fernando@dcc.ufmg.br

Programming Language Laboratory

Code Optimization Techniques for Graphics Processing Units

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

fernando@dcc.ufmg.br

The Objective of this Course is to teach general code optimization principles that apply on the development of CUDA programs

- Graduate course
  - Code generation and optimization
  - High performance computing
- From the perspective of the CUDA developer
- From the perspective of the compiler writer

fernando@dcc.ufmg.br

http://homepages.dcc.ufmg.br/~fernando/classes/gpuOpt/

Code optimization techniques for GPUs

Books on parallel programming theory often talk about such weird beasts like the PRAM model, a hypothetical hardware that would provide the programmer with a number of processors that is proportional to the input size of the problem at hand. Modern general purpose computers afford only a few processing units; four is currently a reasonable number. This limitation makes the development of highly parallel applications quite difficult to the average computer user. However, the low cost and the increasing programmability of graphics processing units, popularly know as GPUs, is contributing to overcome this difficulty. Presently, the application developer can have access, for a few dollars, to a hardware boosting hundreds of processing elements. This brave new world that is now open to many programmers brings, alongside the incredible possibilities, also difficulties and challenges. Perhaps, for the first time since the popularization of computers, it makes sense to open the compiler books on the final chapters, which talk about very unusual concepts, such as polyhedral loops, iteration space and Fourier-Motzkin transformations, only to name a few islands in this enchanted ocean. This material consists of a three-days course, that covers, in a very condensed way, some code generation and optimization techniques that a compiler would use to produce efficient code for graphics processing units. Through these techniques, the compiler writer tries to free the application developer from the intricacies and subtleties of GPU programming, giving him more freedom to focus on algorithms instead of micro-optimizations. We will discuss a little bit of what are GPUs, which applications should target them, how the compiler sees a GPU program and how the compiler can transform this program so that it will take more from this very powerful hardware.

I took much of what is in this page from publicly available books, slides and websites. Hence, nothing more fair than to say that this material can be used free of charges for whoever happens to stumble upon it. Drop me an e-mail if you want the tex sources, if you have any comments whatsoever, of if you just want to say hello.

This book (in Portuguese) contains all the material that I teach in this short course. It is divided in three chapters: GPU fundamentals, memory optimization techniques and divergence optimization techniques. A shorter version of this book will be a chapter in the proceedings of "Jornada de Atualização em Informática" (JAI).

I am using many examples in the textbook and in the slides. All these examples are available in this zip file. I have run them on a Mac OS 10.5.8 feature a NVIDIA GeForce 9400M GPU.

In the first chapter of this course I talk about the history of GPUs and the C for CUDA programming model. I show how to translate some simple C programs to CUDA, relying on the concept of iteration space to help in the understanding.

In the second chapter I discuss many different types of memory optimization techniques that we can use to speed up CUDA programs, such as coalesced access, asynchronous data transference and loop tiling. These are normally the largest sources of performance improvement that we can apply on GPU kernels.

In this final chapter I discuss divergence analysis and optimizations. Because GPUs organize threads as SIMD machines, branches and other irregularities in the program's control flow might be a source of inefficiencies. There are ways to find out which parts of the program may diverge, and which parts may not, and we explore these algorithms.

---

fernando@dcc.ufmg.br

The never ending story

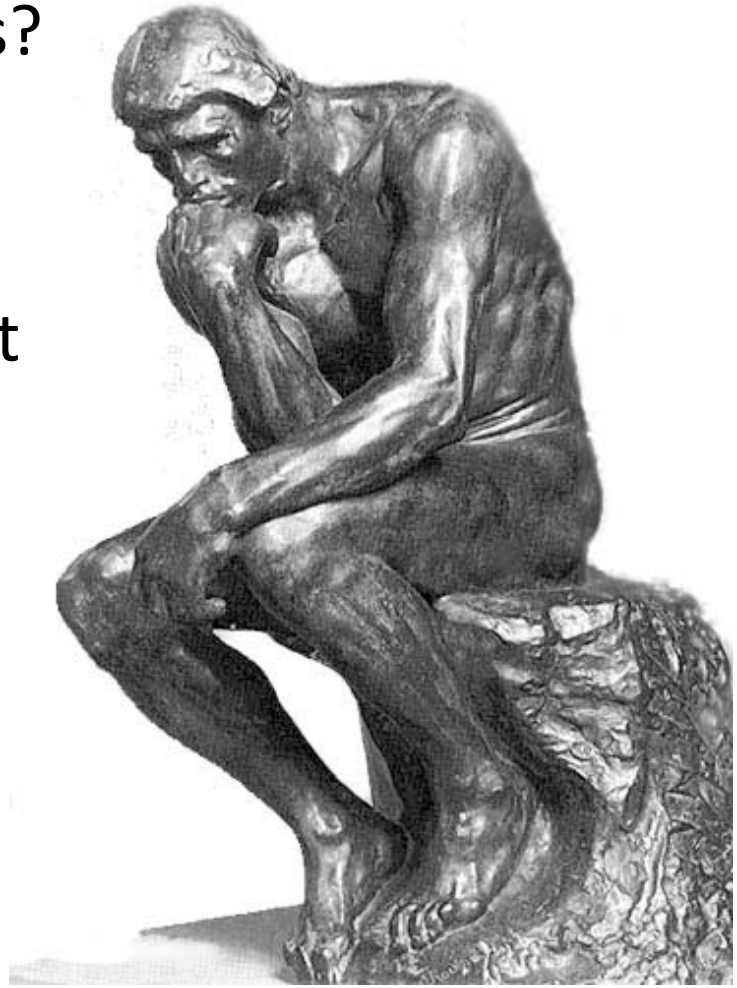
- The hardware becomes each day more efficient.
  - And more complicated
- Programming languages become each day more expressive
  - And harder to implement efficiently.

The compiler links these two worlds

fernando@dcc.ufmg.br

The Rise of GPU computing

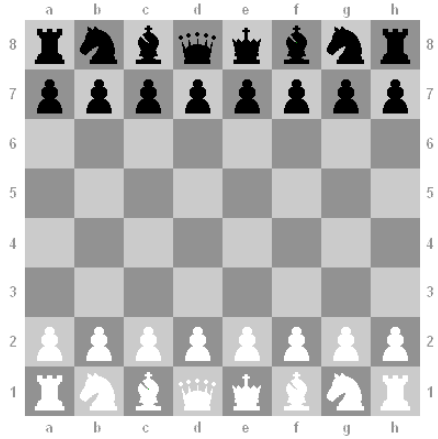
- What are graphics processing units?
- Which applications most benefit from GPUs?
- What are SIMD machines and what does this concept has to do with GPUs?
- How to port my C program to run on a GPU?



fernando@dcc.ufmg.br

Fiat Lux

- What is a Graphics Processing Unit (GPU)?
- 15 years ago nobody would talk about GPUs. What was in charge of doing graphics computing in the PCs at that time?
- How to do rasterization and shading?
- How to get a GPU working on your computer?



fernando@dcc.ufmg.br

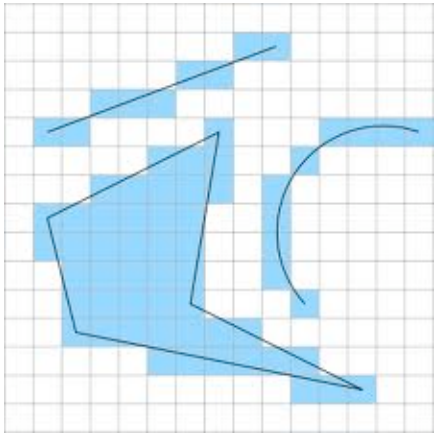
The Wheel of Reincarnation

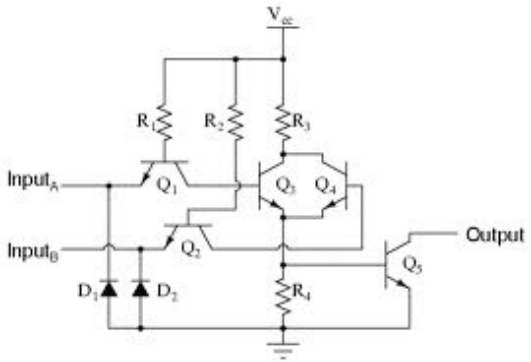
- In the good old days, the graphics hardware was just the VGA. **All the processing was in software**.
- Do you know how the frame buffer works?
- Can you program the VGA standard in any way?
- People started complaining: software is slow…
  - But, what do you want to run at the hardware level?

