

Seminário Java + GPU

Paulina Rehbein

Programação Paralela Heterogênea

Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS

Porto Alegre – RS – Brasil

Novembro - 2025

Tópicos

- JCuda
- matrix-multiplication
- Comparação desempenho

JCuda

- O JCuda é um wrapper JNI (Java Native Interface) que expõe praticamente as mesmas funções da API CUDA, mas acessíveis via Java.
- Faz o espelho das abstrações do CUDA através de objetos e métodos Java que chamam código nativo C/C++ por baixo.
- Ele serve como a plataforma comum para diversas outras bibliotecas de aceleração GPU baseadas em CUDA, como JCublas, JCufft, JCurand, e JCusparse.

JCuda - Runtime API

- A principal aplicação dos bindings do JCuda Runtime é a interação com bibliotecas existentes construídas sobre a CUDA Runtime API
- É usada para interagir com bibliotecas como JCublas, JCufft, JCurand, e JCusparse.

JCuda - Driver API

- O uso principal dos bindings do JCuda Driver é carregar módulos PTX e CUBIN e executar kernels a partir de uma aplicação Java
- O Driver API permite inicializar o driver, criar contextos, carregar arquivos PTX/CUBIN, obter um handle para a função do kernel (cuModuleGetFunction) e finalmente lançar o kernel (cuLaunchKernel).
- Para garantir que o kernel possa ser acessado pelo seu nome na API de Driver, a função do kernel deve ser declarada como extern "C" no código CUDA

JCuda - Pointer

- Referências Java não são adequadas para emular ponteiros nativos, pois não permitem aritmética de ponteiros e não podem ser passadas diretamente para bibliotecas nativas.
- Por não existir ponteiros reais em Java o JCuda introduziu a classe Pointer que pode ser tratada de forma semelhante a um ponteiro void* em C
- Um Pointer pode ser criado instanciando um novo Pointer (que inicialmente será NULL).

```
Pointer pointer = Pointer.to(new int[]{N});
```

JCuda – Abstrações

CUDA C/C++	JCuda	Descrição
<code>cudaSetDevice(int device)</code>	<code>JCuda.cudaSetDevice(int)</code>	Define qual GPU será utilizada pelo programa.
<code>cudaGetDeviceProperties(cudaDeviceProp *prop, int device)</code>	<code>JCuda.cudaGetDeviceProperties(cudaDeviceProp, int)</code>	Obtém as propriedades (nome, memória, SMs etc.) da GPU.
<code>cuInit(0)</code>	<code>JCudaDriver.cuInit(0)</code>	Inicializa a API Driver e prepara o uso do CUDA.
<code>cuDeviceGet(CUdevice device, int ordinal)</code>	<code>JCudaDriver.cuDeviceGet(CUdevice, int)</code>	Obtém uma referência a uma GPU específica.
<code>cuCtxCreate(CUcontext ctx, unsigned int flags, CUdevice device)</code>	<code>JCudaDriver.cuCtxCreate(CUcontext, int, CUdevice)</code>	Cria um contexto CUDA associado a um dispositivo (essencial na Driver API).
<code>CUdeviceptr</code>	<code>Pointer (classe jcuda.Pointer)</code>	Representa um ponteiro para memória no dispositivo (GPU).

JCuda – Abstrações

CUDA C/C++	JCuda	Descrição
<code>cudaLaunchKernel() / kernel<<<grid, block>>>(...)</code>	<code>JCudaDriver.cuLaunchKernel(function, gridX, gridY, gridZ, blockX, blockY, blockZ, sharedMem, stream, params, null)</code>	Lança um kernel na GPU.
<code>cudaMalloc(void **devPtr, size_t size)</code>	<code>JCuda.cudaMalloc(Pointer, long)</code>	Aloca memória na GPU (Runtime API).
<code>cudaFree(void *devPtr)</code>	<code>JCuda.cudaFree(Pointer)</code>	Libera memória alocada na GPU (Runtime API).
<code>cudaMemcpy(void *dst, void *src, size_t size, cudaMemcpyKind kind)</code>	<code>JCuda.cudaMemcpy(Pointer dst, Pointer src, long size, int kind)</code>	Copia dados entre memória host e device (síncrono).
<code>cuMemcpyHtoD(CUdeviceptr dst, Pointer src, long size)</code>	<code>JCudaDriver.cuMemcpyHtoD(CUdeviceptr, Pointer, long)</code>	Copia do host → device.
<code>cuMemcpyDtoH(Pointer dst, CUdeviceptr src, long size)</code>	<code>JCudaDriver.cuMemcpyDtoH(Pointer, CUdeviceptr, long)</code>	Copia do device → host.
<code>__global__ void kernel(...)</code>	Kernel definido em <code>.cu</code> e compilado para <code>.ptx</code> , carregado via <code>cuModuleGetFunction()</code>	Função que será executada na GPU.

