

Projeto Babylon: rodando **Java** na **GPU**

Thiago Gonzaga - Java Noroeste

O que vem pela frente

- | | | |
|----|----------------------------------|---|
| 01 | O problema | como o Java chega nos aceleradores hoje |
| 02 | Project Babylon | code reflection e o code model |
| 03 | HAT | do Java ao CUDA C em tempo de execução |
| 04 | Demo ao vivo | fractal de Julia, 1.440.000 threads por frame |
| 05 | Limites e próximos passos | o que já dá para fazer e o que ainda não |

Quem sou eu

- Bacharel em Ciência da Computação pelo IBILCE/UNESP
- 20 anos de experiência em TI
- Oracle ACE Pro
- AWS Community Builder
- Coordenador do Java Noroeste
- Ajudo nas comunidades técnicas de Rio Preto

Como o Java chega na GPU hoje

- **JNI + CUDA C:** duas linguagens, dois builds e dois times.
- **Bindings como JCuda e JOCL:** o kernel continua sendo escrito em C.
- **Aparapi e TornadoVM:** funcionam bem, mas ficam fora do OpenJDK.
- As cargas de ML rodam em acelerador, e a maior parte do ecossistema Java não alcança esse hardware.

O Babylon parte de outra ideia: dar ao JDK acesso ao seu próprio código, para que ele mesmo possa traduzi-lo.

Como era, e como fica

CUDA C + JNI

```
__global__ void julia(int* out, int w, int h,
                    float cr, float ci) {
    int i = blockIdx.x * blockDim.x
        + threadIdx.x;
    if (i < w * h) {
        ...
    }
}
```

- Um arquivo .cu, compilado no build.
- Uma ponte JNI para chamar do Java.

Java com @Reflect

```
@Reflect
public static void juliaKernel(S32Array out,
                              int w, int h, float cr, float ci) {
    if (GIX() < w * h) {
        ...
    }
}
```

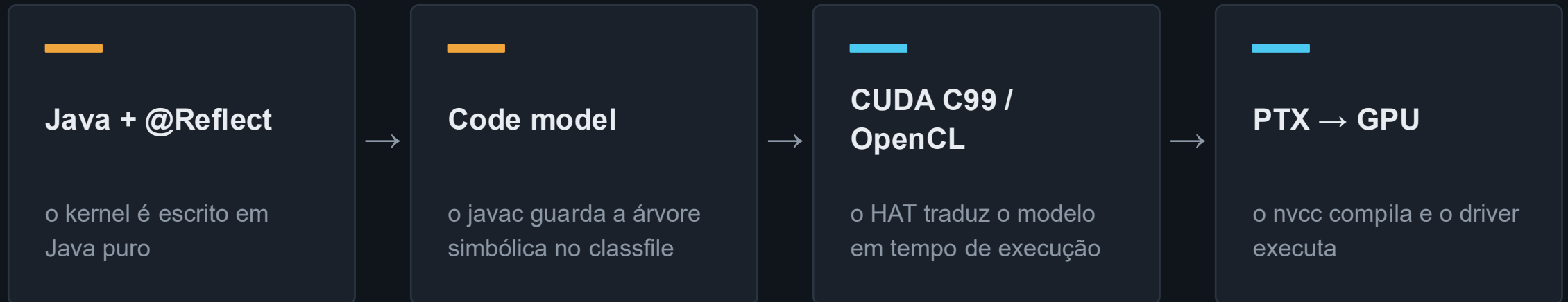
- Um arquivo .java, compilado pelo javac.
- O HAT gera e compila o CUDA em tempo de execução.

Code reflection: acesso ao corpo do método

- Projeto do OpenJDK, no branch `code-reflection`, liderado por Paul Sandoz.
- A reflection clássica descreve classes e métodos. A code reflection descreve o corpo do método.
- Métodos anotados com `@Reflect` guardam um **code model** no classfile: uma árvore simbólica em forma de SSA.
- A API é incubada: `jdk.incubator.code`.