fernando@dcc.ufmg.br

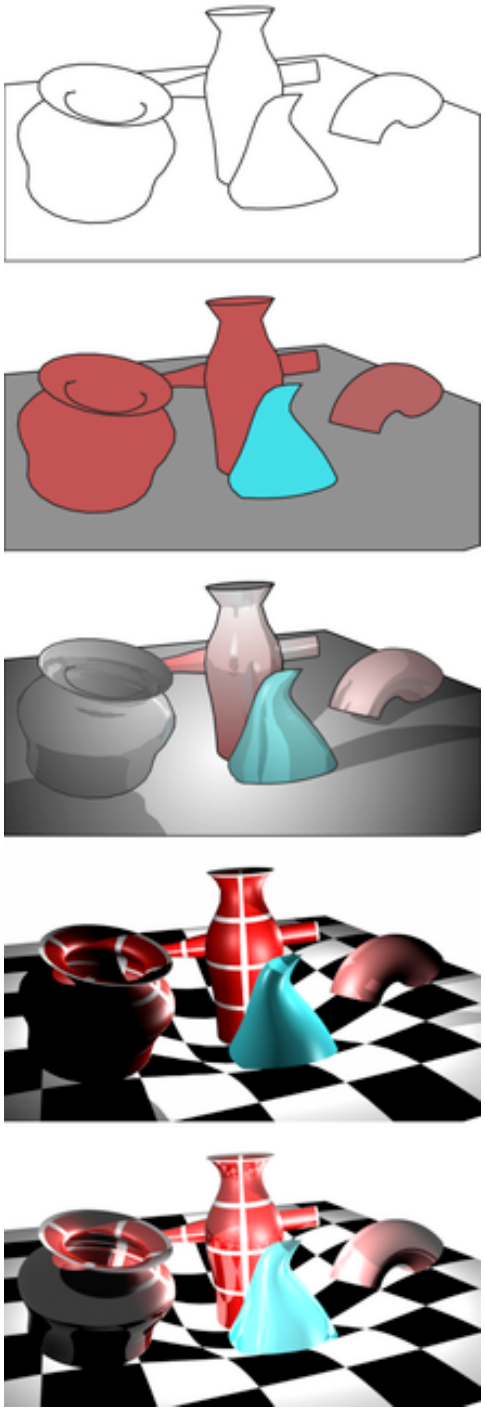
The Wheel of Reincarnation

- Some functions, like the rasterizer, are heavily used. What is rasterization?
- Better to **implement these functions in hardware**.
  - How do you implement a function at the hardware level?
  - What is the main advantage of doing it this way?
  - What are the drawbacks?
- Can you program those functions implemented at the hardware level?





Graphics Pipeline



- Graphics can be processed in a **pipeline**.
  - Transform, project, clip, display, etc…
- Some functions, although different, can be implemented by very similar hardware.
  - Shading is an example. Do you know what is the *shader*?
- Add a graphics API to program the shaders.
  - But this API is so specific… and the hardware is so powerful… what a waste!

fernando@dcc.ufmg.br

General Purpose Hardware

- Let's add a **instruction set** to the shader.
  - Let's augment this hardware with **general purpose** integer operations.
  - What about adding some **branching** machinery too?
- Hum… add a **high level language** on top of this stuff.
  - Plus a lot of documentation. Advertise it! It should look cool!
- Oh boy: we have now two general purpose processors.
  - We should unify them. The rant starts all over again…

fernando@dcc.ufmg.br

1.5 turns around the wheel

- Lets add a display processor to the display processor
  - After all, it is a waste to leave this powerful hardware unused most of the time…

Dedicated rasterizer

fernando@dcc.ufmg.br

Brief Timeline

Year	Transistors	Model	Tech
1999	25M	GeForce 256	DX7, OpenGL
2001	60M	GeForce 3	Programmable Shader
2002	125M	GeForce FX	Cg programs
2006	681M	GeForce 8800	C for CUDA
2008	1.4G	GeForce GTX 280	IEEE FP
2010	3.0G	Fermi	Cache, C++

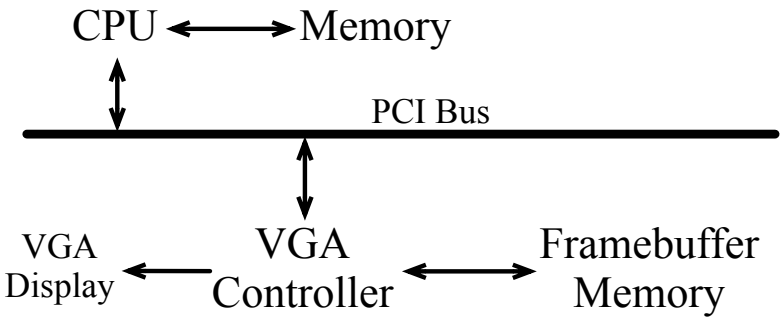
fernando@dcc.ufmg.br

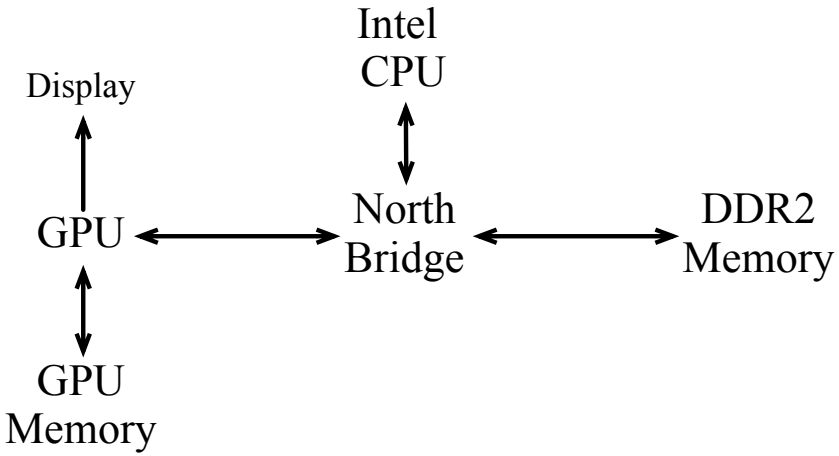
A Cyborg Programming Environment

- CPUs and GPUs form a **heterogeneous** processing environment. What does this mean?
  - How are GPUs and CPUs similar?
  - How are they different?
  - Which applications are good for the CPU?
  - What about the GPU?

fernando@dcc.ufmg.br

More on heterogeneity



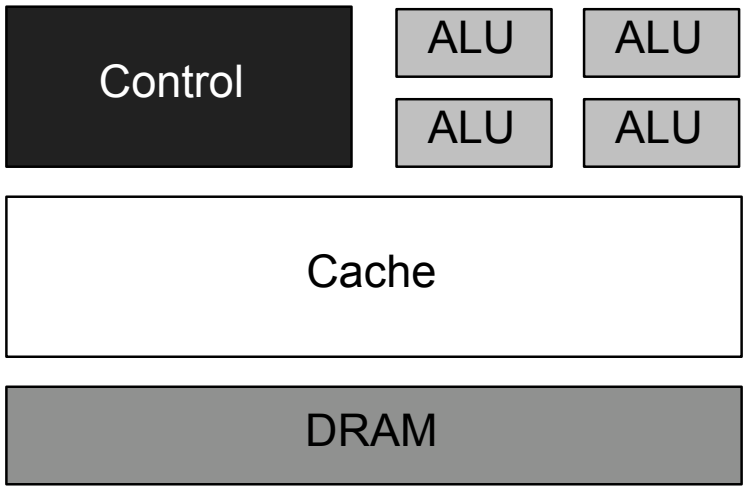


- There are other examples of heterogeneous processors, even in your every-day PC. Can you think about other examples?
- Who decides which tasks run on the CPU, and which tasks are sent to the GPU?

