**
512	jni	0.25	0.23	0.24	-0.02	-0.01
	original	0.04	0.03	0.04	-0.01	-0.00
1024	jni	0.47	0.41	0.39	-0.06	-0.08
	original	0.18	0.14	0.14	-0.02	-0.02
2048	jni	1.48	1.08	0.88	-0.40	-0.60
	original	1.05	0.70	0.55	-0.35	-0.50
4096	jni	49.65	48.15	46.75	-1.50	-2.90
	original	6.97	4.12	2.84	-2.85	-4.13

Tabela 4.1: Comparativo da escalabilidade da biblioteca JNI **MatMul** em relação a aplicação original de que biblioteca JNI foi extraída (aplicação **MatMul** nativa para a plataforma Cell). A tabela mostra os tempos de execução para a versão JNI da biblioteca e da aplicação original para um, dois e quatro SPEs. As suas últimas colunas mostram respectivamente a diferença entre o tempo de execução de um SPE e dois SPEs e entre um SPE e quatro SPEs.

tado com dados em etapas. Assim, um código JNI de persistência de dados teve que ser construído, mostrando uma das situações mais comuns a se enfrentar no porte de uma biblioteca, manter dados persistentes consistentes entre dois paradigmas de programação diferentes. A biblioteca teve sua interface completamente implementada, contudo, não foi possível eliminar completamente problemas na comunicação entre PPE e SPE para que o desempenho da biblioteca pude-se ser avaliada. Sendo o problema não eliminado relativo a comunicação entre PPE e SPE, que não fazem parte da interface propriamente dita, a mesma pode ser considerada concluída e serve como base para implementações de outras bibliotecas, tendo sido empregada como base para implementação da interface da biblioteca RC5.

4.2.3 RC5

A interface para a biblioteca RC5 foi concluída com sucesso dentro das limitações da biblioteca original. Um conjunto de testes foi realizado para testar a capacidade do algoritmo e o impacto da interface JNI no mesmo. Lembrando que o algoritmo RC5 utilizado somente foi codificado para uso de quatro SPEs. O Algoritmo foi executado em blocos de dados de tamanho de 1MB, incrementando em potência de 2 até 256MB. Sendo que o algoritmo executa sobre conjuntos de inteiros de quatro *bytes*, a quantidade

de elementos é quatro vezes menor que o tamanho do bloco de dados.

Foram executados dois exemplos, um escrito em C e utilizando a biblioteca original e outro escrito em java utilizando a biblioteca JNI. Ambos inicializam a biblioteca para uso de quatro SPEs, que é a quantidade definida no algoritmo original. O exemplo aloca a quantidade desejada de dados e executa em sequência a codificação e decodificação dos dados. Uma versão em que se realiza a checagem dos dados decodificados com os dados originais foi feita para garantir que a biblioteca criptografa e descriptografa corretamente os dados. Os testes foram executados dez vezes, tanto para a verificação de consistência do algoritmo, quanto para a medição de desempenho. O algoritmo JNI falhou em todas as rodadas para bloco de dados 256MB, isso se deve a memória disponível no sistema, pois essa versão usa muito mais memória que a versão nativa, dado a cópia de dados entre as estruturas Java e JNI, consequentemente em todos os teste usando blocos de 256MB o programa terminou precipitadamente com código de erro informando que o sistema estava sem memória suficiente.

As medições de desempenho para a biblioteca foram realizadas usando as versões que não fazem a checagem de correção e e como já dito no começo dessa sessão foi usado o programa **gnu time** para análise de tempo. O gráfico da figura 4.5 mostra o uso do processador para cada uma das configurações. Pode-se observar que a partir de 64MB o uso do processador para a versão JNI cai drasticamente, sendo resultado do uso da memória virtual dado a limitação de memória da plataforma de teste utilizada. O mesmo acontece com a versão nativa que tem seu desempenho afetado drasticamente a partir de blocos de dados de 128MB, evidenciando a diferença do consumo de memória entre as duas versões. O uso da cpu é sempre menor para a versão JNI devido a, por exemplo, o *overhead* gerado pela cópia de dados entre o ambiente Java e JNI deixando o processo mais tempo ocioso esperando por DMA, pelas a maiores trocas de contexto entre o processo em ambiente ambientes Java e o processo em ambiente JNI e pela a execução de processos internos da JVM.