Com o modelo em mãos, dá para analisar, transformar e gerar código para outro alvo: C, PTX ou o que for.

Do **bytecode** até a **GPU**



Tudo acontece em tempo de execução, dentro do processo Java. Âmbar: mundo JVM. Ciano: mundo GPU.

HAT: Heterogeneous Accelerator Toolkit

- Toolkit construído sobre a code reflection do Babylon.
- **Modelo SIMT:** você descreve o trabalho de uma thread.
- Dados off-heap via FFM/Panama, sem cópia para o heap.
- O NDRange 1D, 2D ou 3D define quantas threads a GPU lança.
- Backends plugáveis: `ffi-cuda`, `ffi-openssl`, `java-mt`, `java-seq`.
- O mesmo bytecode roda em qualquer um deles.
- Fica no repositório do Babylon, em `babylon/hat`.

Como a GPU executa esse código

Grade

todas as threads de um lançamento.
Aqui, 1.440.000: uma por pixel

Bloco

grupo de threads que roda num
mesmo SM e compartilha memória
local

Warp

32 threads que executam a mesma
instrução no mesmo ciclo

- Cada thread descobre quem é pela posição: `blockIdx.x * blockDim.x + threadIdx.x`. No HAT, isso é `GIX()`.
- Se as threads de um warp tomam caminhos diferentes num if, o hardware executa os dois caminhos em sequência.
- Uma thread por pixel funciona bem porque cada pixel é independente: nenhuma precisa olhar o vizinho.

As classes que aparecem no código

`@Reflect`

guarda o code model do método

`Accelerator`

ponto de entrada; escolhe o backend

`Backend`

quem executa: ffi-cuda, java-mt, ...

`S32Array`

buffer nativo compartilhado com a GPU

`ComputeContext`

de onde o kernel é despachado

`NDRange`

quantas threads lançar: 1D, 2D, 3D

Dentro do kernel, `GIX()` vem de `hat.KernelContext` e devolve o id global da thread.

O kernel do fractal, escrito em Java

```
@Reflect
public static void juliaKernel(S32Array out, int w, int h, float cr, float ci) {
    if (GIX() < w * h) { // 1 thread por pixel
        int x = GIX() % w;
        int y = GIX() / w;
        float zr = (x - w * 0.5f) * (3.0f / h);
        float zi = (y - h * 0.5f) * (3.0f / h);
        int iter = 0;
        while (zr * zr + zi * zi < 4.0f && iter < MAX_ITER) {
            float t = zr * zr - zi * zi + cr;
            zi = 2.0f * zr * zi + ci;
            zr = t;
            iter = iter + 1;
        }
        ... // cor ARGB a partir de iter
        out.array(GIX(), color);
    }
}
```

O despacho a partir da JVM

```
@Reflect
public static void julia(ComputeContext cc, S32Array out, int w, int h, ...) {
    cc.dispatchKernel(NDRange.of1D(w * h), () -> juliaKernel(out, w, h, cr, ci));
}

var accelerator = new Accelerator(MethodHandles.lookup(), Backend.FIRST);
var out = S32Array.create(accelerator, W * H); // 1.440.000 inteiros ARGB
accelerator.compute(cc -> julia(cc, out, W, H, cr, ci)); // um frame
```

- `S32Array` é uma interface mapeada para memória nativa: o mesmo buffer que a GPU escreve e o `BufferedImage` lê.
- `NDRange.of1D(w * h)` lança uma thread por pixel; o HAT cuida da transferência e da sincronização.
- Para rodar na CPU, basta trocar o jar do backend no classpath.

O CUDA C gerado pelo HAT

```
HAT_KERNEL void juliaKernel(HAT_GLOBAL_MEM S32Array_t* out,
                           int w, int h, float cr, float ci) {
    if (HAT_GIX < w * h) {
        int x = HAT_GIX % w;
        int y = HAT_GIX / w;
        float zr = (x - w * 0.5f) * (3.0f / h);
        ...
        out->array[(long)HAT_GIX] = color;
    }
}

// #define HAT_GIX (blockIdx.x * blockDim.x + threadIdx.x)
```

O HAT escreve o arquivo em `var/cuda/tmp_*.cu`, compila para PTX com o `nvcc` e despacha, com a JVM em execução.

