

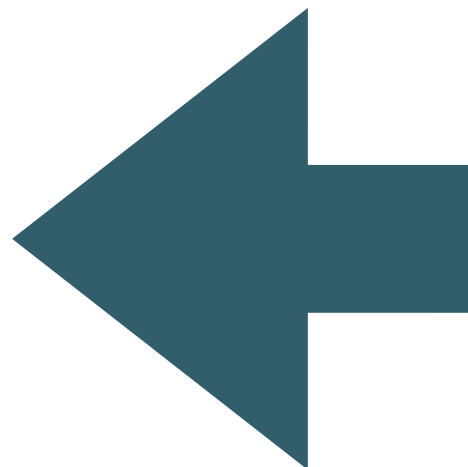
Programar para GPUs

Afinal tínhamos um Ferrari parado no nosso computador, mesmo ao lado de um 2 Cavalos

Alcides Fonseca
me@alcidesfonseca.com
Universidade de Coimbra, Portugal

About me

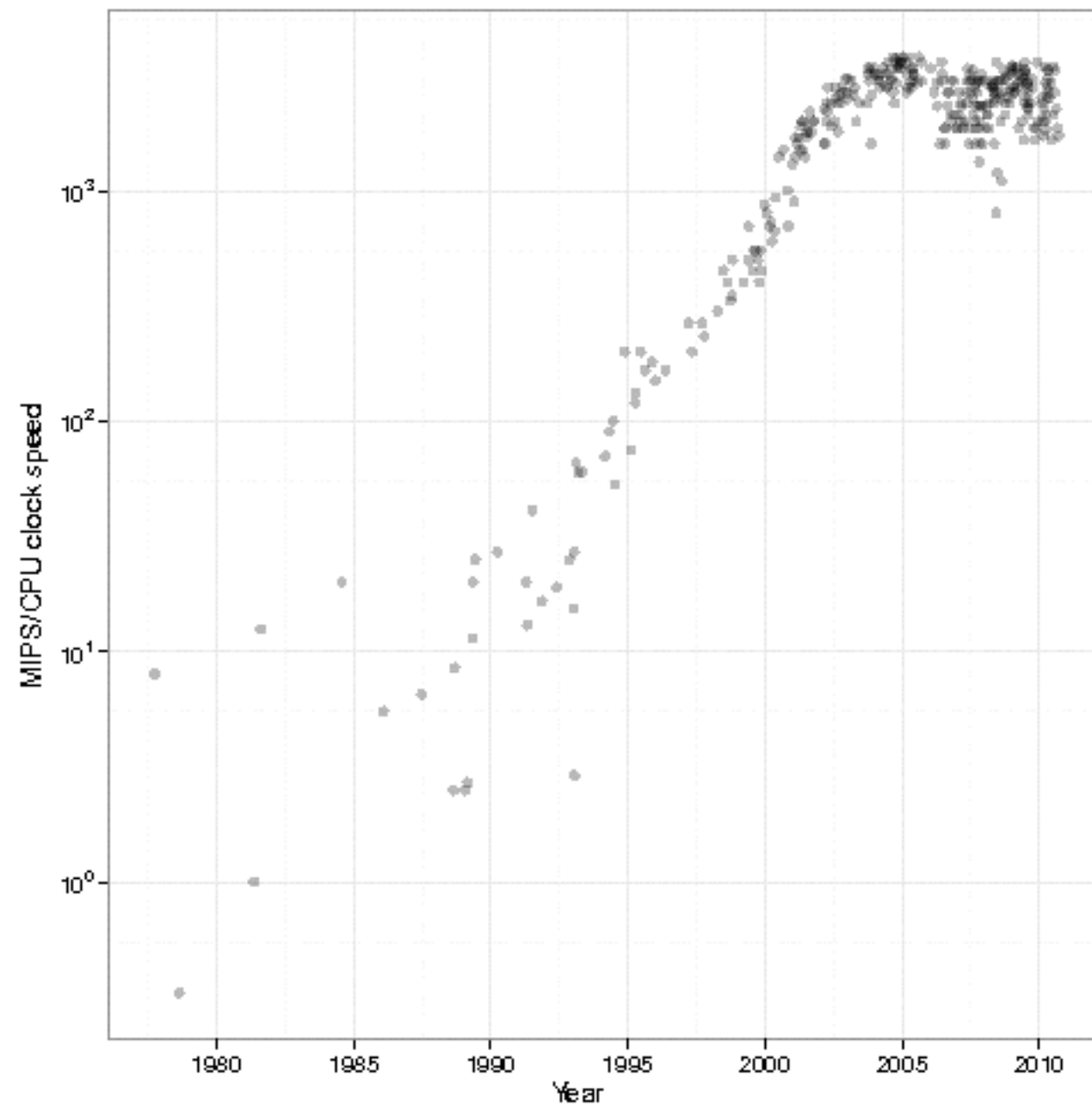
- Web Developer (Django, Ruby, PHP, ...)
- Programador Excêntrico (Haskell, Scala)
- Investigador (GPGPU Programming)
- Docente (Sistemas Distribuídos, Sistemas Operativos e Compiladores)



Esta apresentação

- 20 Minutos - Bla bla bla
- 20 Minutos - `printf("Code\n");`
- 20 Minutos - Q&A

Lei de Moore

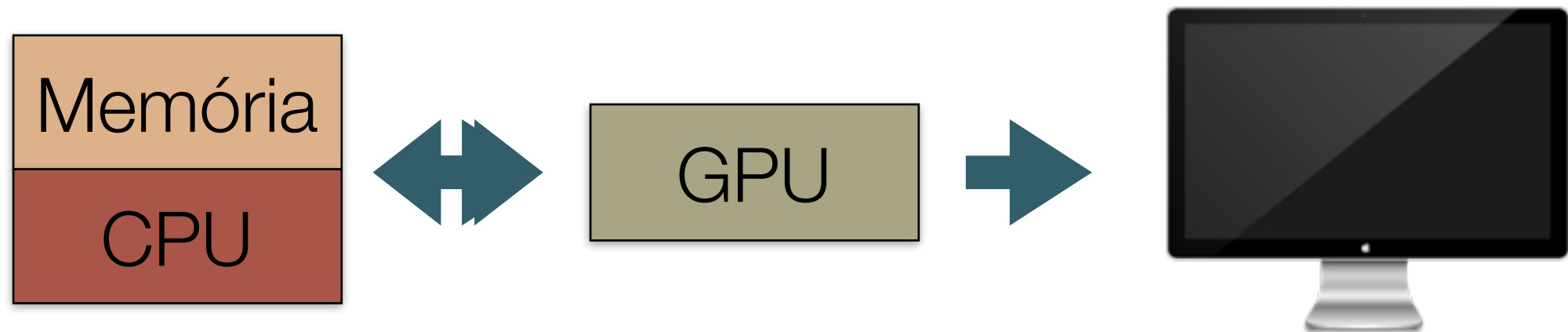


Go multicore!