A figura 4.6 mostra o tempo de execução do exemplo. Os resultados possuem crescimento exponencial de acordo com o crescimento do bloco de dados de entrada também exponencial. O tempo de execução da versão JNI aumenta drasticamente para blocos de entrada de 64MB e 128MB e o tempo de execução da versão nativa aumenta significativamente para blocos de entrada de 128MB. Como já observado anteriormente, esse comportamento se deve à limitação de memória da plataforma e o consequente uso de memória virtual. Pode-se notar uma anomalia nas versões JNI e nativa para blocos de dados de 1MB. Um suposição para tal anomalia pode-se ser o fato de o barramento de transferência de dados entre a memória principal e as memórias locais dos SPEs ter melhor desempenho para transferência de grande volume de dados. Outra possibilidade é a concorrência entre os SPEs e o PPE pelo mesmo barramento ser muito constante dado

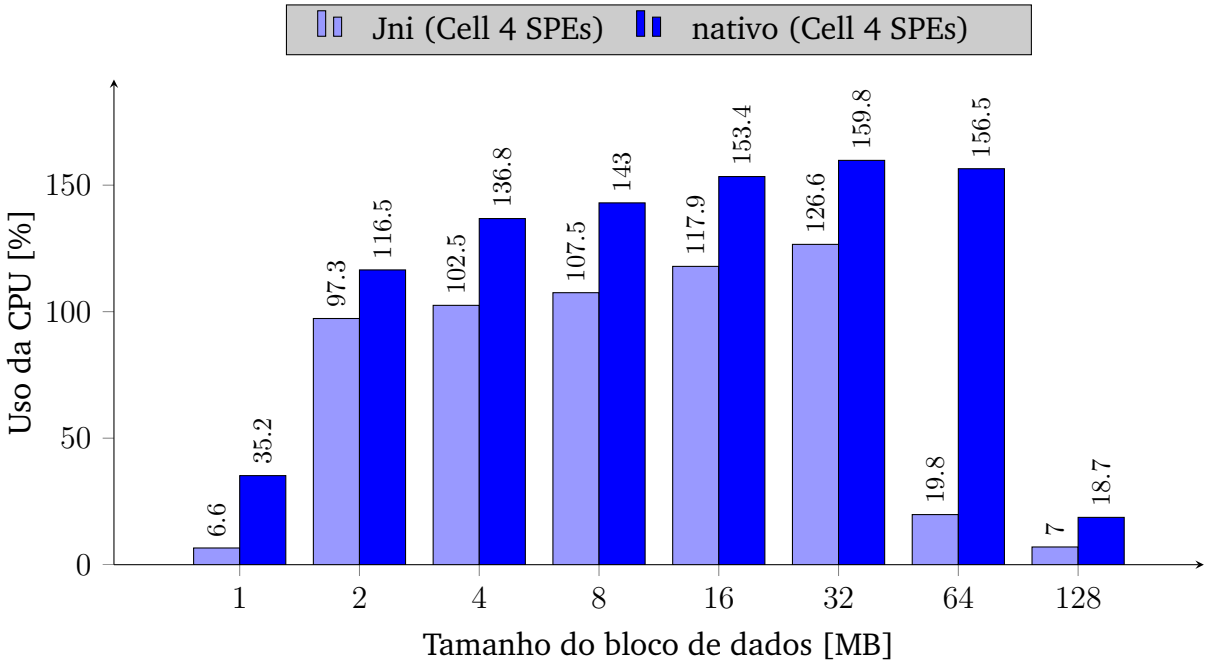

Figura 4.5: Gráfico de uso da CPU pelas bibliotecas RC5 JNI e nativa, o gráfico mostra a média de dez execuções para cada uma das configurações.

o pequeno volume de dados e resultar em muitas colisões.