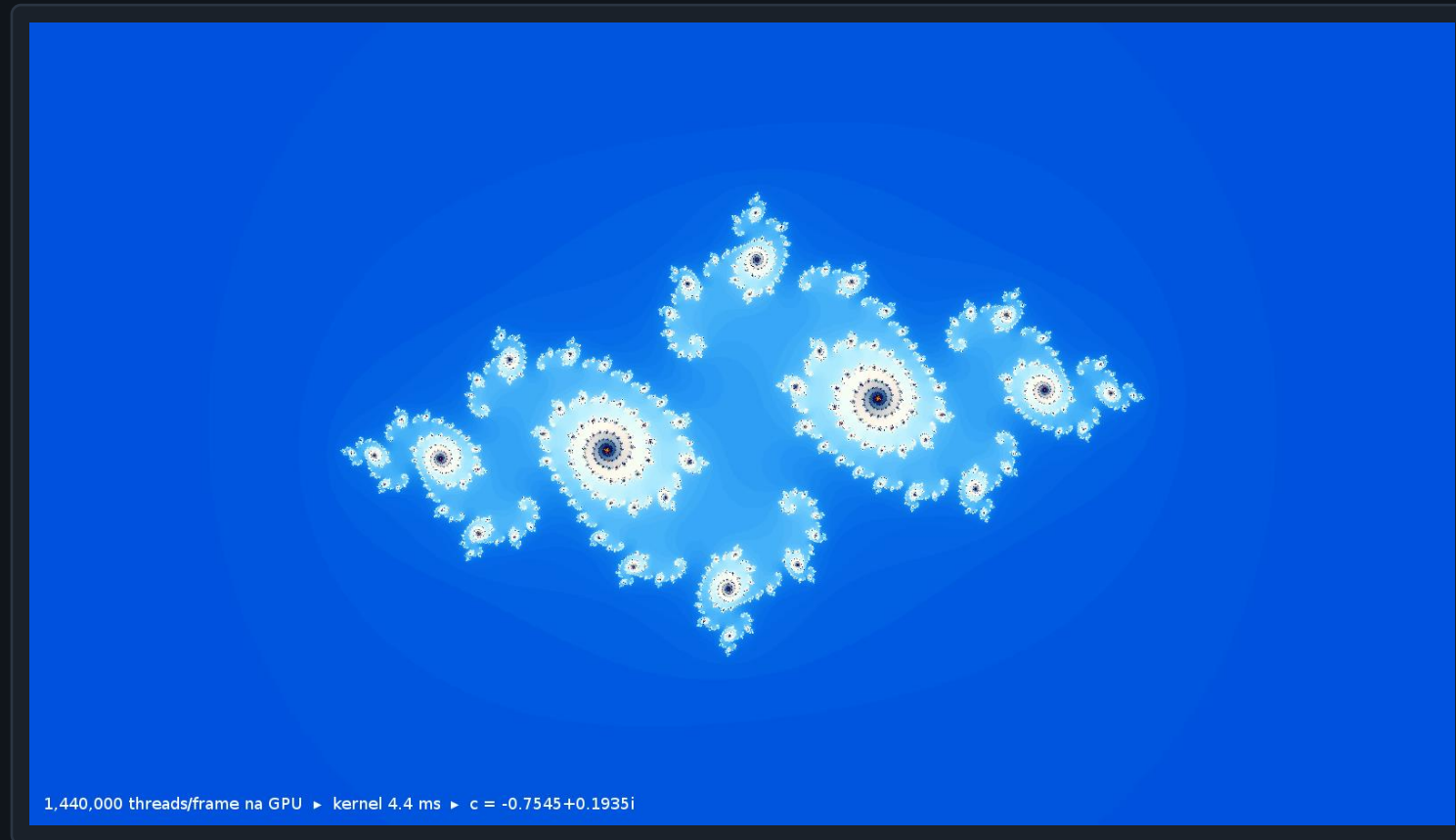
Requisitos para rodar

- **JDK Babylon** compilado do branch `code-reflection`. Ainda não há binário publicado.
- **Boot JDK 26+** para compilar o Babylon, além de make, autoconf, cmake, gcc 12+ e os headers do X11.
- **Maven** para compilar o HAT (`mvn clean package`).
- **Linux ou WSL2, GPU NVIDIA, CUDA Toolkit 13+**. O HAT chama o `nvcc` em tempo de execução; o driver sozinho não basta.
- Sem GPU, o backend `java-mt` roda o mesmo código na CPU, bem mais devagar.

Flags da JVM. A API é incubada, então nada roda sem elas:

```
java --enable-preview --add-modules=jdk.incubator.code \  
    --enable-native-access=ALL-UNNAMED -cp ... JavaDayGpu.java
```

demo



Fractal de Julia animado · 1.440.000 threads por frame
O nvidia-smi fica aberto no segundo terminal



`github.com/devops-thiago/javaday-babylon-gpu`

Um arquivo Java, o run.sh e o passo a passo completo no README

Alguns numeros

matmul 1024×1024 · 100 iterações · mesmo bytecode

105 ms

CPU · java-mt · 32 threads

2,6 ms

GPU · ffi-cuda · RTX 5060 Ti

~40×

de diferença entre os dois
backends

A primeira iteração na GPU leva cerca de 460 ms: é o custo de traduzir o code model, compilar com o nvcc e carregar o módulo. As seguintes são só execução.

O que cabe num kernel HAT

- Um subconjunto do Java: primitivos, buffers do HAT, controle de fluxo e chamadas a outros métodos `@Reflect`.
- Memória local (`__shared__`), barreiras e kernels 1D, 2D e 3D.
- Não cabem objetos arbitrários, GC nem exceções dentro do kernel.
- O projeto está em incubação: a API muda e ainda é pesquisa, não produção.

É o mesmo recorte que outras linguagens de kernel fazem, só que dentro do Java.