JCuda – matrix multiplication

- <https://github.com/PaulinaEster/jcuda-samples/tree/master/seminario-gpu>

JCuda – matrix multiplication Passo 1

// passo 1 - principais variáveis

```
public class JCudaMatrixMultiplication {  
    public static void main(String args[]) throws IOException  
{  
    Config.setupCommon();  
    int N = Config.getWorkload().getN();  
    int[][] matrix1 = new int[N][N];  
    int[][] matrix2 = new int[N][N];  
    int[][] matrix3 = new int[N][N];  
    MatrixMultiplicationUtils.initialization(matrix1, matrix2, matrix3);  
    int[] matrix1Linear = new int[N*N];  
    int[] matrix2Linear = new int[N*N];  
    int[] matrix3Linear = new int[N*N];  
    linearization(matrix1, matrix2, matrix3, matrix1Linear, matrix2Linear, matrix3Linear);  
    ....
```

JCuda – matrix multiplication

```
// passo 1 - principais variáveis  
// passo 2 - cria PTX e inicializa JCuda Driver  
// Enable exceptions and omit all subsequent error checks  
JCudaDriver.setExceptionsEnabled(true);  
// Create the PTX file by calling the NVCC  
String ptxFileName = JCudaSamplesUtils.preparePtxFile(  
"src/main/resources/kernels/JCudaMatrixMultiplicationKernel.cu");  
// Initialize the driver  
culInit(0);  
CUdevice device = new CUdevice();  
cuDeviceGet(device, 0);
```

JCuda – matrix multiplication

```
// passo 3 - Cria contexto para o device
CUcontext context = new CUcontext();
cuCtxCreate(context, 0, device);
int[] maxThreadsPerBlock = new int[1];
JCudaDriver.cuDeviceGetName(nameBytes,
    nameBytes.length,
    device);

JCudaDriver.cuDeviceGetAttribute(
    maxThreadsPerBlock,
    CUdevice_attribute.CU_DEVICE_ATTRIBUTE_MAX_
    THREADS_PER_BLOCK,
    device);
```

```
// Load the ptx file.
CUmodule module = new CUmodule();
cuModuleLoad(module, ptxFileName);
// Obtain a function pointer to the "matrix_multiplication"
// function.
CUfunction function = new CUfunction();
cuModuleGetFunction(function, module,
    "matrix_multiplication");
```

JCuda – matrix multiplication

// passo 4 - Criar objetos na memória

```
CUdeviceptr deviceMatrix1 = new CUdeviceptr();  
CUdeviceptr deviceMatrix2 = new CUdeviceptr();  
CUdeviceptr deviceMatrix3 = new CUdeviceptr();
```

// Alocação de dados para o device

```
cuMemAlloc(deviceMatrix1, N * N * Sizeof.INT);  
cuMemAlloc(deviceMatrix2, N * N * Sizeof.INT);  
cuMemAlloc(deviceMatrix3, N * N * Sizeof.INT);
```

// Tranferencia de dados para a memoria do device

```
cuMemcpyHtoD(deviceMatrix1,  
    Pointer.to(matrix1Linear),  
    N * N * Sizeof.INT);  
  
cuMemcpyHtoD(deviceMatrix2,  
    Pointer.to(matrix2Linear),  
    N * N * Sizeof.INT);  
  
cuMemcpyHtoD(deviceMatrix3,  
    Pointer.to(matrix3Linear),  
    N * N * Sizeof.INT);
```

JCuda – matrix multiplication

```
// passo 5 - Cria ponteiro para os parametros do kernel  
// Set up the kernel parameters: A pointer to an array  
// of pointers which point to the actual values.  
  
Pointer kernelParameters = Pointer.to(  
    Pointer.to(deviceMatrix1),  
    Pointer.to(deviceMatrix2),  
    Pointer.to(deviceMatrix3),  
    Pointer.to(new int[]{N})  
);
```

JCuda – matrix multiplication

```
// passo 1 - principais variáveis  
  
// passo 2 - cria PTX e inicializa JCuda Driver  
  
// passo 3 - Cria contexto Cuda  
  
// passo 4 - criar objetos na memória  
  
// passo 5 - Cria ponteiro para os parametros do kernel  
  
// passo 6 – executa o kernel  
  
// Call the kernel function.  
  
int threads = maxThreadsPerBlock[0];  
  
int blockSize = (int)Math.ceil((double)(N * N) / threads);
```

```
cuLaunchKernel(function,  
blockSize, 1, 1, // Grid dimension  
threads, 1, 1, // Block dimension  
0, null, // Shared memory size and stream  
kernelParameters, null // Kernel- and extra  
parameters  
);  
  
cuCtxSynchronize();
```