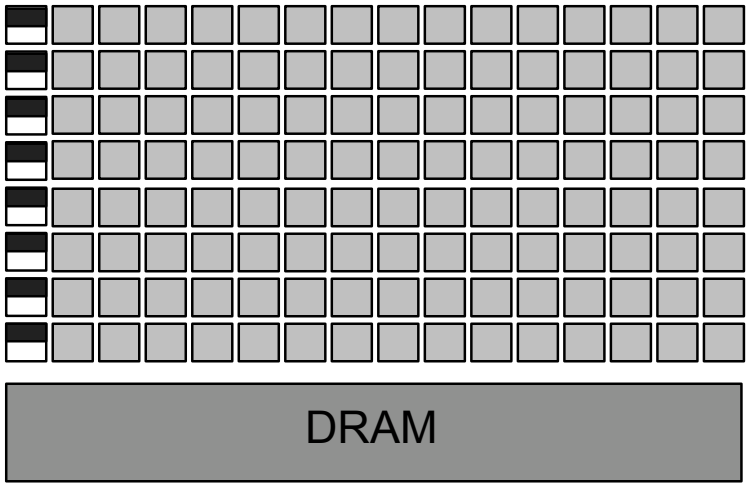
fernando@dcc.ufmg.br

Set, Aim, Fire!!!

- How can we transpose captain, soldiers, guns (and roses) to the SIMD world?
  - What about the fake bullets?



CPU



GPU

fernando@dcc.ufmg.br

Computer Organization

- GPUs show different types of parallelism
  - Single Instruction Multiple Data (SIMD)
  - Single Program Multiple Data (SPMD)
- In the end, we end up with a **MSIMD** hardware.
- Why GPUs are so parallel? And why traditional CPUs do not show off all this parallelism?
- SIMD, MIMD, … what else can we find in Flynn's taxonomy?

fernando@dcc.ufmg.br

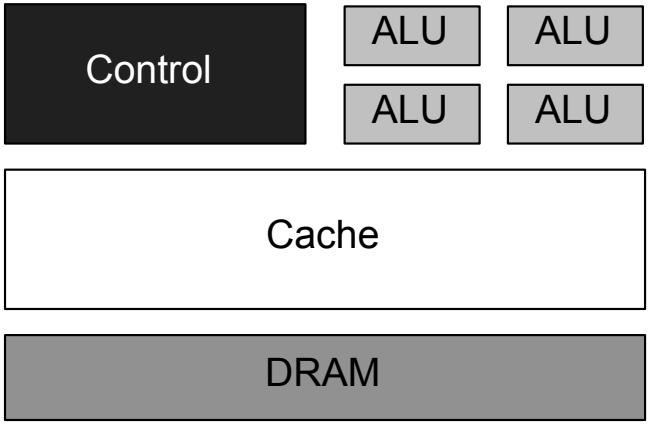
Why SIMD?

- What are the advantages of a SIMD machine?
- What about the disadvantages?

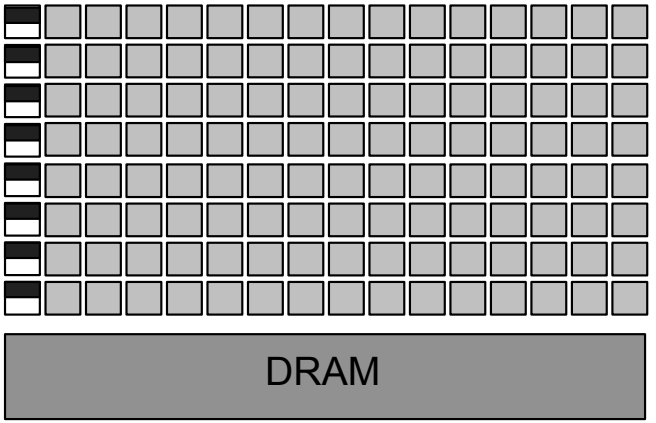
fernando@dcc.ufmg.br

Quite not traditional…

- Caches are good things…
  - The CPU chip reserves a lot of area for caches.
- Why do caches play such a big role in CPUs?
- Caches are relatively small in GPUs.
  - Why?



CPU



GPU

fernando@dcc.ufmg.br

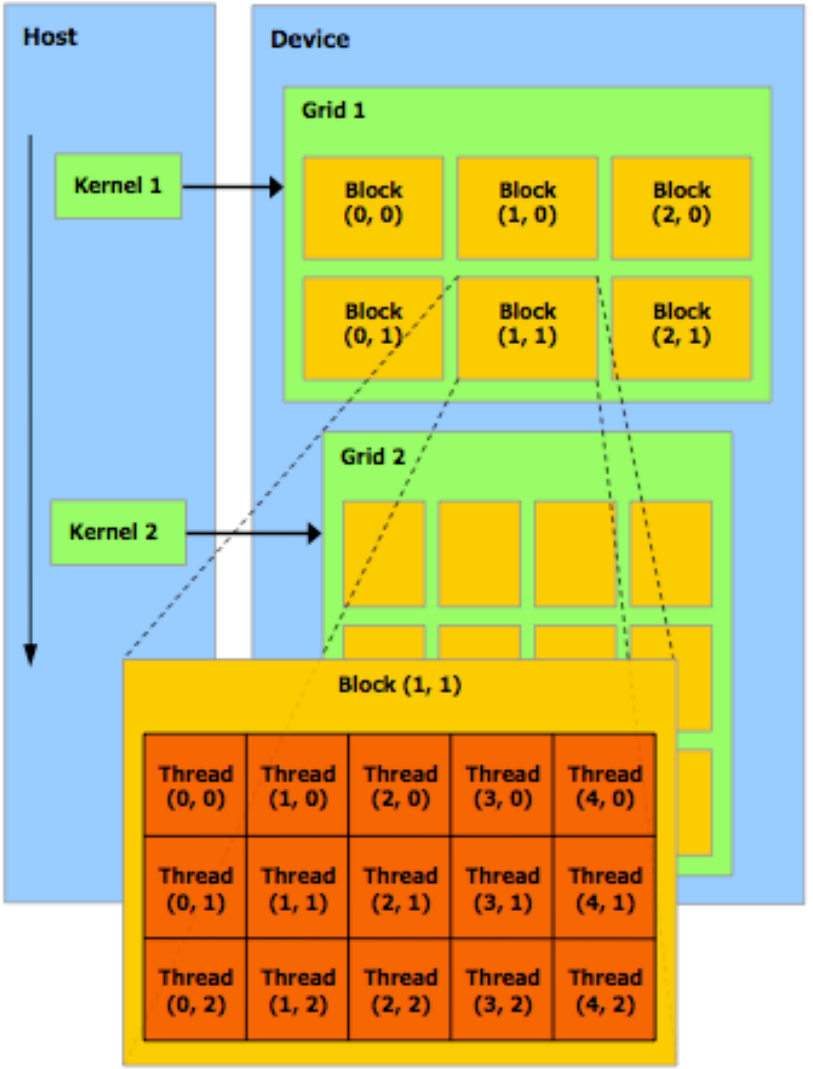
Man Plus

- There are programming languages for graphics processing:
  - Cg, HLSL
  - These languages are very domain specific.
- C for Cuda is much more general purpose!
- What do you think a language like C for CUDA should have? What are the main abstractions that this language should provide?

fernando@dcc.ufmg.br

A Hybrid Programming Language

- Threads are grouped in warps, blocks and grids
- Threads in different grids do not talk to each other
  - Grids are divided in blocks
- Threads in the same block share memory and barriers
  - Blocks are divided in warps
- Threads in the same warp follow the SIMD model.



fernando@dcc.ufmg.br

Threads

- What is really a CUDA thread?
  - At the hardware level
  - At the programming language level
- Why do we have this hierarchy, grouping threads in warps, blocks, and grids?
  - Is there any of these elements that would be wasteful?

fernando@dcc.ufmg.br

The first Cuda program

```
void saxpy_serial(int n, float alpha, float *x, float *y) {
  for (int i = 0; i < n; i++)
    y[i] = alpha*x[i] + y[i];
}
// Invoke the serial function:
saxpy_serial(n, 2.0, x, y);
```

- What is this program doing?
- What is it asymptotic complexity?
- How much can we parallelize it?
  - How much can we parallelize it in CUDA?

fernando@dcc.ufmg.br

The first Cuda program

```
void saxpy_serial(int n, float alpha, float *x, float *y) {
  for (int i = 0; i < n; i++)
    y[i] = alpha*x[i] + y[i];
}
// Invoke the serial function:
saxpy_seral(n, 2.0, x, y);
```

What happened to the loop in the CUDA program?