O Desempenho da biblioteca JNI foi próximo do desempenho da biblioteca nativa nos casos normais (ou seja, excluindo o primeiro e os casos em que ouve uso de memória virtual). O tempo de execução extra se deve ao gasto extra de processamento devido a interface JNI. Ainda, tal gasto extra não é proporcional ao tamanho do bloco de dados de entrada, sendo constante. Isso pode ser observado através da diferença entre os tempos da biblioteca nativa e da biblioteca JNI de acordo com o tamanho da entrada (0,3 segundos em média). Mais detalhes dos dados de execução podem ser vistos na tabela 4.2. O resultado obtido é satisfatório e valida a abordagem proposta.

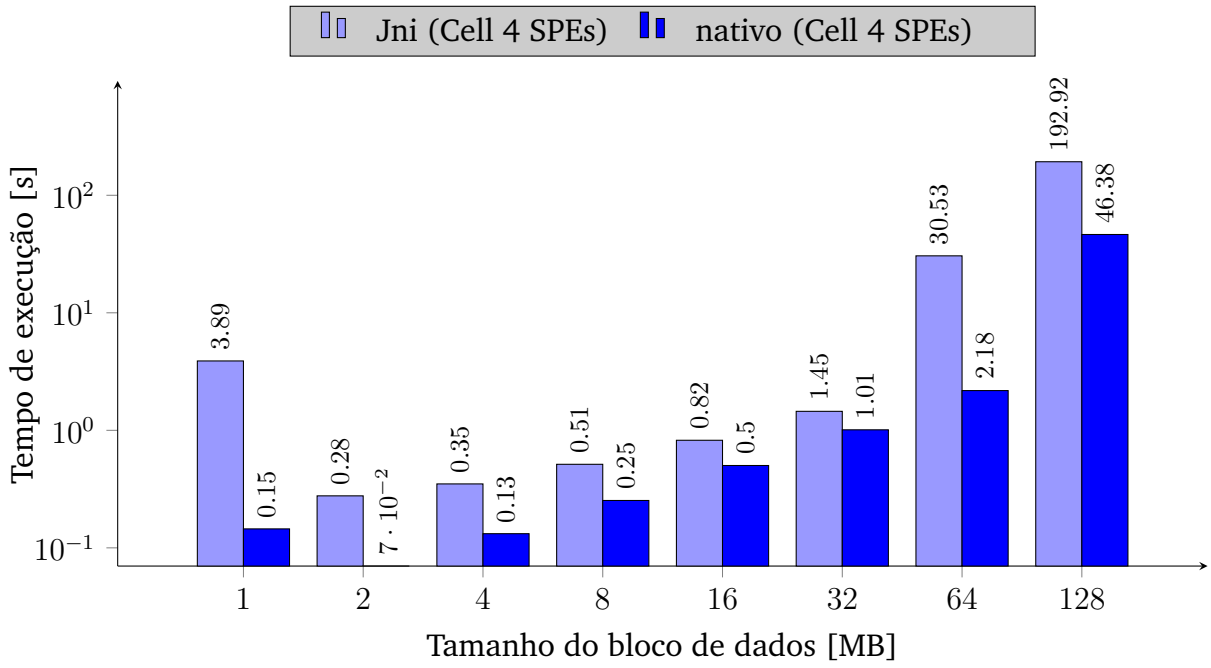

Figura 4.6: Gráfico do tempo de execução das bibliotecas RC5 JNI e nativa. O eixo do tempo mostra o tempo real de execução do processo e esta em escala logarítmica.

```
1  public class matmul {
2          public static void main(String[] args) {
3                  int size = 128;
4                  int numSPEs = 1;
5                  boolean checkResult = false;
6                  boolean useNative = false;
7                  for (int i = 0; i < args.length; i++) {
8                          if (args[i].equals("--check"))
9                                  checkResult = true;
10                         if (args[i].equals("--native"))
11                                 useNative = true;
12                         if (args[i].equals("--java"))
13                                 useNative = false;
14                         if (args[i].equals("--size"))
15                                 size = Integer.parseInt(args[++i]);
16                         if (args[i].equals("--spe"))
17                                 numSPEs = Integer.parseInt(args[++i]);
18                 }
19                 float[][] A = new float[size][size];
20                 float[][] B = new float[size][size];
21                 float[][] R = (useNative) ?
22                         Matrix.multiply(A, B) :
23                         matmul.multiply(A, B);
24                 if (checkResult == true) {
25                         float[][] R2 = (!useNative) ?
26                                 Matrix.multiply(A, B) :
27                                 matmul.multiply(A, B);
28                         for (int i = 0; i < size; i++)
29                                 for (int j = 0; j < size; j++)
30                                         if (R[i][j] != R2[i][j]) {
31                                                 System.out.println("fail!");
32                                                 return;
33                                         }
34                         System.out.println("right!");
35                 }
36         }
37
38         static public float[][] multiply(float[][] A, float[][] B) {
39                 int n = A.length;
40                 float[][] R = new float[n][n];
41                 for (int i = 0; i < n; i++)