JCuda – matrix multiplication

```

// passo 1 - principais variáveis
// passo 2 - cria PTX e inicializa JCuda Driver
// passo 3 - Cria contexto Cuda
// passo 4 - criar objetos na memória
// passo 5 - Cria ponteiro para os parametros do kernel
// passo 6 – executa o kernel
// passo 7 - copiar os resultados para o host

cuMemcpyDtoH(
    Pointer.to(matrix3Linear),
    deviceMatrix3,
    N * N * Sizeof.FLOAT
);
  
```

JCudaMatrixMultiplicationKernel.cu

```

extern "C"

__global__ void matrix_multiplication(int* matrix1,
int* matrix2,
int* matrix3,
int N
){
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if(tid > (N * N)) {return;}
    int j = (tid / N) % N; // altura largura
    int k = tid % N;
    for(int i = 0; i < N; i++){
        matrix3[j + k * N] += matrix1[j + i * N] * matrix2[i + k * N];
    }
}
  
```

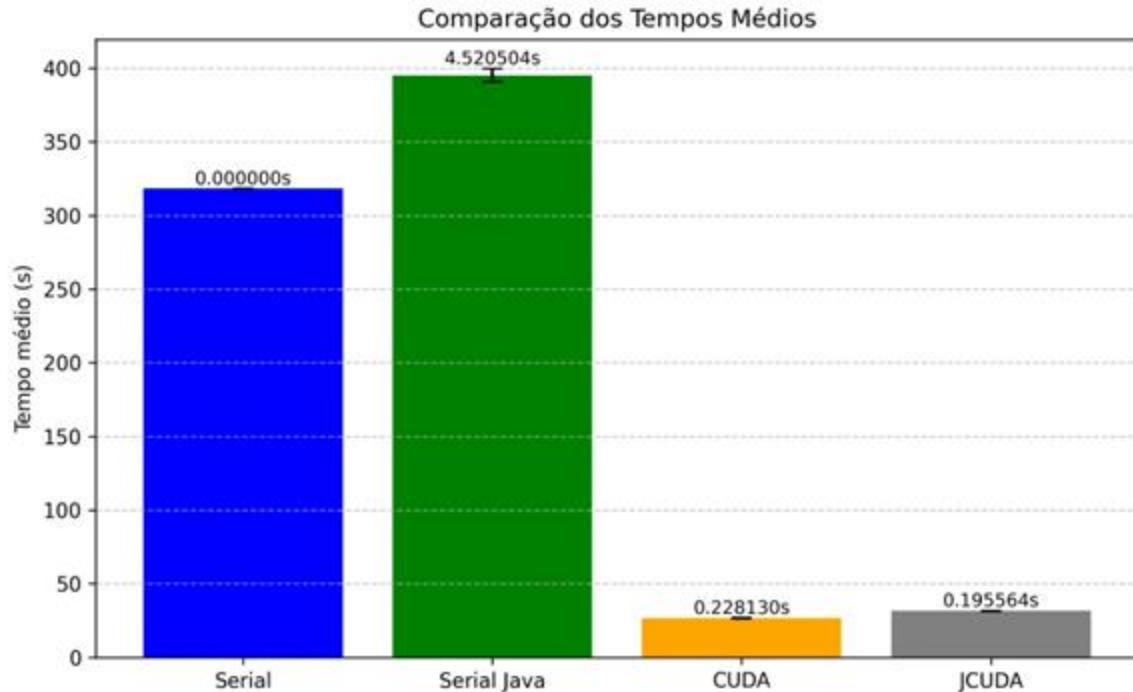
Comparação JCuda x CUDA

- SO: Debian 6.1.148-1
- Kernel Linux: Linux slurm-head 6.1.0-39-amd64
- CPU: Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz
- GPU: NVIDIA TITAN Xp
- Versões de Software:
 - Compilador: GCC 11.3 G++ 11.3
 - CUDA: 11.8
 - JDK: 8
 - Maven: 3.9

Executado multiplicação de matrizes de dimensão 4096 x 4096 Workload G

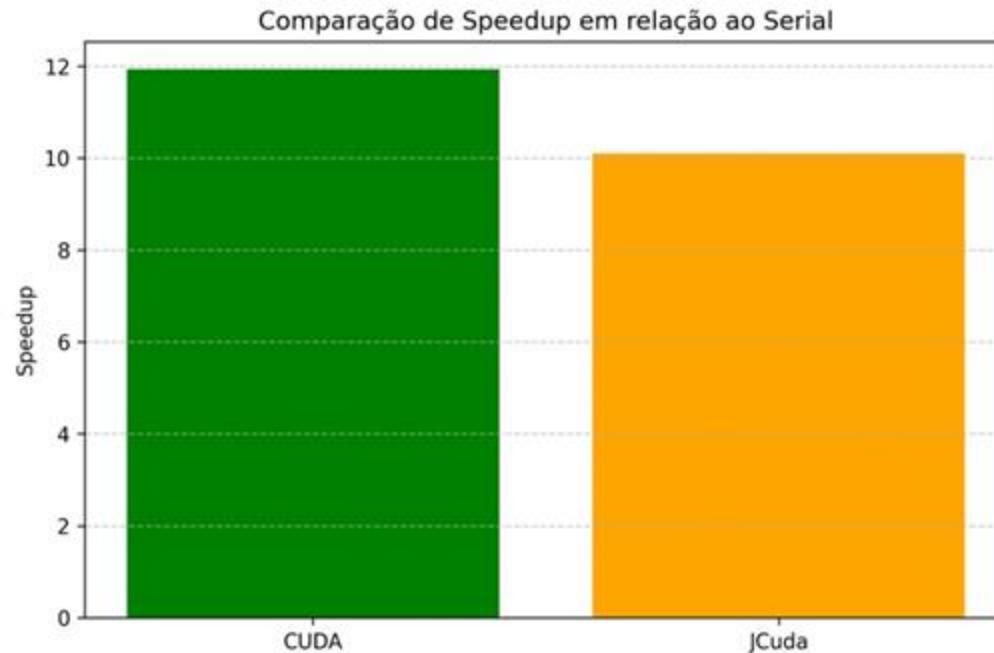
Comparação JCuda x CUDA

- Tempo médio serial em C = 318.261150s
- Tempo médio serial JAVA = 395.117114s
- Tempo médio CUDA = 26.666288s
- Tempo médio JCuda = 31.517022s



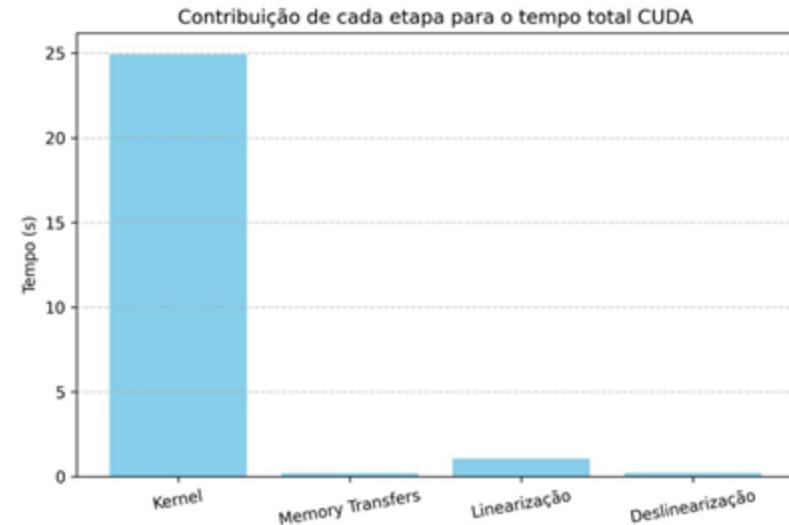
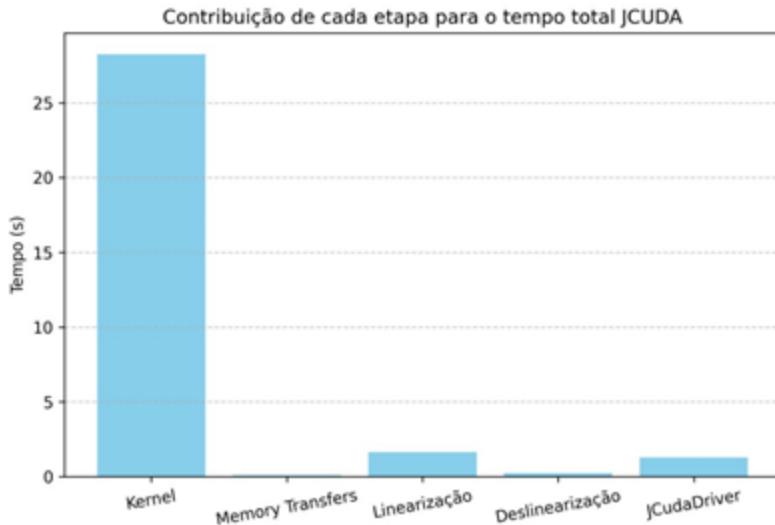
Comparação JCuda x CUDA

- Speedup
- Speedup CUDA
11.93x
- Speedup JCuda
10.10x



Comparação JCuda x CUDA

- Tempo por etapa da execução



Referências

- <http://www.jcuda.org/jcuda/JCuda.html#Features>
- <https://github.com/jcuda/jcuda-samples>
- <https://mvnrepository.com/artifact/org.jcuda/jcuda>