ML, IA e Tensor Cores

- O code model serve como alvo de tradução para destinos além de CUDA e OpenCL, incluindo as instruções de Tensor Core (MMA e WMMA).
- O repositório já traz `flashattention`, `tensors`, `blacksholes` e `nbody`.
- A mesma base abre caminho para autodiferenciação, DSLs no estilo Triton e interoperação com stacks de ML.

É pesquisa em andamento no OpenJDK, sem data para virar recurso final.

Como rodar na sua máquina

```
git clone --depth 1 -b code-reflection https://github.com/openjdk/babylon
bash configure --with-boot-jdk=$JAVA_HOME && make images      # JDK Babylon
cd hat && mvn clean package                                     # HAT + backends

git clone https://github.com/devops-thiago/javaday-babylon-gpu
BABYLON_HOME=/caminho/do/babylon ./run.sh                    # abre a janela
```

- Requisitos: Linux ou WSL2, GPU NVIDIA, CUDA 13+ e um JDK 26+ para compilar o Babylon.
- O README traz os ajustes de WSL2 e o workaround do `rsqrt` com glibc 2.41+.

Três ideias para levar

01

O JDK enxerga o próprio código

a code reflection guarda o corpo do método como um code model

02

O HAT traduz esse modelo

de code model para CUDA C ou OpenCL, em tempo de execução

03

O kernel continua sendo Java

o mesmo código roda na GPU e na CPU; muda só o backend

O projeto está em incubação: já dá para experimentar, ainda não dá para colocar em produção.

Obrigado

Perguntas? Bora conversar depois da sessão.



`thiagogonzaga.dev`

meu site



`javaday-babylon-gpu`

código desta palestra



`ThrillhouseBot`

projeto open source

`github.com/devops-thiago · openjdk.org/projects/babylon · thiago@ensina.tech`