42                         for (int j = 0; j < n; j++) {
43                                 R[i][j] = 0;
44                                 for (int k = 0; k < n; k++)
45                                         R[i][j] += A[i][k] * B[k][j];
46                         }
47                 return R;
48         }
49 }
```

Código 4.23: Declaração da classe `matmul`.

	biblioteca JNI					biblioteca nativa				
datasize[MB]	**tuser[s]**	**tsystem[s]**	**telapsed[s]**	**cpu[%]**	**signal**	**tuser[s]**	**tsystem[s]**	**telapsed[s]**	**cpu[%]**	**signal**
1	0.11	0.15	3.89	6.60	0	0.00	0.04	0.15	35.20	0
2	0.13	0.13	0.28	97.30	0	0.00	0.08	0.07	116.50	0
4	0.14	0.21	0.35	102.50	0	0.00	0.18	0.13	136.80	0
8	0.18	0.37	0.51	107.50	0	0.00	0.37	0.25	143.00	0
16	0.23	0.74	0.82	117.90	0	0.00	0.78	0.50	153.40	0
32	0.34	1.50	1.45	126.60	0	0.00	1.62	1.01	159.80	0
64	0.57	5.48	30.53	19.80	0	0.00	3.42	2.18	156.50	0
128	0.72	13.34	192.92	7.00	0	0.00	8.89	46.38	18.70	0
256	0.95	43.41	673.09	6.00	9	0.00	20.17	131.37	14.90	0

Tabela 4.2: Dados da execução da biblioteca RC5. A tabela mostra para os tempos e uso de cpu a média de dez execuções para cada uma das configurações, tanto para a biblioteca JNI quanto para a versão nativa. As colunas **signal** mostram o sinal de termino da aplicação, que foi a mesma para todas as execuções do teste. O sinal 9 na versão nativa para 256MB significa que o programa foi finalizado pelo sistema operacional por falta de memória.

Capítulo 5

Conclusão

A reutilização de aplicações e compatibilidade com o código legado sempre foi uma barreira para o desenvolvimento de novas arquiteturas e paradigmas de computação. Com as tecnologias atuais de fabricação de processadores chegando ao seu limite, é inevitável que arquiteturas novas surjam e que não mais seja possível manter a tradição de compatibilidade com códigos legados e com os mesmos paradigmas de programação.

Para minimizar o impacto e tornar mais transparente a migração para novas arquiteturas, o porte da máquina virtual Java para a arquitetura Cell Broadband Engine tem sido abordado por diversas iniciativas de pesquisas como uma alternativa para esse fim. Todas tentando usar o máximo possível da capacidade de processamento da arquitetura heterogênea do processador Cell através de diferentes abordagens tanto na distribuição de carga quanto no modelo de JVM utilizado. Essa dissertação apresentou uma opção viável de se utilizar a plataforma Cell com desempenho e sem abandonar totalmente os paradigmas e aplicações desenvolvidas até então através da melhoria da máquina virtual Java, e resume os desafios e possibilidades do porte de programas inicialmente estruturados para plataformas homogenias, com modelo de memória compartilhada, amplo suporte do sistema operacional e chamadas de sistemas para uma arquitetura que rompe com a maioria das convenções de *design* de plataformas e de interação com o sistema operacional.