Paralelismo

	Workstation 2010	Server #1 2011	Server #2 2013
CPU	Dual Core @ 2.66GHz	2x6x2 Threads @ 2.80 GHz	2x8x2 Threads @ 2.00 GHz
RAM	4GB	24GB	32 GB

GPGPU



GPGPU

- **Surgiu de Hackers Cientistas**

- Análise visual de Robots
- Cracking de passwords UNIX
- Redes Neurais

- **Hoje em dia:**

- Sequenciação de DNA
- Previsão de Sismos
- Geração de compostos Químicos
- Previsões e Análises Financeiras
- Cracking de passwords WiFi
- BitCoin Mining

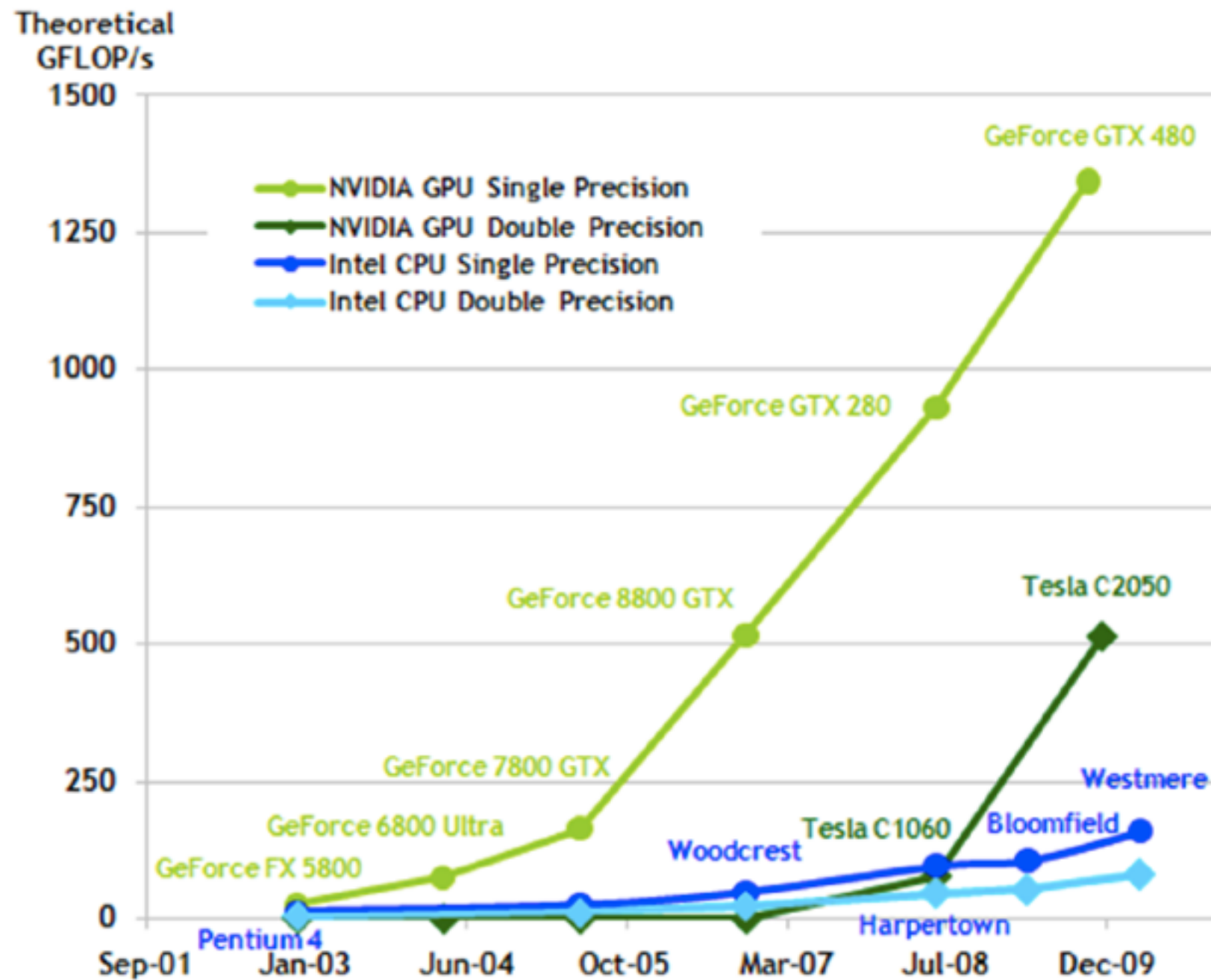
Paralelismo

	Workstation 2010	Server #1 2011	Server #2 2013
CPU	Dual Core @ 2.66GHz	2x6x2 Threads @ 2.80 GHz	2x8x2 Threads @ 2.00 GHz
RAM	4GB	24GB	32 GB
GPU	NVIDIA Geforce GTX	NVIDIA Quadro 4000	AMD Firepro V4900
GPU #Cores	240 (1508MHz)	256 (950MHz)	480 (800MHz)
GPU memory	1GB	2GB	1GB

Back of the napkin

	Workstation 2010	Server #1 2011	Server #2 2013
CPU	2 Cores @ 2.66GHz	2x6x2 Threads @ 2.80 GHz	2x8x2 Threads @ 2.00 GHz
CPU Cores x Frequency	5,32 GHz	<67,2 GHz	<64 GHz
GPU #Cores	240 (1508MHz)	256 (950MHz)	480 (800MHz)
GPU Cores x Frequency	361,92 GHz	243,2 GHz	384 GHz