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) y[i] = alpha * x[i] + y[i];
}
// Invoke the parallel kernel:
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

fernando@dcc.ufmg.br

Understanding the Code

- Can you simplify the program to use only one thread block?
- How many threads will be running simultaneously in this program?
- CUDA did not support recursion originally. Do you know why?

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) y[i] = alpha * x[i] + y[i];
}
// Invoke the parallel kernel:
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

fernando@dcc.ufmg.br

Raising the level

- Cuda programs contain **CPU programs** plus **kernels**
- Kernels are called via a special syntax:

```
kernel<<<dGrd, dBck>>>(A,B,w,C);
```

```
__global__ void matMul1(float* B, float* C, float* A, int w) {
 float Pvalue = 0.0;

 for (int k = 0; k < w; ++k) {
  Pvalue += B[threadIdx.y * w + k] * C[k * w + threadIdx.x];
 }

 A[threadIdx.x + threadIdx.y * w] = Pvalue;
}
void Mul(const float* A, const float* B, int width, float* C)  {
  int size = width * width * sizeof(float);

  // Load A and B to the device
  float* Ad;
  cudaMalloc((void**)&Ad, size);
  cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
  float* Bd;
  cudaMalloc((void**)&Bd, size);
  cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

  // Allocate C on the device
  float* Cd;
  cudaMalloc((void**)&Cd, size);

  // Compute the execution configuration assuming
  // the matrix dimensions are multiples of BLOCK_SIZE
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

  // Launch the device computation
  Muld<<<dimGrid, dimBlock>>>(Ad, Bd, width, Cd);

  // Read C from the device
  cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

  // Free device memory
  cudaFree(Ad);
  cudaFree(Bd);
  cudaFree(Cd);
}
```

fernando@dcc.ufmg.br

Why is coprocessing so good?

CPU core is 5x faster and 50x the area of a GPU core

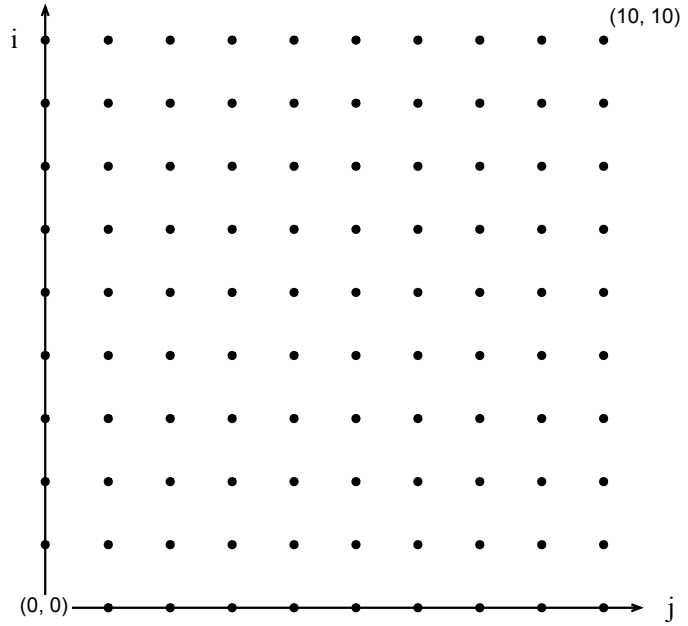
Configuration	**1 CPU core**	**500 GPU cores**	**10 CPUs**	**1 CPU + 450 GPU cores**
Area	**50**	**500**	**500**	**500**
0.5% serial	1.0	5.0	1.0	1.0
99.5% parallel	199.0	1.99	19.9	2.21
Total	**200.0**	**6.99**	**20.9**	**2.21**
75% serial	150.0	750.0	150.0	150.0
25% parallel	50.0	0.5	5.0	0.55
Total	**200.0**	**750.5**	**155.0**	**150.55**

fernando@dcc.ufmg.br

Electronic Cinderella

- Many CUDA programs are direct translations of sequential C programs
  - A very useful concept to obtain these translations is the notion of *iteration space*

```
void matSum(float** s, float** a,
float** b, unsigned int side) {
  int i, j;
  for (i = 0; i < side; i++) {
    for (j = 0; j < side; j++) {
      s[i][j] = a[i][j] + b[i][j];
    }
  }
}
```

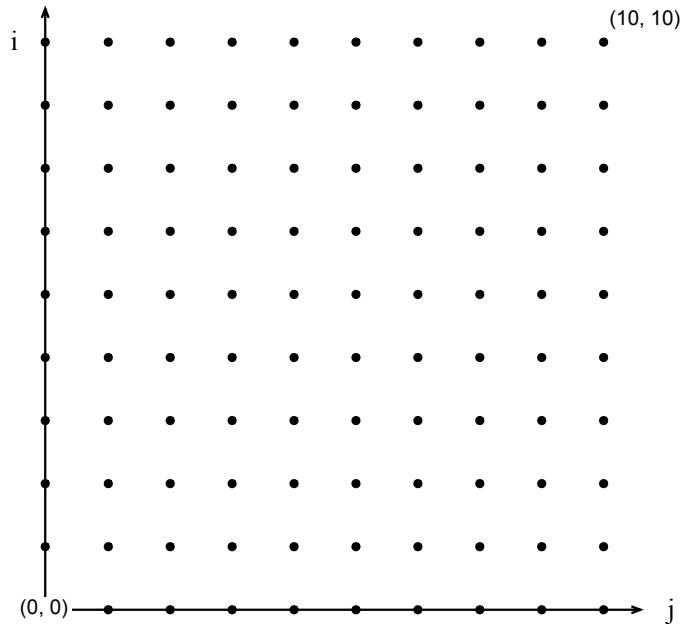


fernando@dcc.ufmg.br

Iteration Space

- One dot for each combination of loop indices
- If iteration *(i, j)* depends on iteration *(x, y)*, then we add an arrow from *(x, y)* to *(i, j)*

```
void matSum(float** s, float** a,
float** b, unsigned int side) {
  int i, j;
  for (i = 0; i < side; i++) {
    for (j = 0; j < side; j++) {
      s[i][j] = a[i][j] + b[i][j];
    }
  }
}
```



fernando@dcc.ufmg.br

Matrix addition

```
__global__
void matSumKernel(float* S, float* A,
float* B, int side) {
  int i = bid.y * bdim.y + tid.y;
  int j = bid.x * bdim.x + tid.x;
  int ij = i*side + j;
  if (ij < side*side) {
    S[ij] = A[ij] + B[ij];
  }
}
```

```
void matSum(float** s, float** a,
float** b, unsigned int side) {
  int i, j;
  for (i = 0; i < side; i++) {
    for (j = 0; j < side; j++) {
      s[i][j] = a[i][j] + b[i][j];
    }
  }
}
```

- We have linearized the matrices in the kernel. Guess why?
- What is the complexity of the original **C program**? What about the **Cuda kernel** on a **PRAM** model?
- What is the block size?

fernando@dcc.ufmg.br

Temporal dependences

- Not every program has a completely independent iteration space
- What is the iteration space of the program below?
  - Are there dependences between different iterations?