Inicialmente foi proposto buscar a melhora de desempenho da máquina virtual Java sobre a arquitetura Cell através do porte da mesma. O porte do compilador *Just-In-Time* para o SPE foi contemplado com o objetivo de deslocar parte da carga de trabalho da máquina virtual. Dado as características do processador Cell, adaptar implementações já existentes que não preveem porte para arquiteturas com sistema de memória distribuída ou arquiteturas heterogêneas se mostrou pouco viável. A melhor forma de se melhorar o desempenho da JVM neste sentido, seria a escrita completa de uma JVM específica para o processador Cell. Assim sendo, novas abordagens foram buscadas

resultando na abordagem contemplada nessa dissertação.

Este trabalho propôs um modelo de porte e criação de interfaces nativas Cell para programas Javas utilizando JNI com o objetivo de melhorar o desempenho das aplicações Javas. Mostrou-se a através de implementações de bibliotecas selecionadas a validade da abordagem como possível solução para aumento de desempenho, e se consolidou uma metodologia para explorar essa possibilidade em cenários favoráveis.

Dos dois programas foram portados com sucesso como prova de conceito e obteve-se um excelente desempenho. A multiplicação de matrizes obteve um ganho de escalabilidade incomparável com relação ao código Java original em plataforma Cell e x86 e a implementação JNI em plataforma x86 e desempenho comparável a implementação nativa em C. O RC5 executou apenas aproximadamente 0,3 segundos mais lento que o código C original (perda citada em segundos pois se manteve constante independente do tempo levado para as diferentes configurações de execução).

5.1 Trabalhos futuros

Esse trabalho portou com sucesso bibliotecas que realizam trabalhos de computação intensiva e que podem obter grande desempenho na plataforma Cell. Mais ainda é necessário avaliar o porte de bibliotecas com outros perfis e a possibilidade de mover cargas de trabalho, mesmo que não sejam de computação intensiva, para os SPEs da plataforma Cell. O porte de partes da API padrão de Java, especialmente APIs de criptografia e de matemática, podem não somente melhorar o desempenho das aplicações Java mais consolidar a abordagem aqui proposta para cenários mais abrangentes.

Utilizar uma camada de interface provida pela plataforma Java não só permite a melhor utilização da arquitetura Cell como também capacita para o porte para outras arquiteturas seja simplificada. Ainda assim sempre existira a necessidade de se portar a máquina virtual para a nova arquitetura. Sendo as iniciativas de pesquisa até então sobre o porte da JVM baseadas em implementações pré-existentes que não contemplavam o porte de seu código para uma arquitetura de paradigma diferente, o trabalho em se modularizar uma JVM para que seja facilmente portada entre arquiteturas com diferentes características, como modelo de memória, comunicação entre núcleos e assim por diante, é um trabalho de grande importância para que se possa ter um ambiente favorável ao surgimento de novas arquiteturas que buscam o melhor desempenho possível, utilizando as técnicas de construção de processadores disponíveis.

Focando-se na arquitetura Cell, ainda que muitas iniciativas para uma melhor utilização pela JVM das características não convencionais do processador, uma implementação da mesma planejada desde o início para o Cell produzira a melhor integração possível entre as duas plataformas e o melhor desempenho na execução de aplicações

Java tendo como a plataforma Cell como hospedeira da JVM.

Apêndice A

Projetos para a plataforma Cell

A Lista a seguir apresenta uma transcrição da lista de projetos desenvolvidos com foco na plataforma Cell cadastrados no site do centro de desenvolvimento da arquitetura Cell [1]. A lista esta ordenada por ordem alfabética e cada item representa um projeto, composto pelo titulo original, entre parenteses as categorias do do projeto e a descrição original em inglês do projeto que esta disponivel no site.

adicell (API): Finite difference modeling of human head electromagnetics using alternating direction implicit (ADI) method.

AutonomousSim (toolkit, development kit, IDE): A simulation framework for autonomous objects, including particles and boids.

Block Movement Library (library): Project library targets the SPU and is intended to help program SPUs by facilitating the movement of data into and out of the local store of an SPU without having to directly program the DMA hardware.