Benchmarks

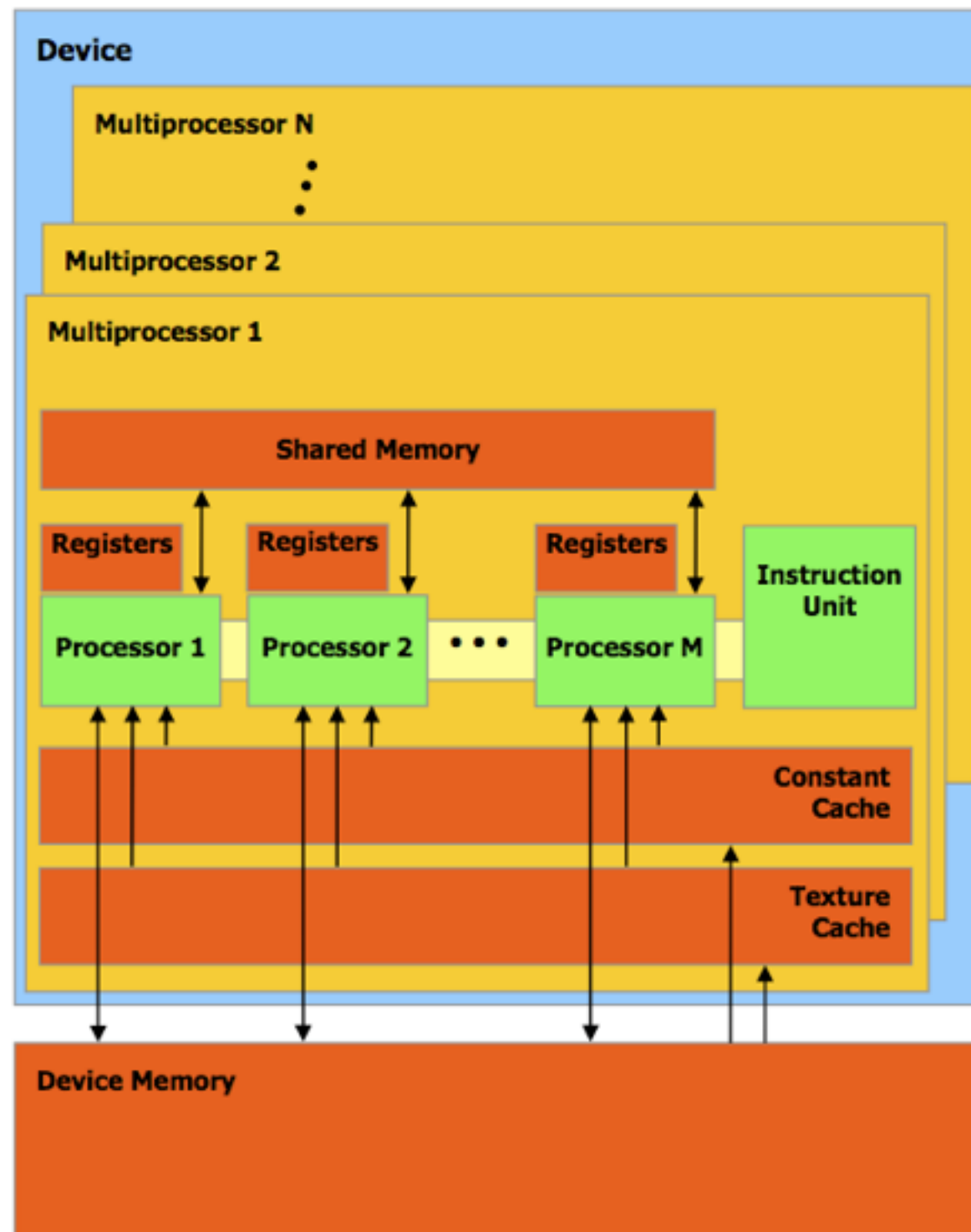


Mas se as GPUs são assim tão poderosas, porque é que ainda usamos CPUs???

Problema #1 - Memória limitada

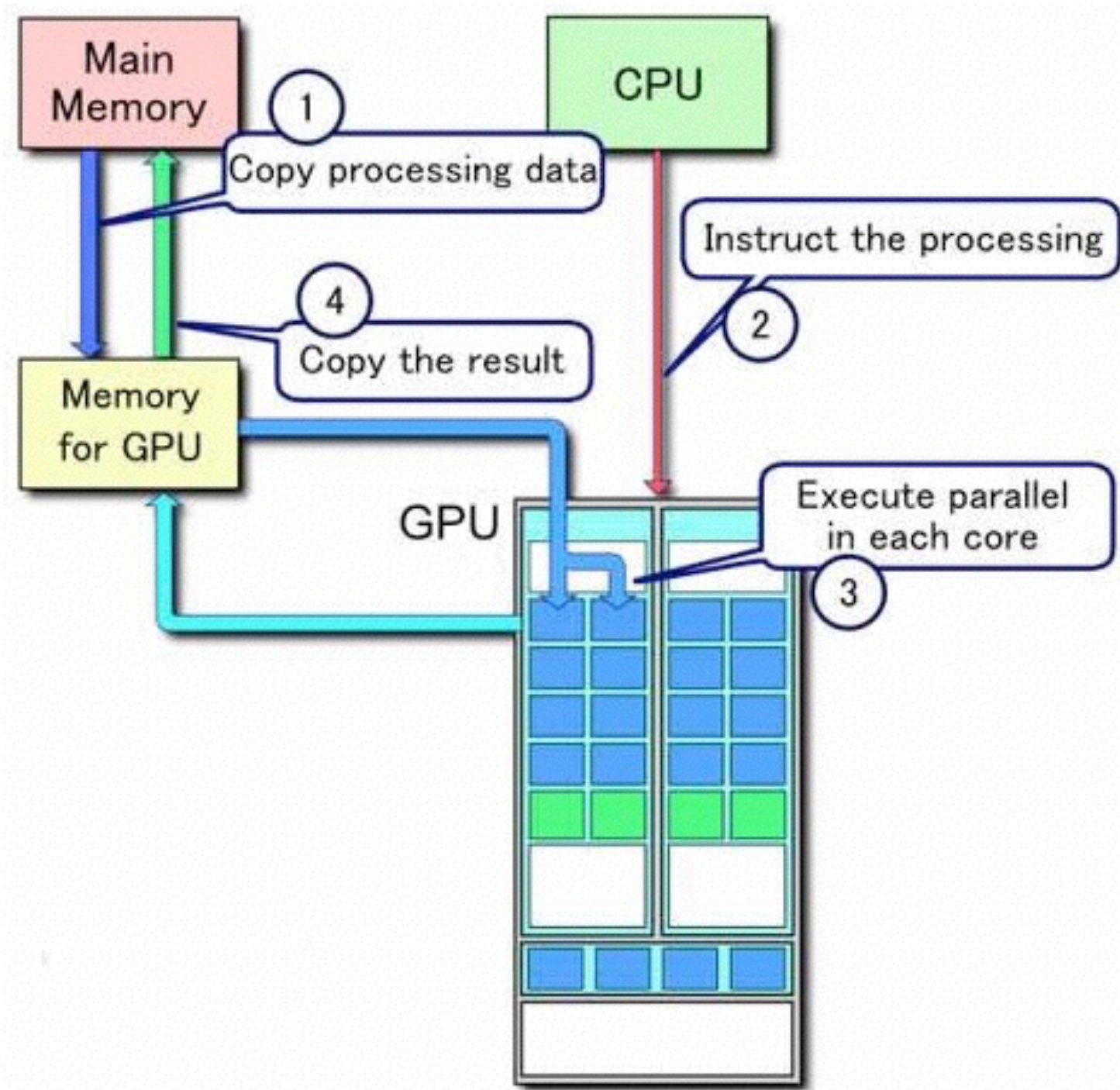
	Workstation 2010	Server #1 2011	Server #2 2013
RAM	4GB	24GB	32 GB
GPU memory	1GB	2GB	1GB