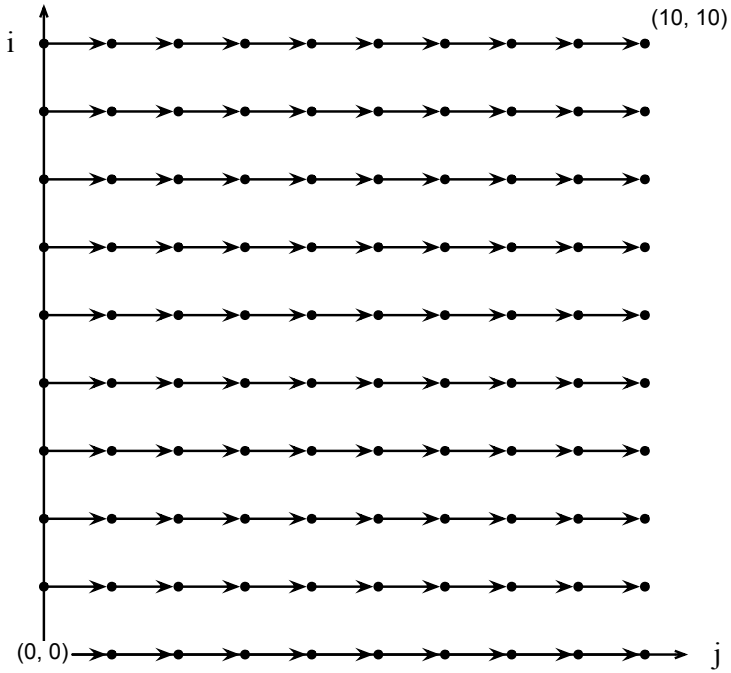
```
void depSum(float** s, int side) {
  int i, j;
  for (i = 0; i < side; i++) {
    for (j = 1; j < side; j++) {
      s[i][j] = s[i][j] - s[i][j-1];
    }
  }
}
```

fernando@dcc.ufmg.br

Temporal dependences

- Not every program has a completely independent iteration space
- What is the iteration space of the program below?
- What is the equivalent kernel?

```
void depSum(float** s, int side) {
  int i, j;
  for (i = 0; i < side; i++) {
    for (j = 1; j < side; j++) {
      s[i][j] = s[i][j] - s[i][j-1];
    }
  }
}
```



fernando@dcc.ufmg.br

Temporal dependences

```
__global__ void depSumKernel
  (float* S, int side) {
  int i = bid.x * bdim.x + tid.x;
  if (i<side) {
    for (int j=1; j<side; j++) {
      int ij = i*side + j;
      S[ij] = S[ij] - S[ij-1];
    }
  }
}
```

Why do we have this conditional test?

```
void depSum(float** s, int side) {
  int i, j;
  for (i = 0; i < side; i++) {
    for (j = 1; j < side; j++) {
      s[i][j] = s[i][j] - s[i][j-1];
    }
  }
}
```

- What is the complexity of the **C program** and the **Cuda program** on a **PRAM** model?
- Do all the threads perform the same amount of work?
- Any geometric intuition about the translation process?

fernando@dcc.ufmg.br

A more complicated example

- What is the iteration space of the program below?

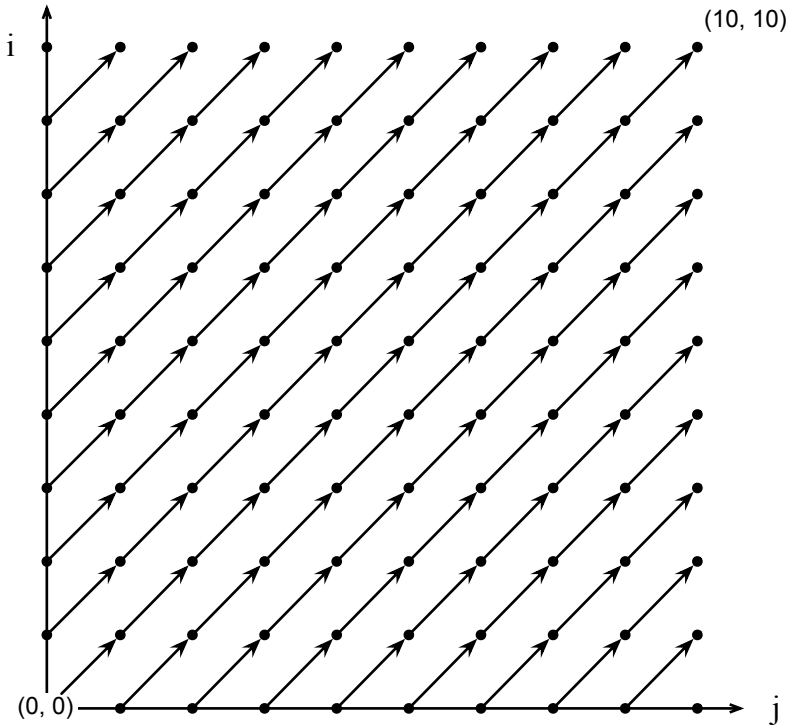
```
void digSum(float** s, int side) {
  int i, j;
  for (i = 1; i < side; i++) {
    for (j = 1; j < side; j++) {
      s[i][j] = s[i][j]
                - s[i-1][j-1];
    }
  }
}
```

fernando@dcc.ufmg.br

A more complicated example

- What is the iteration space of the program below?
- What is the equivalent CUDA Kernel?
- Do you think each thread will do the same work?

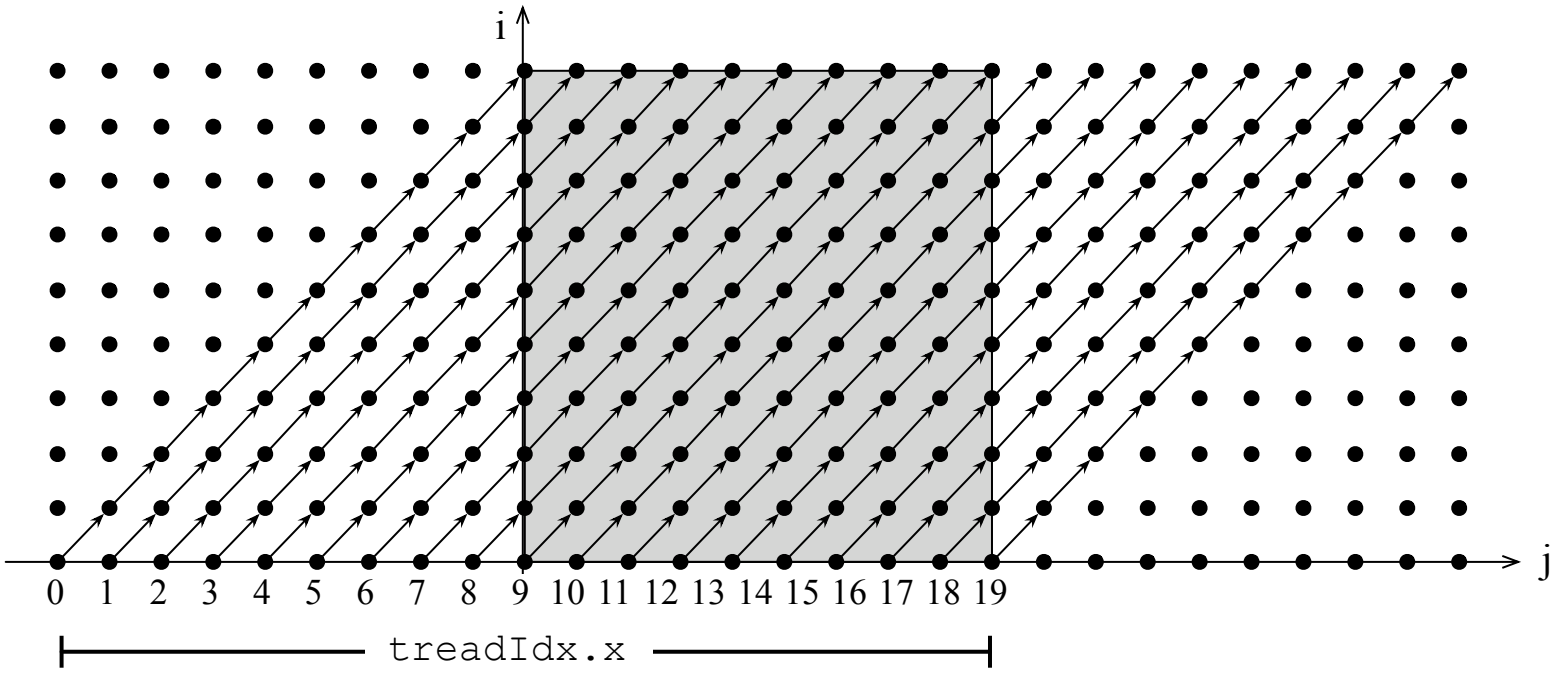
```
void digSum(float** s, int side) {
  int i, j;
  for (i = 1; i < side; i++) {
    for (j = 1; j < side; j++) {
      s[i][j] = s[i][j]
                - s[i-1][j-1];
    }
  }
}
```



fernando@dcc.ufmg.br

A more complicated example

- It is easier if we assume that the threads will all to the same amount of work.
- Can you come up with a kernel?