Brain Derived Vision (programming model): A high-level model of the cortex region of the brain in C.

Cell/B.E. 2D FFT (library, algorithm): A lightweight, 2D FFT library intended to do in-core, 2D FFTs distributed across the SPEs to achieve new heights in high-speed 2D FFTs.

Cell/B.E. IDE (toolkit, development kit, IDE, plugin): Project to develop a set of Eclipse plugins to enable a better experience for developers programming in C/C++.

Cell/B.E. SFMT lib (library, application): SIMD-oriented Fast Mersenne Twister Random generator library: an implementation of the SFMT.

CellCV (development project): A C/C++ computer vision project that performs parallel elaboration of image sequences using the STI Cell processor in order to achieve a speedup.

Cell Messaging Layer (library, API): Extremely fast, MPI-like communication library for clusters of Cell/B.E. processors that allows any SPE to communicate directly with any other SPE.

CellSOM (development project): Implementation of SOM (self-organizing networks): a neural network.

Celluster (application): Easy, adaptable producer/consumer software for distributed computing on a cluster of Cell B.E. processors.

CIV Toolkit (toolkit, development kit, IDE, library, algorithm): A collection of computational intelligence and voice processing algorithms implemented in C.

Cooley-Tukey algorithm (FFT) implementation (algorithm): Most common fast Fourier transform algorithm, it re-expresses the discrete Fourier transform (DFT) of an arbitrary composite size N = N1N2 in terms of smaller DFTs of sizes N1 and N2, recursively, in order to reduce the computation time to O(N log N) for highly-composite N (smooth numbers).

C-ray (application): Volume raycasting software optimized for Cell that achieves interactive volume rendering on very large volumetric data.

CTK (Cell ToolKit) Project (toolkit, development kit, IDE, library): A C/C++ toolkit library for multicore programming.

czip (application): A UNIX-based utility that allows users to compress files in gzip format.

Data flow processing parallelization (library): Library for parallelizing flow data computations.

debiancell (toolkit, development kit, IDE): Cell simulator, part of Debian distribution.

Exact CT reconstruction algorithm (algorithm): Implemented for Cell.

Facedetect (application, algorithm): Fast Haar-like object (the difference of the sum of pixels of areas inside the rectangle) detection implementation that reimplements stump-based algorithm of OpenCV library.

Fis data movement compiler (toolkit, development kit, IDE, application): Dataflow-based compiler for the purpose of simplifying data transfers to and from SPU processor elements.

Genetic algorithms (library, documentation, algorithm, development project): Project to create an open source genetic algorithms library for Cell. It possesses a well-documented implementation of GAs and a number of sample fitness functions illustrating how to use the GA library.

Genetic algorithm for JSSP (development project, application, algorithm): Project to create a Cell implementation of an application to solve the job shop schedule problem using genetic algorithm.

Georgia Tech Cell/B.E. libraries (library, algorithm): Optimized libraries from Georgia Tech, for example for FFT, MPEG, compression, and encryption.

GLIMPSES (toolkit, development kit, IDE, application): A profiling tool for understanding program memory behavior and evaluating program regions for execution on the SPEs.

Grid Space (application): A novel grid system that is Python-based. It extends Namespace into GridSpace so any objects are accessible throughout the grid; the codes are distributed executed and can be JIT-compiled into SPE instructions automatically.

H.264 on Cell (development project, algorithm): Project focuses on integrating the H.264 standard for video compression also known as MPEG-4 Part 10 or MPEG-4 AVC.

HD Cell Raytracer (HDCellRT) (application): An interactive ray tracer for use on the Sony PlayStation 3, it can output to the PS3's high definition frame buffer. Most of the project is written in assembly.

IDELCell (library, development project): Images Decoding/Encoding Libraries project planned to support all kinds of images decoding/encoding libraries.

image-pcnn-cell (algorithm): Implementation of Pulse-Coupled Neural Networks (PCNN) image-processing algorithm.

Image processing routines (application): A set of image processing functions.

janus (algorithm): CFD solver that will implement streaming and data parallel programming models to leverage throughput gains currently exhibited by GPU or Cell processors.