Problema #2 - Diferentes memórias

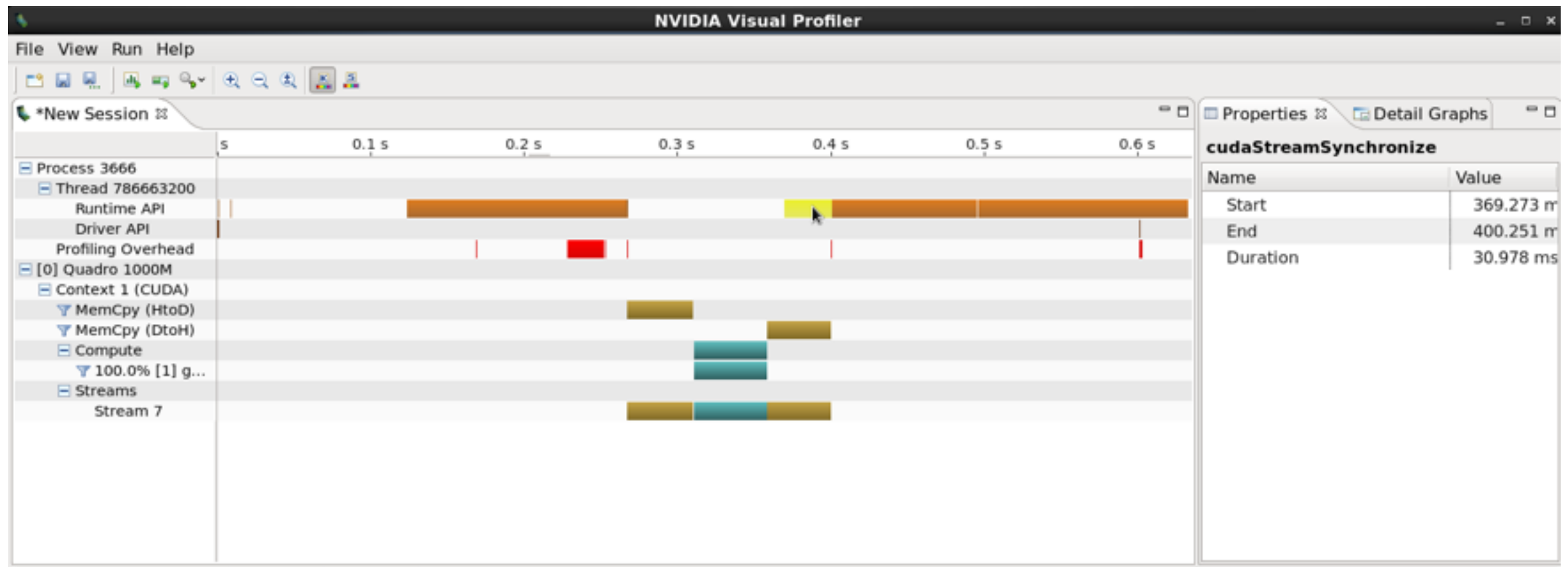


Lentíssimo

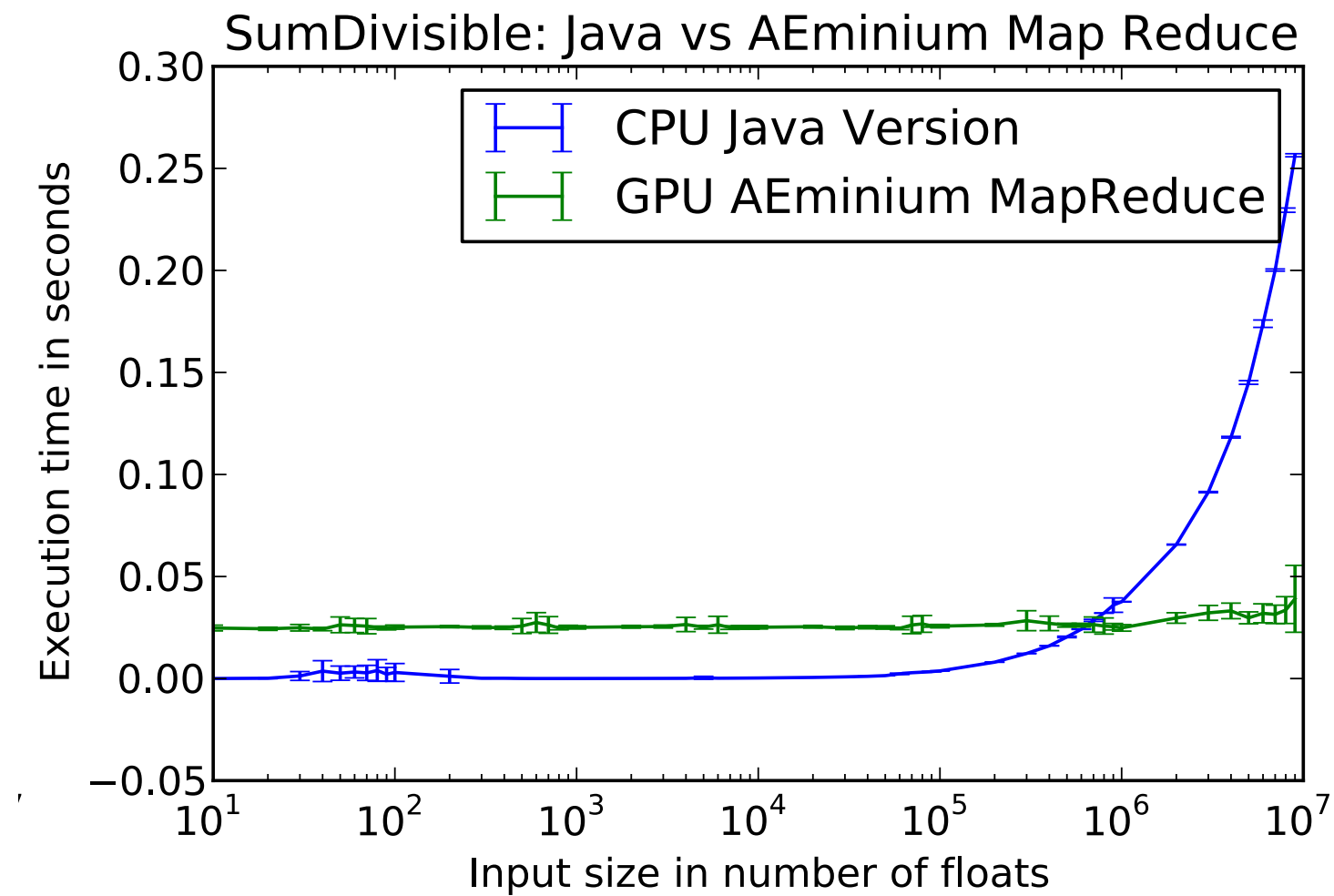
Problema #2 - Diferentes memórias



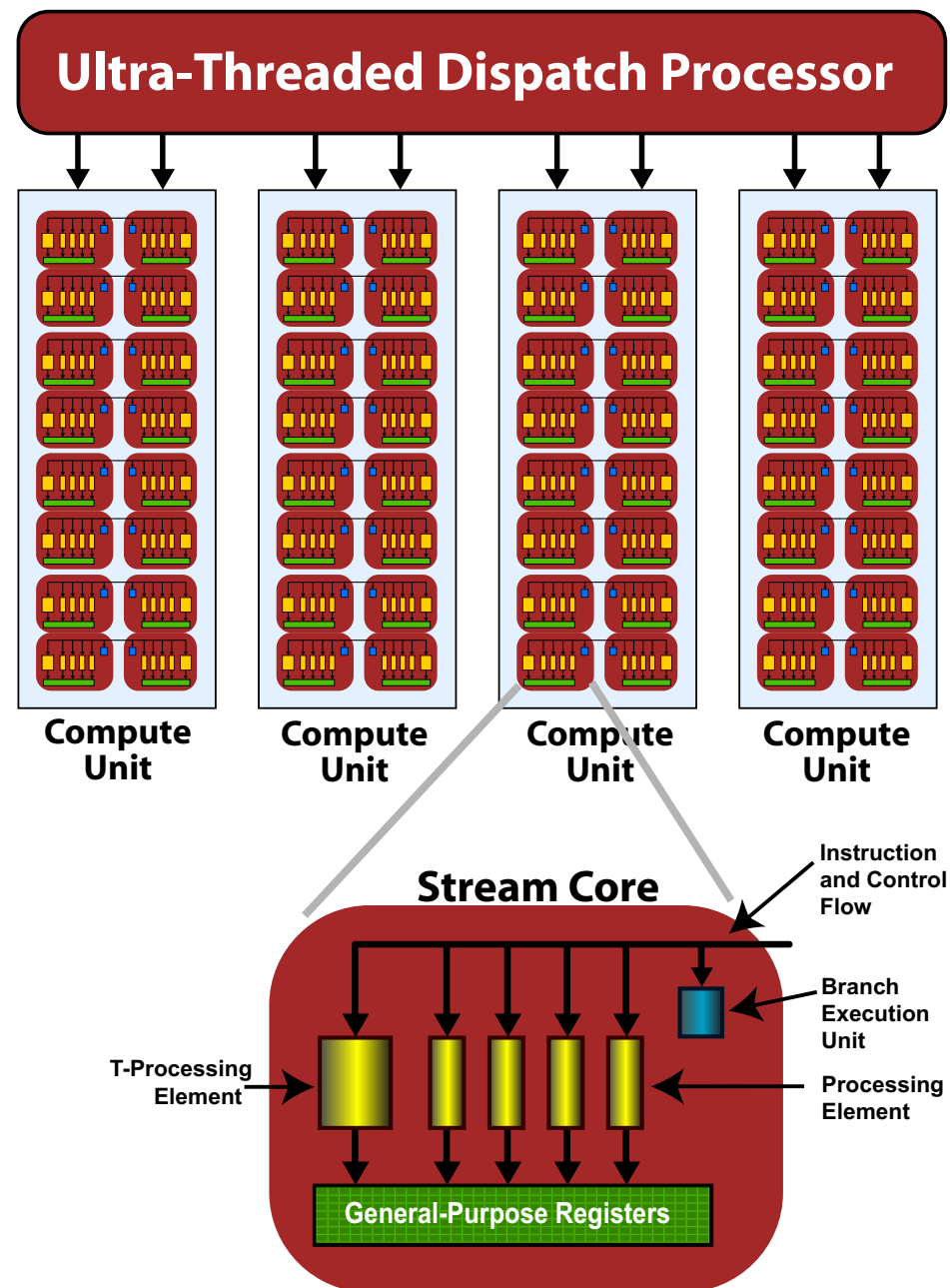
Problema #2 - Diferentes memórias



Problema #2 - Diferentes memórias



Problema #3 - Branching is a bad idea



```
if (threadId.x%2==0)
{
    // do something
} else {
    // do other thing
}
```