fernando@dcc.ufmg.br

A more complicated example

```
__global__ void digSumKernel(float* S, int side) {
  int tx = blockIdx.x * blockDim.x + threadIdx.x;
  for (int it=1; it<side; it++) {
    int i = it;
    int j = tx - (side-1) + it;
    if (j >= 1 && j < side) {
      int ij = j + i * side;
      int ijp = j - 1 + (i - 1) * side;
      s[ij] = s[ij] - s[ijp];
    }
  }
}
```

- Does each thread perform the same amount of work?
- What happens at the branch when threads **do not** have work to do?
- What is the complexity of this kernel, on a **PRAM** model?

fernando@dcc.ufmg.br

Multidimensional Iteration Spaces

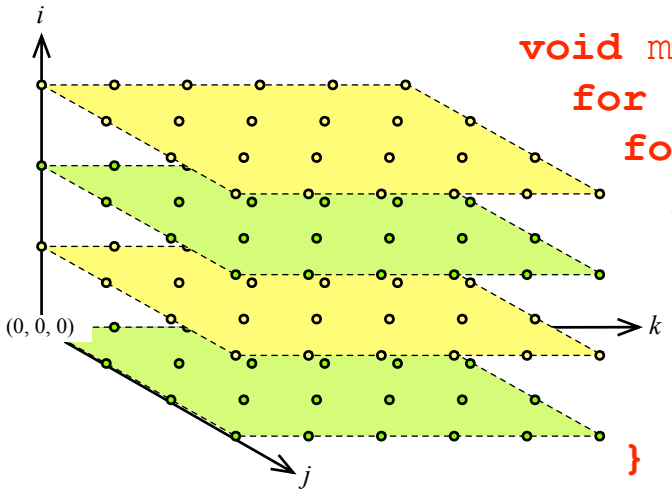
- What is this program below doing?
- How many dimensions does its iteration space have?

```
void mystery(float* B, float* C, float* A, int w) {
  for (unsigned int i = 0; i < w; ++i) {
    for (unsigned int j = 0; j < w; ++j) {
      A[i * w + j] = 0.0;
      for (unsigned int k = 0; k < w; ++k) {
        A[i * w + j] +=
          B[i * w + k] * C[k * w + j];
      }
    }
  }
}
```

fernando@dcc.ufmg.br

Multidimensional Iteration Spaces

- How is the CUDA kernel like?
- What is the complexity of this program, in the **PRAM** model?




```
void matmult(float* B, float* C, float* A, int w) {
  for (unsigned int i = 0; i < w; ++i) {
    for (unsigned int j = 0; j < w; ++j) {
      A[i * w + j] = 0.0;
      for (unsigned int k = 0; k < w; ++k) {
        A[i * w + j] +=
          B[i * w + k] * C[k * w + j];
      }
    }
  }
}
```

fernando@dcc.ufmg.br

Multidimensional Iteration Spaces

```
__global__ void matMul1(float* B, float* C, float* A, int w) {
  float Pvalue = 0.0;
  for (int k = 0; k < w; ++k) {
    Pvalue += B[threadIdx.y * w + k] * C[k * w + threadIdx.x];
  }
  A[threadIdx.x + threadIdx.y * w] = Pvalue;
}
```

- Is there a way to lower this complexity?

```
void matmult(float* B, float* C, float* A, int w) {
  for (unsigned int i = 0; i < w; ++i) {
    for (unsigned int j = 0; j < w; ++j) {
      A[i * w + j] = 0.0;
      for (unsigned int k = 0; k < w; ++k) {
        A[i * w + j] +=
          B[i * w + k] * C[k * w + j];
      }
    }
  }
}
```

fernando@dcc.ufmg.br

Lowering the level

- CUDA assembly is called Parallel Thread Execution (PTX)
- What do you think a SIMD assembly should have?

```
__global__
void saxpy_parallel(int n, float a,
                    float *x, float *y) {
  int i = bid.x * bid.x + tid.x;
  if (i < n) y[i] = a * x[i] + y[i];
}
```

```
.entry saxpy_GPU (n, a, x,  y) {
        .reg .u16 %rh<4>;
        .reg .u32 %r<6>;
        .reg .u64 %rd<8>;
        .reg .f32 %f<6>;
        .reg .pred %p<3>;
$LBB1__Z9saxpy_GPUifPfS_:
        mov.u16         %rh1, %ctaid.x;
        mov.u16         %rh2, %ntid.x;
        mul.wide.u16    %r1, %rh1, %rh2;
        cvt.u32.u16     %r2, %tid.x;
        add.u32         %r3, %r2, %r1;
        ld.param.s32    %r4, [n];
        setp.le.s32     %p1, %r4, %r3;
        @%p1 bra        $Lt_0_770;
        .loc    28      13      0
        cvt.u64.s32     %rd1, %r3;
        mul.lo.u64      %rd2, %rd1, 4;
        ld.param.u64    %rd3, [y];
        add.u64         %rd4, %rd3, %rd2;
        ld.global.f32   %f1, [%rd4+0];
        ld.param.u64    %rd5, [x];
        add.u64         %rd6, %rd5, %rd2;
        ld.global.f32   %f2, [%rd6+0];
        ld.param.f32    %f3, [alpha];
        mad.f32         %f4, %f2, %f3, %f1;
        st.global.f32   [%rd4+0], %f4;
        exit;

}
```

fernando@dcc.ufmg.br

The weird PTX

- PTX is an assembly language, but has some high-level constructs too.
  - Function syntax
  - Data types
  - Special regs: **`%tid, %laneid, %warpid`**, etc
- Instructions include arithmetics (**`add, sub, mul, mad`**), control flow (**`bra`**), memory access (**`ld, st`**), synchronization (**`sync, atom`**), etc

Ex.: compute unified thread id for 2D block

```
mov.u32   %r0, %tid.x;
mov.u32   %h1, %tid.y;
mov.u32   %h2, %ntid.x;
mad.u32   %r0, %h1, %h2, %r0;
```

fernando@dcc.ufmg.br

Many Points of View

- We can embed CUDA in <u>many</u> languages, not only **C**.

```
mod = comp.SourceModule("""
__global__ void vec_mul(float *dest, float *a, float *b) {
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}""")

vec_mul = mod.get_function(”vec_mul")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
vec_mul(drv.Out(dest), drv.In(a), drv.In(b), block=(400,1,1))

print dest-a*b
```

fernando@dcc.ufmg.br

Parallel Prefix Sum

- Design pattern for parallel algorithms.
  - Also called all-prefix-reductions.
- Inclusive scan:

$[a_0, a_1, \ldots, a_{n-1}] \rightarrow [a_0, (a_0 \odot a_1), \ldots, (a_0 \odot a_1 \odot \ldots \odot a_{n-1})]$

- Exclusive scan receives the list plus an identity *I*:

$[a_0, a_1, \ldots, a_{n-1}] \rightarrow [I, a_0, (a_0 \odot a_1), \ldots, (a_0 \odot a_1 \odot \ldots \odot a_{n-2})]$

- $\odot$ must be an operation that is commutative and associative.

fernando@dcc.ufmg.br

Sequential Implementation

- What can we compute by changing the operation ⊙?
- What is the complexity of computing the all-prefix sum of a list with N elements in a **sequential** machine?
- What about on the **PRAM** model?
- Write an implementation of all-prefix sum in C.

fernando@dcc.ufmg.br

Sequential Implementation

- What can we compute by changing the operation ⊙?
- What is the complexity of computing the all-prefix sum of a list with N elements in a **sequential** machine?
- What about on the **PRAM** model?
- Write an implementation of all-prefix sum in C.

```
void allPrefixSum(float* in, float* out, int N) {
  out[0] = 0;
  for (int j = 0; j < N; j++) {
    out[j] = out[j - 1] + in[j - 1];
  }
}
```