K-means algorithm (algorithm): Used to cluster n objects based on attributes into k partitions, k < n.

LGB vector quantization (VQ) algorithm (algorithm): A project that implements the parallelized version of the LBG vector quantization algorithm on Cell.

Library for genetic algorithms (library, algorithm): Library.

libspe (library): Cell/B.E. implementation of the SPE Runtime Management Library.

LibSPEPort (toolkit, development kit, IDE, library): Tools library to make it easier to port existing applications (and write new ones) to the SPEs; includes PPE- and SPE-side functions and conventions.

Linux 2.6.21.5 kernel extension (development project): Project to increase Cell efficiency on this kernel.

MapReduce (programming model, application): A simple and flexible parallel programming model initially proposed by Google for large scale data processing in a distributed computing environment.

MLEM for small animal PET (algorithm): Algorithm for reconstructing small animal PET scans.

MPI on Cell (API): Implementation of MPI_Allgather (gathers data from all tasks and distributes it to all) and MPI_Alltoall (sends data from all to all processes).

MR Cell (programming model, application): Another MapReduce.

Multicore SWARM (library, toolkit, development kit, IDE): An open source library for developing efficient and portable implementations that make use of multicore processors.

Multiview 3D Reconstruction (development project, toolkit, development kit, IDE): Project to make a real multi-view 3D reconstruction system.

Neural gas algorithm (algorithm): Competitive learning/machine vision algorithm.

nOStradamus (development project): Project to create an AI-based OS; being revived on PS3.

OGR implementation (algorithm): Implementation of Golomb ruler having length of 10 marks on a Cell Broadband Engine simulator. (It is marks at integer positions along an imaginary ruler such that no two pairs of marks are the same distance apart – a practical use is in the design of phased array radio antennas, such as radio telescopes.)

OpenCV (library, development project): Performance improvements to OpenCV for the PS3, an open source computer vision library.

OpenMP for Cell (API, library): API that supports multi-platform shared-memory parallel programming in C/C++ and FORTRAN on all architectures.

parallel for (programming model): A data parallel scientific programming model that compiles efficiently to different platforms like distributed memory (MPI), shared memory multi-processor (pthreads), Cell processor (SPElib), SIMD vectorization (SSE, Altivec), and sequential C++ code.

Parallel implementation of Watershed4d (algorithm): A four-dimensional image processing segmentation algorithm that splits an image into areas, based on the topology of the image.

Parallel Radix Sort (algorithm): A sorting algorithm that sorts integers by processing individual digits.

Particle Swarm Optimization (PSO) (algorithm): A swarm-intelligence-based algorithm to find a solution to an optimization problem in a search space or model and predict social behavior in the presence of objectives. Swarm intelligence is an AI based on the collective behavior of decentralized, self-organized systems.

PCNN (algorithm): Cell version of the Pulse-Coupled Neural Network, a neural network algorithm that is the basis for many visual cortex models.

PlayStation 3 Multimedia Center (application): Allows PlayStation 3 to decode MPEG2 and H.264 video streams and become an HDTV receiver.

Poisson solver (algorithm): Numerical solution of Poisson equation for 2D and cylindrical coordinates.

Prime Number Generator (application): Code produces prime numbers.

PS3 battery model (programming model): Project enables the use of battery geometry information to help determine the battery state of health in an embedded application on a PS3.

qd4cell (application): Quantum dynamic program for matrix computation.

Random Number Generator (application): Exploits SPU cores to get a better performance out of the Scalable Parallel Random Number Generators Library.

Ray tracing (application): A Whitted-style (more realistic) ray tracer in C.

SPE Execution Management System (library): Simple library for creating SPE accelerators.

SPE TaskLib framework (library, toolkit, development kit, IDE): Low-latency high-speed hybrid framework-library for utilizing SPE units on the PS3, QS20 to carry out computational tasks.

swcell (toolkit, development kit, IDE): Platform-independent Cell-processor emulation software.

Texture Synthesis (algorithm): An algorithm for multi-resolution texture synthesis.

Time Domain Correlator (application): An optimized time domain correlator.

tp4cell (application): A port of the GDB Tracepoints extension.

Using Cell (development project): A new way to compute differences between images.

W4A-SW (algorithm): Implementation of the Smith-Waterman sequence alignment algorithm for performing local sequence alignment.

WLAN MAC simulator (application, toolkit, development kit, IDE): An event simulator for simulating MAC layer of IEEE 802.11 WLAN written in C, primarily developed as a benchmark application.

Referências Bibliográficas

[1] Cell Broadband Engine Resource Center, may 2010.

[2] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh, and J. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 280–290, 1998.

[3] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray Tracing on the CELL Processor. *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 15–23, 2006.

[4] N. Breese. Overview of SPU-Optimized Cryptography "CrackStation". In *Back Hat Europe 2008*. security-assessment.com, Black Hat, feb 2008.

[5] C. Cher and M. Gschwind. Cell GC: using the cell synergistic processor as a garbage collection coprocessor. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/-SIGOPS international conference on Virtual execution environments*, pages 141–150, New York, NY, USA, mar 2008. ACM.

[6] M. Cierniak and W. Li. Just-in-time optimizations for high-performance Java programs. *Concurrency Practice and Experience*, 9(11):1063–1073, 1997.

[7] J. Greene and R. Cooper. A Parallel 64K Complex FFT Algorithm for the IBM/-Sony/Toshiba Cell Broadband Engine Processor. *Technical Conference Proceedings of the Global Signal Processing Expo (GSPx)*, 2005.

[8] M. Gschwind. Chip multiprocessing and the cell broadband engine. *Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, 2006.

[9] D. Hackenberg. Fast Matrix Multiplication on Cell (SMP) Systems, may 2010. `http://www.tu-dresden.de/zih/cell/matmul`.

[10] P. Hofstee. Introduction to the Cell Broadband Engine. Technical report, IBM Corp., 2005.

[11] F. Hoyos. Um Modelo de Execução para Java no Processador Cell BE. Dissertação de mestrado, Universidade Estadual de Campinas (Unicamp), Instituto de Computação (IC), Avenida Albert Einstein, 1251, Caixa Postal 6176, 13084-971, Campinas, São Paulo, Brasil, nov 2009.

[12] IBM Corp. *Cell Broadband Engine Architecture, version 1.02*, oct 2007.

[13] IBM Corp. *Cell Programming Workshop*, 2007.

[14] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. *Proceedings of the ACM 1999 conference on Java Grande*, pages 119–128, 1999.

[15] J. Kahle et al. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, 2005.

[16] M. Kistler, M. Perrone, and F. Petrini. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, 26(3):10–23, 2006.

[17] S. Lee, B. Yang, S. Moon, et al. Efficient Java exception handling in just-in-time compilation. *Software Practice and Experience*, 34(15):1463–1480, 2004.

[18] S. Liang. *The Java Native Interface Programmer's Guide and Specification*. Addison-Wesley, jun 1999.

[19] R. Mcllroy and J. Sventek. Hera-JVM: abstracting processor heterogeneity behind a virtual machine. *Proceedings of the 12th Workshop on Hot Topics in Operationg Systems*, may 2009.

[20] A. Noll, A. Gal, and M. Franz. CellVM: A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor. *Proceedings of the Workshop on Cell Systems and Applications in conjunction with The 35th international Symposium on Computer Architecture*, jun 2008.

[21] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java. *Proceedings of ECOOP*, 1850:362–387, 2000.

[22] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109, 2003.

[23] G. Sorft. Java on Cell B.E. Diploma thesis, Fachhochschule Aachen, Fachbereich Elektrotechnik, Eupenerst. 70, 52066 Aachen, sep 2007.

[24] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[25] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 180–195, 2001.

[26] J. Taylor. Nick Breese's "PS3 MD5 benchmark" code improvement, sep 2008. `http://www.ibm.com/developerworks/forums/thread.jspa?threadID=226415`.

[27] B. Venners. *Inside the Java 2 Virtual Machine*. McGraw-Hill, second edition, 1999.

[28] J. Whaley. Joeq: A virtual machine and compiler infrastructure. *Science of Computer Programming*, 57(3):339–356, 2005.