Thread Divergence

Figure 1.2 Simplified Block Diagram of the GPU Compute Device¹

Resumindo

CPU	GPU
MIMD	SIMD
task parallel	data parallel
low throughput	high throughput
low latency	high latency

Problema #4 - It's hard

CPU sum

```
int sum = 0;
for (int i=0; i<array.length; i++)
    sum += array[i];
```

```
#ifndef GROUP_SIZE
#define GROUP_SIZE (64)
#endif

#ifndef OPERATIONS
#define OPERATIONS (1)
#endif

////////////////////////////////////
////////////////////////////////////

#define LOAD_GLOBAL_I2(s, i) \
    vload2((size_t)(i), (__global const int*)(s))

#define STORE_GLOBAL_I2(s, i, v) \
    vstore2((v), (size_t)(i), (__global int*)(s))

////////////////////////////////////
////////////////////////////////////

#define LOAD_LOCAL_I1(s, i) \
    ((__local const int*)(s))[(size_t)(i)]

#define STORE_LOCAL_I1(s, i, v) \
    ((__local int*)(s))[(size_t)(i)] = (v)

#define LOAD_LOCAL_I2(s, i) \
    (int2)((LOAD_LOCAL_I1(s, i)), \
           (LOAD_LOCAL_I1(s, i + GROUP_SIZE)))

#define STORE_LOCAL_I2(s, i, v) \
    STORE_LOCAL_I1(s, i, (v)[0]); \
    STORE_LOCAL_I1(s, i + GROUP_SIZE, (v)[1])

#define ACCUM_LOCAL_I2(s, i, j) \
{ \
    int2 x = LOAD_LOCAL_I2(s, i); \
    int2 y = LOAD_LOCAL_I2(s, j); \
    int2 xy = (x + y); \
    STORE_LOCAL_I2(s, i, xy); \
}

////////////////////////////////////
////////////////////////////////////

__kernel void
reduce(
    __global int2 *output,
    __global const int2 *input,
    __local int2 *shared,
    const unsigned int n)
{
    const int2 zero = (int2)(0.0f, 0.0f);
    const unsigned int group_id = get_global_id(0) / get_local_size(0);
    const unsigned int group_size = GROUP_SIZE;
    const unsigned int group_stride = 2 * group_size;
    const size_t local_stride = group_stride * group_size;
    unsigned int op = 0;
    unsigned int last = OPERATIONS - 1;
    for(op = 0; op < OPERATIONS; op++)
    {
        const unsigned int offset = (last - op);
        const size_t local_id = get_local_id(0) + offset;
        STORE_LOCAL_I2(shared, local_id, zero);

        size_t i = group_id * group_stride + local_id;
        while (i < n)
        {
            int2 a = LOAD_GLOBAL_I2(input, i);
            int2 b = LOAD_GLOBAL_I2(input, i + group_size);
            int2 s = LOAD_LOCAL_I2(shared, local_id);
            STORE_LOCAL_I2(shared, local_id, (a + b + s));
            i += local_stride;
        }
    }
}
```

GPU sum

```
barrier(CLK_LOCAL_MEM_FENCE);
#if (GROUP_SIZE >= 512)
    if (local_id < 256) { ACCUM_LOCAL_I2(shared, local_id, local_id + 256); }
#endif

barrier(CLK_LOCAL_MEM_FENCE);
#if (GROUP_SIZE >= 256)
    if (local_id < 128) { ACCUM_LOCAL_I2(shared, local_id, local_id + 128); }
#endif

barrier(CLK_LOCAL_MEM_FENCE);
#if (GROUP_SIZE >= 128)
    if (local_id < 64) { ACCUM_LOCAL_I2(shared, local_id, local_id + 64); }
#endif

barrier(CLK_LOCAL_MEM_FENCE);
#if (GROUP_SIZE >= 64)
    if (local_id < 32) { ACCUM_LOCAL_I2(shared, local_id, local_id + 32); }
#endif

barrier(CLK_LOCAL_MEM_FENCE);
#if (GROUP_SIZE >= 32)
    if (local_id < 16) { ACCUM_LOCAL_I2(shared, local_id, local_id + 16); }
#endif

barrier(CLK_LOCAL_MEM_FENCE);
#if (GROUP_SIZE >= 16)
    if (local_id < 8) { ACCUM_LOCAL_I2(shared, local_id, local_id + 8); }
#endif

barrier(CLK_LOCAL_MEM_FENCE);
#if (GROUP_SIZE >= 8)
    if (local_id < 4) { ACCUM_LOCAL_I2(shared, local_id, local_id + 4); }
#endif

barrier(CLK_LOCAL_MEM_FENCE);
#if (GROUP_SIZE >= 4)
    if (local_id < 2) { ACCUM_LOCAL_I2(shared, local_id, local_id + 2); }
#endif

barrier(CLK_LOCAL_MEM_FENCE);
#if (GROUP_SIZE >= 2)
    if (local_id < 1) { ACCUM_LOCAL_I2(shared, local_id, local_id + 1); }
#endif

}

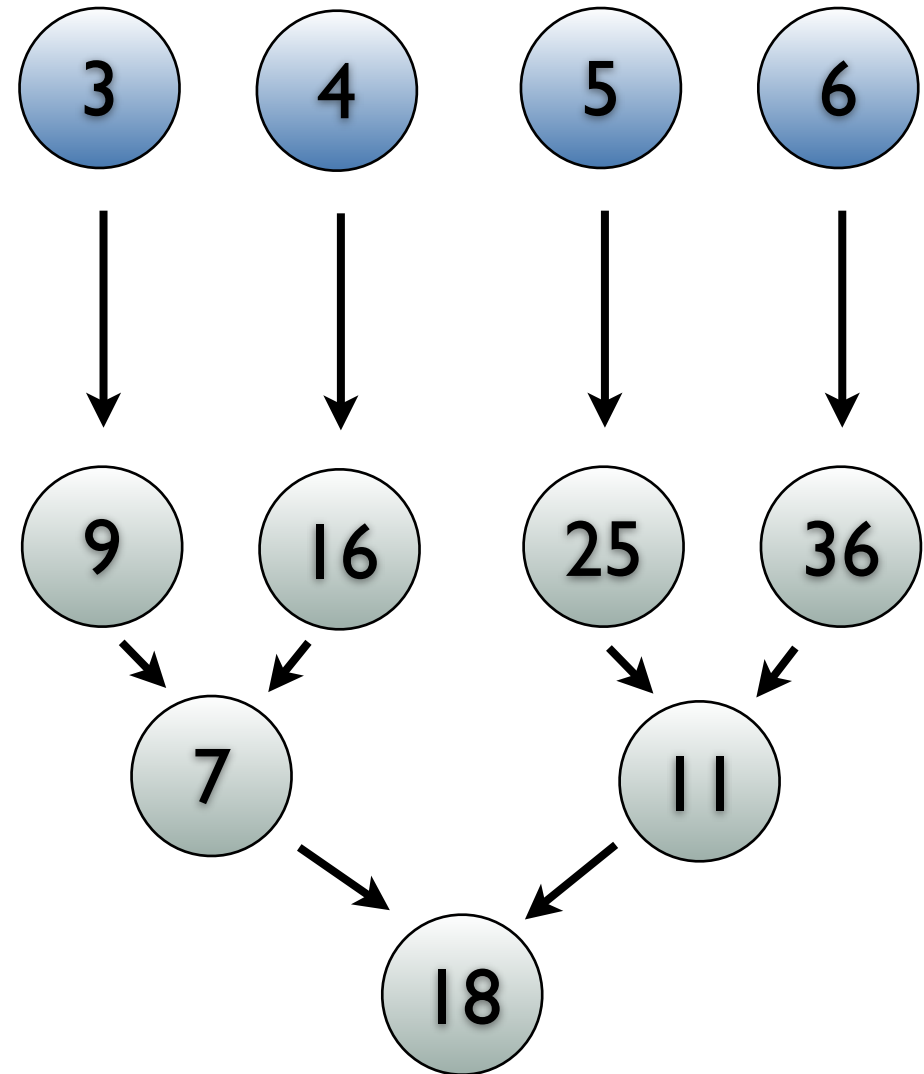
barrier(CLK_LOCAL_MEM_FENCE);
if (get_local_id(0) == 0)
{
    int2 v = LOAD_LOCAL_I2(shared, 0);
    STORE_GLOBAL_I2(output, group_id, v);
}
}
```