- **Can it be parallelized?**

fernando@dcc.ufmg.br

Parallel Implementation

- x[0] = 0
- **for** *d* := 1 to $\log_2 n$ **do**
  - **forall** *k* in **parallel do**
    - **if** *k* >= 2^{d-1} **then** x[k] := x[k – 2^{d-1}] + x[k]

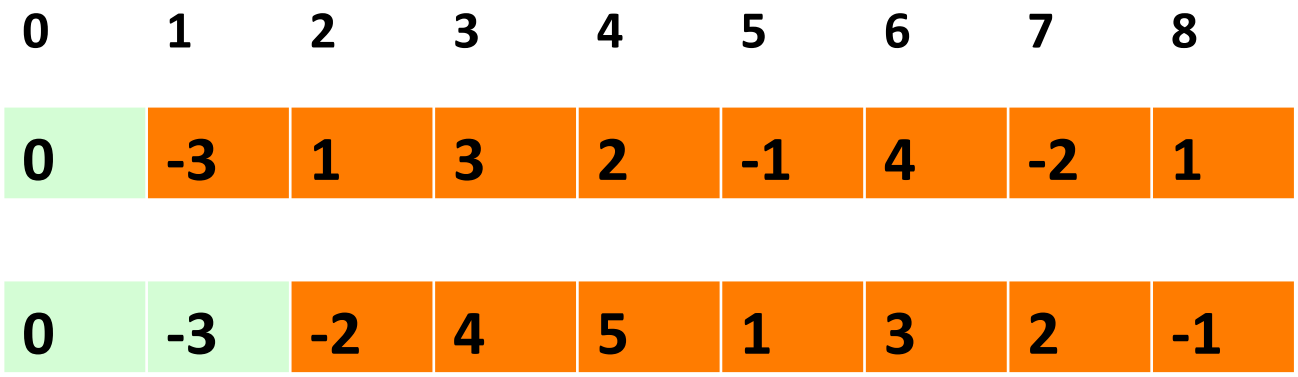
	0	1	2	3	4	5	6	7	8
d = 1	0	-3	1	3	2	-1	4	-2	1

- What would be produced by the first iteration?

fernando@dcc.ufmg.br

Parallel Implementation

- x[0] = 0
- **for** *d* := 1 to $\log_2 n$ **do**
  - **forall** *k* in **parallel do**
    - **if** $k >= 2^{d-1}$ **then** $x[k] := x[k - 2^{d-1}] + x[k]$



fernando@dcc.ufmg.br

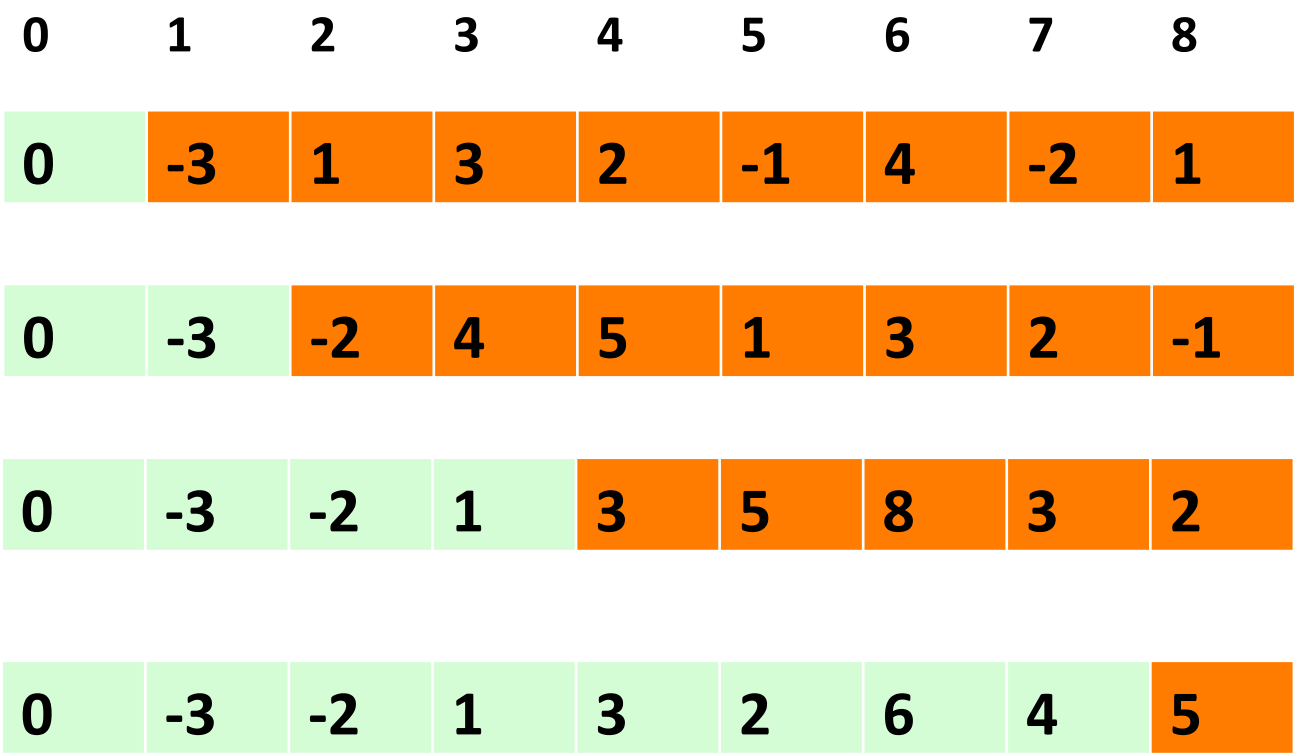
Parallel Implementation

- x[0] = 0
- **for** *d* := 1 to $\log_2 n$ **do**
  - **forall** *k* in **parallel do**
    - **if** $k >= 2^{d-1}$ **then** $x[k] := x[k - 2^{d-1}] + x[k]$

	0	1	2	3	4	5	6	7	8
	0	-3	1	3	2	-1	4	-2	1
	0	-3	-2	4	5	1	3	2	-1
d = 3	0	-3	-2	1	3	5	8	3	2

fernando@dcc.ufmg.br

Parallel Implementation



fernando@dcc.ufmg.br

Parallel Programming is Hard

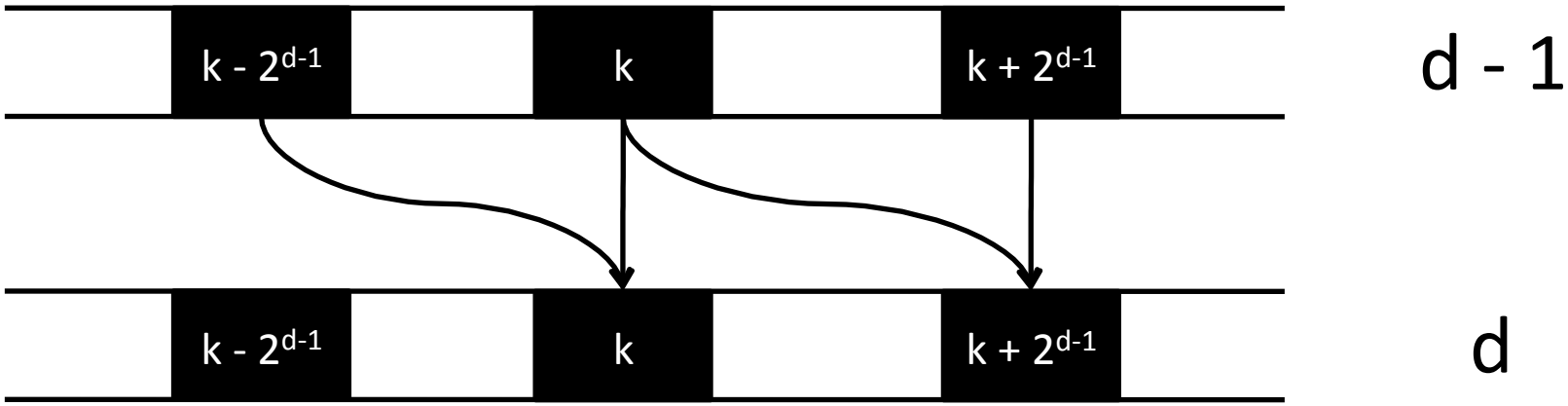
- x[0] = 0
- **for** *d* := 1 to $\log_2$n **do**
  - **forall** *k* in **parallel do**
    - **if** *k* >= 2^{d-1} **then** x[k] := x[k – 2^{d-1}] + x[k]