Como programar para GPUs?

- CUDA (NVidia)
- OpenCL (Apple, Intel, NVidia, AMD)
- OpenACC (Microsoft)
- MATLAB
- Accelerate, MARS, **ÆminiumGPU**

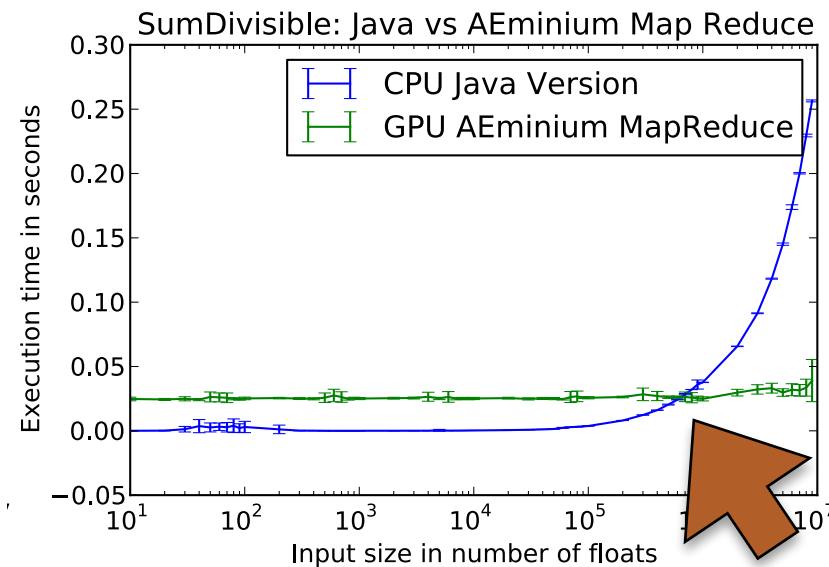
ÆminiumGPU

`map( $\lambda x . x^2$ , [3,4,5,6])`



`reduce(  $\lambda xy . x+y$  , [3,4,5,6])`

ÆminiumGPU Decision Mechanism



Name	Size	C/R	Description
OuterAccess	3	C	Global GPU memory read.
InnerAccess	3	C	Local (thread-group) memory read. This area of the memory is faster than the global one.
ConstantAccess	3	C	Constant (read-only) memory read. This memory is faster on some GPU models.
OuterWrite	3	C	Write in global memory.
InnerWrite	3	C	Write in local memory, which is also faster than in global.
BasicOps	3	C	Simplest and fastest instructions. Include arithmetic, logical and binary operators.
TrigFuns	3	C	Trigonometric functions, including <i>sin</i> , <i>cos</i> , <i>tan</i> , <i>asin</i> , <i>acos</i> and <i>atan</i> .
PowFuns	3	C	<i>pow</i> , <i>log</i> and <i>sqrt</i> functions
CmpFuns	3	C	<i>max</i> and <i>min</i> functions
Branches	3	C	Number of possible branching instructions such as <i>for</i> , <i>if</i> and <i>whiles</i>
DataTo	1	R	Size of input data transferred to the GPU in bytes.
DataFrom	1	R	Size of output data transferred from the GPU in bytes.
ProgType	1	R	One of the following values: Map, Reduce, PartialReduce or MapReduce, which are the different types of operations supported by ÆminiumGPU.

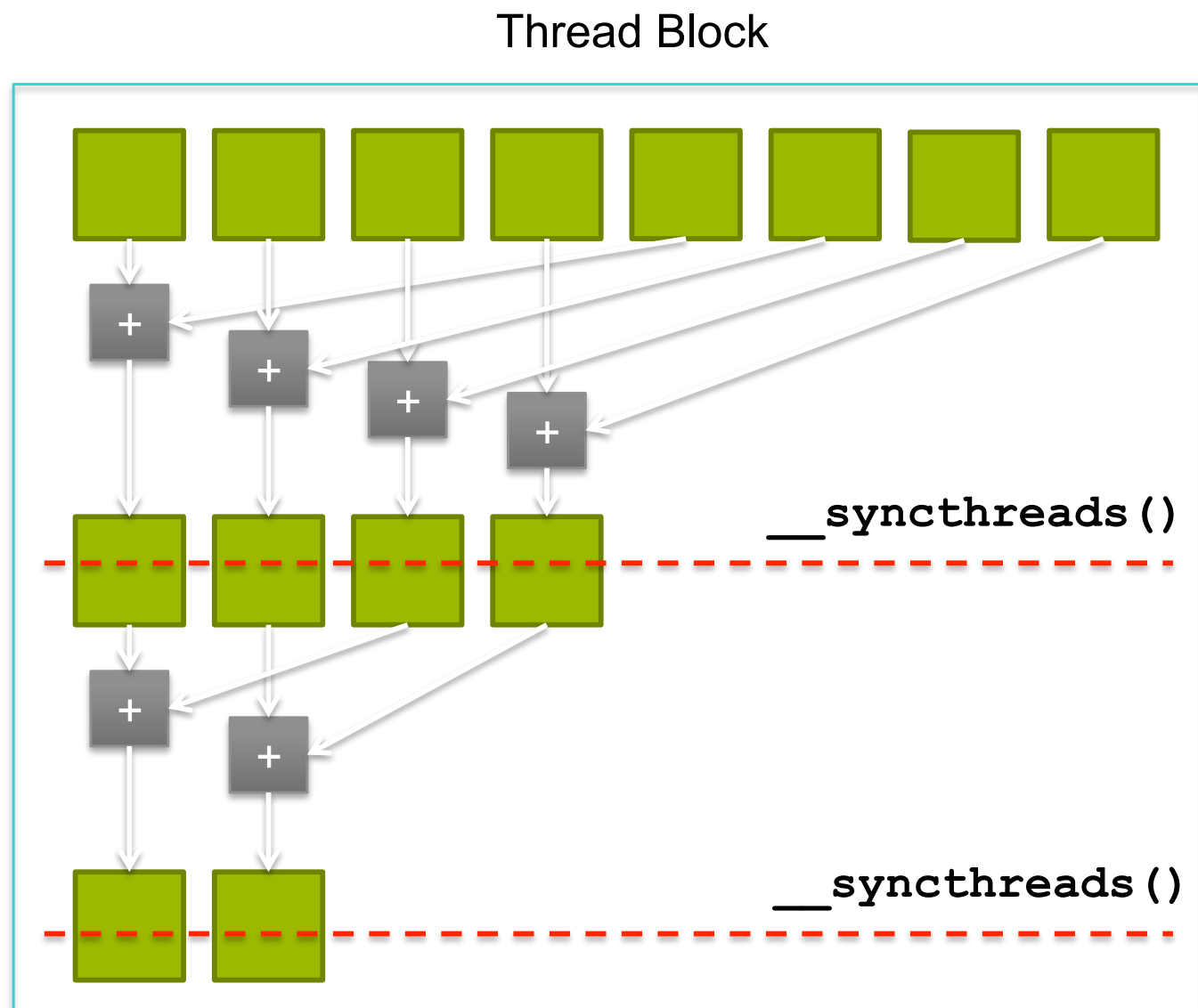
Código (Cuda & OpenCL)

Reduction

Input:

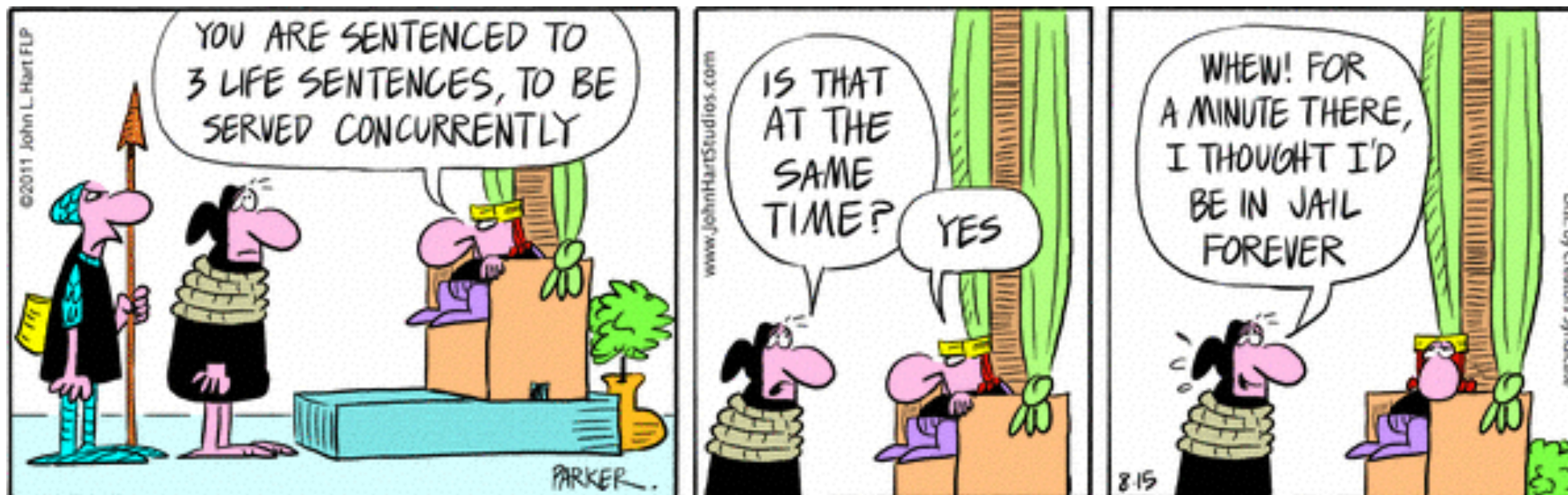
Reduction step 1:

Reduction step 2:



Avanços recentes

- Kernel calls from GPU
- Suporte para Multi-GPU
- Unified Memory
- Task parallelism (HyperQ)
- Melhores profilers
- Suporte para C++ (auto e lambda)



Alcides Fonseca
me@alcidesfonseca.com