- This implementation is not exactly correct. Can you find a bug?

fernando@dcc.ufmg.br

Parallel Programming is Hard

- x[0] = 0
- **for** *d* := 1 to $\log_2$n **do**
  - **forall** *k* in **parallel do**
    - **if** *k* >= 2^{d-1} **then** x[k] := x[k – 2^{d-1}] + x[k]



fernando@dcc.ufmg.br

Parallel Programming is Hard

- x[0] = 0
- **for** *d* := 1 to $\log_2$n **do**
  - **forall** *k* in **parallel do**
    - **if** *k* >= 2^{d-1} **then** x[k] := x[k – 2^{d-1}] + x[k]

- The bug is a **data-race**. If every thread were in the same warp, then there would be no bug. Do you see why?
- Programming **SIMD** machines is normally easier than programming **SPMD** machines.
- How to fix this?

fernando@dcc.ufmg.br

Trading space for safety

```
x[0][0] = 0;
in = 0, out = 1;
for d := 1 to log2n do
    forall k in parallel do
        if k >= 2^(d-1) then
            x[out][k] := x[in][k – 2^(d-1)] + x[in][k];
        else
            x[out][k] := x[in][k];
    sync;
    swap(in, out);
```

- Ok, all is good; but how is the **CUDA kernel** for this program?

fernando@dcc.ufmg.br

Prefix sum in CUDA

```
__global__ void scan(float *g_odata, float *g_idata, int n) {
  __shared__  float temp[BLOCK_SIZE * 2];
  int thid = threadIdx.x;
  int pout = 0, pin = 1;
  temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;
  __syncthreads();
  for (int offset = 1; offset < n; offset *= 2) {
    pout = 1 - pout;
    pin  = 1 - pout;
    if (thid >= offset)
      temp[pout * n + thid] += temp[pin * n + thid - offset];
    else
      temp[pout * n + thid] = temp[pin * n + thid];
    __syncthreads();
  }
  g_odata[thid] = temp[pout*n+thid1];
}
```

fernando@dcc.ufmg.br

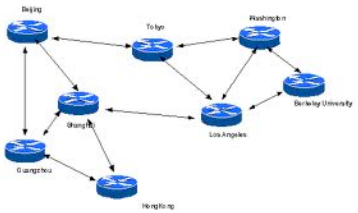
Implementation details

- What happened to the line "**forall** *k* in **parallel do**"?
- Why do we have two data arrays (**`g_odata`** and **`g_idata`**)?
  - Would it be possible to use only one array?
- How do we implement the index swap?
- What would happen if we use more than one thread block in this program?

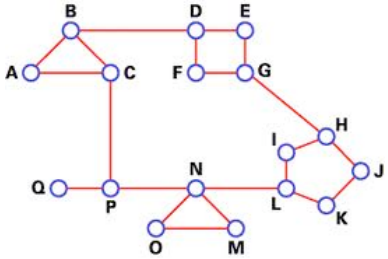
fernando@dcc.ufmg.br

And what can we do with GPUS?

- GPUs have been used to accelerate many different types of applications…
- Lots of publicly available examples:

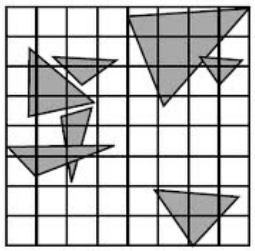


Routing:
Jin Zhao, Xinya Zhang, Xin Wang, and Xiangyang Xue. *Achieving O(1) IP lookup on GPU-based software routers*. SIGCOMM, 40:429–430, 2010.

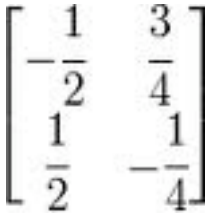


Graphs:
Songpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. *Accelerating CUDA graph algorithms at maximum warp*. In PPoPP. ACM, 2011.

Sorting:
Daniel Cederman and Philippas Tsigas. GPU-quicksort: *A practical quicksort algorithm for graphics processors*. Journal of Experimental Algorithmics, 14(1):4–24, 2009.



Ray-Tracing:
Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. *On-the-fly elimination of dynamic irregularities for GPU computing*. In ASPLOS, pages 369–380. ACM, 2011.



Linear algebra:
Yao Zhang, Jonathan Cohen, and John D. Owens. *Fast tridiagonal solvers on the gpu*. In PPoPP, pages 127–136. ACM, 2010.

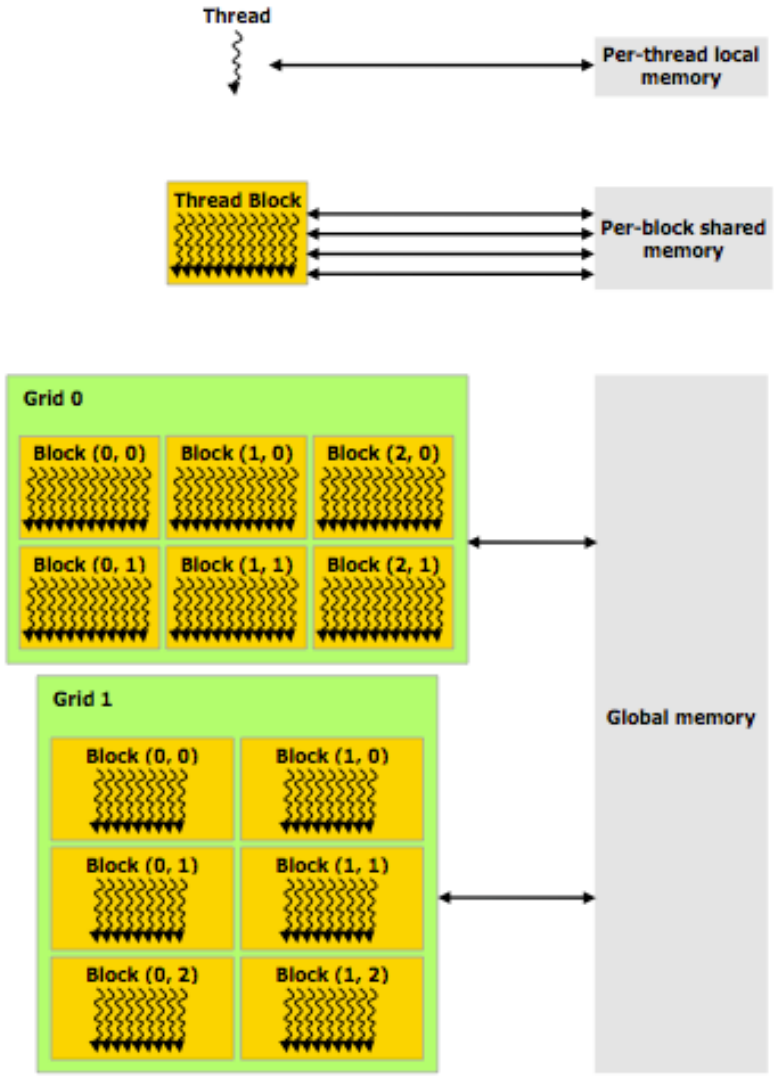
Gene sequencing:
Edans Flavius O. Sandes and Alba Cristina M.A. de Melo. *CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences*. In PPoPP, pages 137–146. ACM, 2010.

Program analysis:
Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. *EigenCFA: Accelerating flow analysis with GPUs*. In POPL. ACM, 2011.

fernando@dcc.ufmg.br

Quick Review

- Each thread has local registers and local memory
- Threads are grouped in warps
  - Warps run in SIMD exec
- Warps are grouped in blocks
  - Shared memory + syncs
- Blocks are grouped in grids
  - Each grid represents a kernel



fernando@dcc.ufmg.br

Pós-graduação DCC-UFMG

- Programa 7 (CAPES)
- Vários grupos de pesquisa. Mestrado e doutorado
- **Doutorado**: inscrição de forma eletrônica - digitalizada, pelo site do PPGCC:

http://www.dcc.ufmg.br/pos

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG