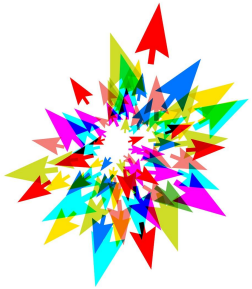
Introduction to GPU programming with CUDA

February 9, 2017

guillimin@calculquebec.ca

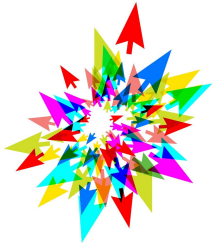
Par : Pier-Luc St-Onge



compute canada | calcul canada

Calcul Québec

Financial Partners



Outline

- Login and Setup
- Overview of GPGPU Computing
- OpenACC directives
- Introduction to CUDA-C
  - Simple CUDA programs
  - Global memory
  - Blocks and threads
  - Syntax for common tasks
  - Error Checking
  - Shared Memory and Thread Cooperation
- Big data case study: K-means clustering

Setup for the workshop

Calcul Québec

1. Get a user ID and password paper (provided in class):

   ##:

   ****

2. Access to local computer (replace "**##**" and "**___**" with appropriate values, "**___**" is provided in class):
   a. User name: **csuser##** (ex.: **csuser99**)
   b. Password: **___@[S##** (ex.: **sec@[S99**)

3. SSH connection to Guillimin (replace **************):
   a. Host name: **guillimin.calculquebec.ca**
   b. User name: **class##** (ex.: **class99**)
   c. Password: **************

Setup for the workshop (Hadès)

Calcul Québec

1. Slides: tinyurl.com/cq-intro-cuda-20170209
2. Each participant will get a unique UserID from 01 to 10
3. Access to local computer (replace "**##**" with UserID, and "**___**" with 3 characters provided in class):
   a. User name: **`csuser##`** (ex.: **`csuser99`**)
   b. Password: **`___@[S##`** (ex.: **`SEC@[S99`**)
4. SSH connection to Hadès:
   a. Host name: **`hades.calculquebec.ca`**
   b. User name: **`user04`** (ex.: **`class99`**)
   c. Password: (given in class)
   d. GPU node: **`ssh ngpu-a4-07`**

Setup for the workshop

Calcul Québec

- (Optional) Start a screen session and note the host name (**`lg-1r**-n**`**):

```
screen -S $USER
screen -d -R $USER  # Only in case connection dropped
```

- Create an interactive job on a K20 node:

```
qsub -I -l nodes=1:ppn=1:gpus=1 -l walltime=7:00:00
# Wait until your job starts on a K20 node
module load CUDA/7.5.18
```

Question #1
Recall: Pointers in C

Calcul Québec

- Go to: http://socrative.com/
- Click on button “Student Login”
- Enter room name: **`PierLucStOnge`**
  - It will display the room name in capital letters
- Wait for the quiz

```
int x = 1;
int *ip;

ip = &x;
*ip = 0;
printf("x = %d\n", x);
```

What is a GPU?

Calcul Québec

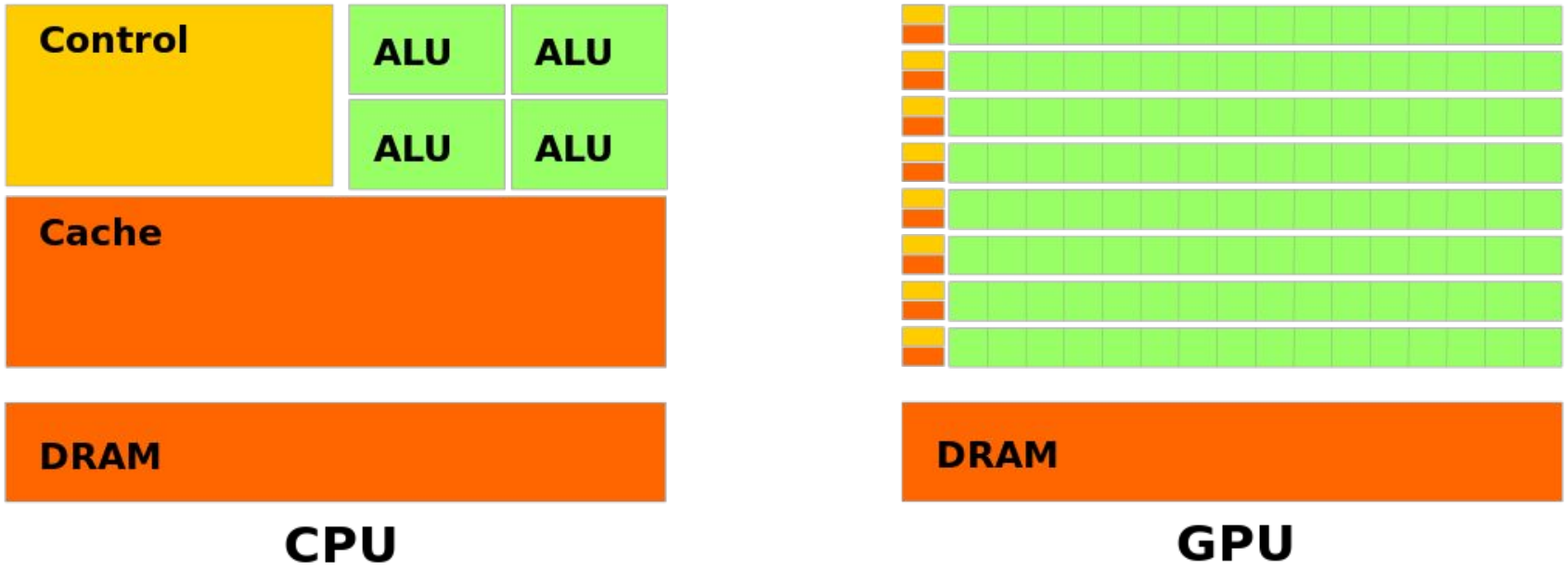
What is a GPU?

Calcul Québec

- A device for handling computationally expensive hot spots in your code (accelerator, coprocessor)
- Large number of low-powered, but low cost (monetary & power) processors
- Incredible computing speeds (teraflops) through massive parallelism (1000s of parallel threads or more)
- Heterogeneous computing: CPU and GPU work together on the problem

GPUs Under the Hood

Calcul Québec



ALU = Arithmetic and Logic Unit - The workhorse

Nvidia GPUs on Guillimin

Calcul Québec

- Nodes
  - 50 with Dual Intel Sandy Bridge EP E5-2670 (8-core, 2.6 GHz, 20MB Cache, 115W)
    - 25 with 64 GB RAM + 25 with 128 GB RAM
  - 8 with Dual Intel Ivy Bridge EP E5-2650v2 (8-core, 2.6 GHz, 20MB Cache, 115W)
    - 8 with 64 GB RAM
- Cards
  - 2 x nVidia Tesla Kepler K20 cards per node
  - Peak SP FP: 3.52 TFLops, Peak DP FP: 1.17 TFlops
  - 5GB memory

What was the ASCI Red?

- 1997, first teraflop supercomputer, with same compute power as a single K20 GPU
- 4,510 nodes (9298 processors), total:
  - 1,212 GB of RAM
  - 12.5 TB of disk storage
- 850 kW vs. 144 W for Nvidia K20

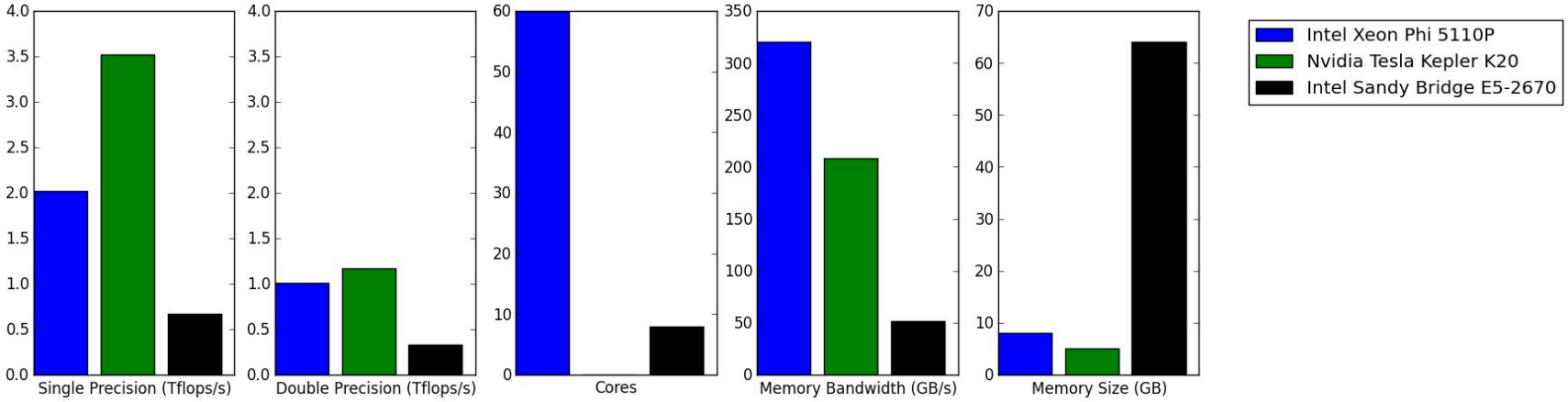
Notable Compute Canada GPU Resources

Calcul Québec

- Guillimin
  - 58 nodes * (2 * K20 + >64GB RAM)
- Helios
  - 15 nodes * (8 * K20 + 128GB RAM) (8 GPUs)
  - 6 nodes * (8 * K80 + 256GB RAM) (16 GPUs)
- Cedar (GP2)
  - 114 nodes * (4 * P100 + 128GB RAM)
  - 32 nodes * (4 * P100 same PCI root + 256GB RAM)
- Graham (GP3)
  - 160 nodes * (2 * P100 + 128GB RAM)

(Double Prec.) **K20**:1.17 Tflops, **K80**:1.8–2.9 Tflops, **P100**:4.3 Tflops

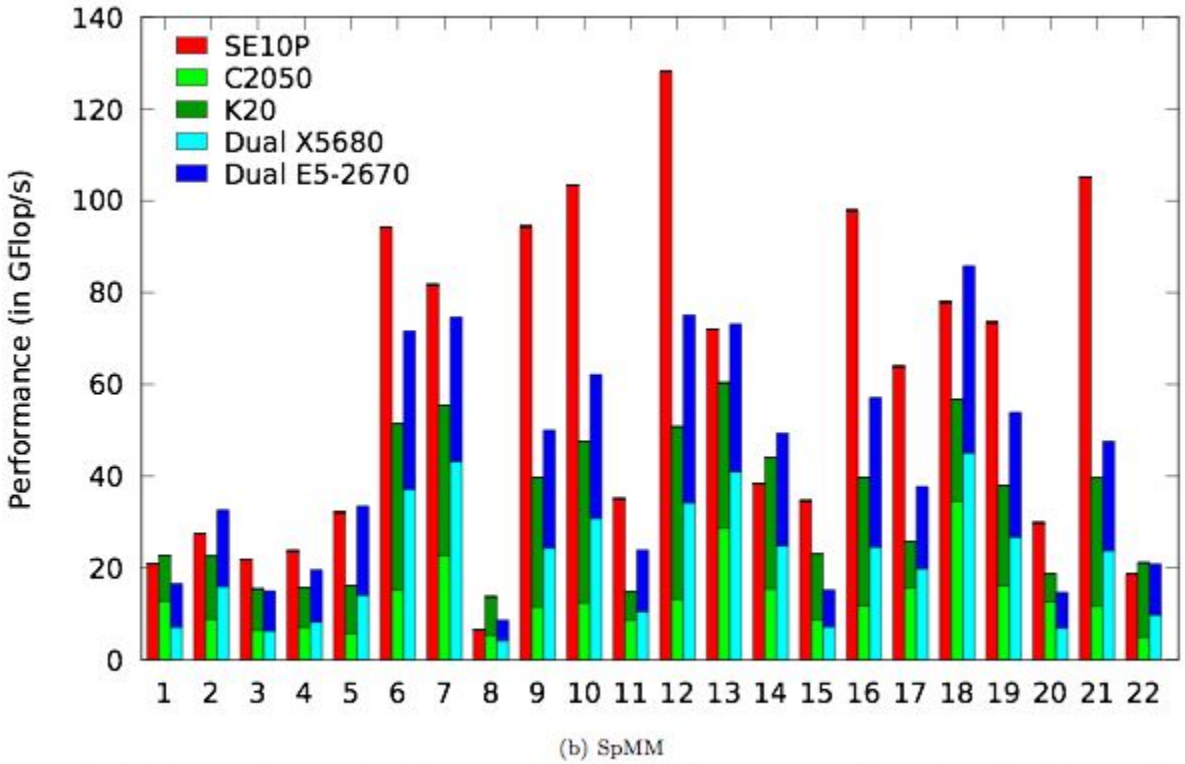
Comparisons



Notes:

- Chart denotes theoretical maximum values. Actual performance is application dependent
- The K20 GPU has 13 streaming multiprocessors (SMXs) with 2496 CUDA cores, not directly comparable to x86 cores
- The K20 GPU and Xeon Phi have GDDR5 memory, the Sandy Bridge has DDR3 memory

Benchmark Tests



(b) SpMM

Figure 10: Architectural comparison between a Intel Xeon Phi coprocessor (Pre-release SE10P), two NVIDIA GPUs (C2050 and K20) and two dual CPU architectures (Intel Xeon X5680 and Intel Xeon E5-2670).

Matrix multiplication results

Note: SE10P is a Xeon Phi Coprocessor with slightly higher specifications than the 5110P

Source: Saule et. al, 2013 - http://arxiv.org/abs/1302.1078

Benchmark Tests

Paths	Sequential	Sandy-Bridge CPU[1,2]	Xeon Phi[1,2]	Tesla GPU[2]
128K	13,062ms	694ms	603ms	146ms
256K	26,106ms	1,399ms	795ms	280ms
512K	52,223ms	2,771ms	1,200ms	543ms

[1] The Sandy-Bridge and Phi implementations make use of SIMD vector intrinsics.

[2] The MRG32K3a random generator from the cuRAND library (GPU) and MKL library (Sandy-Bridge/Phi) were used.

Embarrassingly parallel financial monte-carlo

Paths	Sequential	Sandy-Bridge CPU[1,2]	Xeon Phi[1,2]	Tesla GPU[2,3]
128K	4,234ms	89ms	231ms	110ms
256K	8,473ms	171ms	329ms	185ms
512K	17,192ms	339ms	608ms	343ms

[1] The Sandy-Bridge and Phi implementations make use of OpenMP and vectorization pragmas/attributes as much as possible.

[2] The MRG32K3a random generator from the cuRAND library (GPU) and MKL library (Sandy-Bridge/Phi) were used.

[3] Many small CUDA kernels need to be executed on the GPU, as parallelisation can only be done within each time step

Iterative financial monte-carlo with regression across all paths

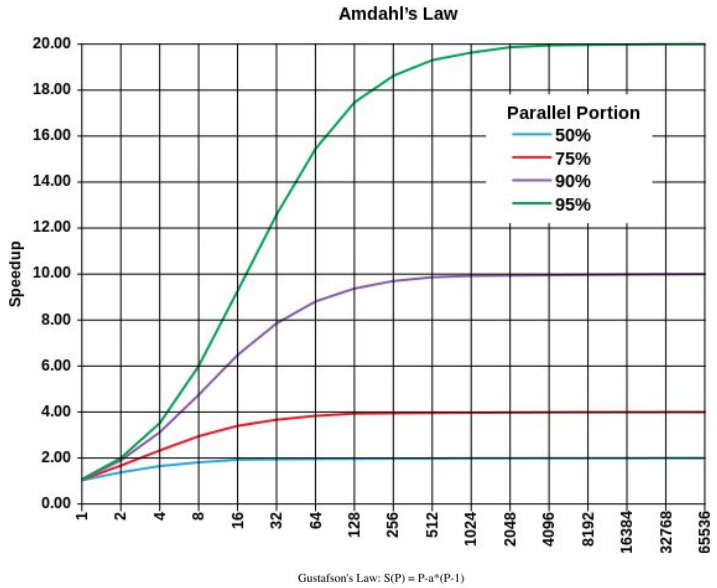
Note: the tested GPU is a K20X, which has slightly higher specifications than the K20

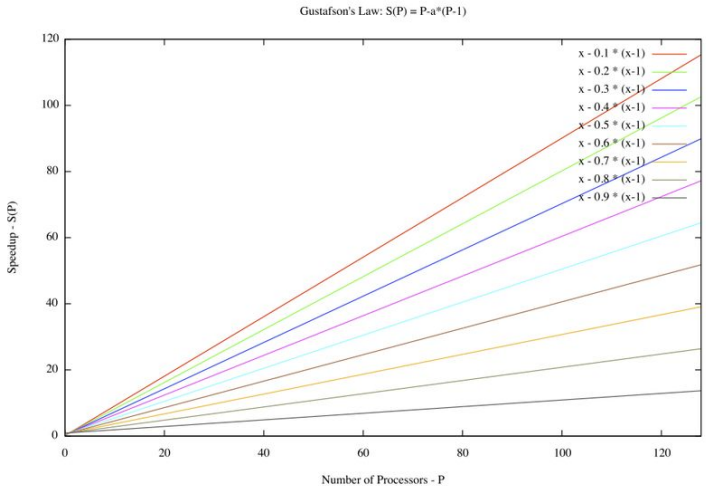
Source: xcelerit blog, Sept. 4, 2013, http://blog.xcelerit.com/intel-xeon-phi-vs-nvidia-tesla-gpu/

How can accelerators help you do science?

Calcul Québec

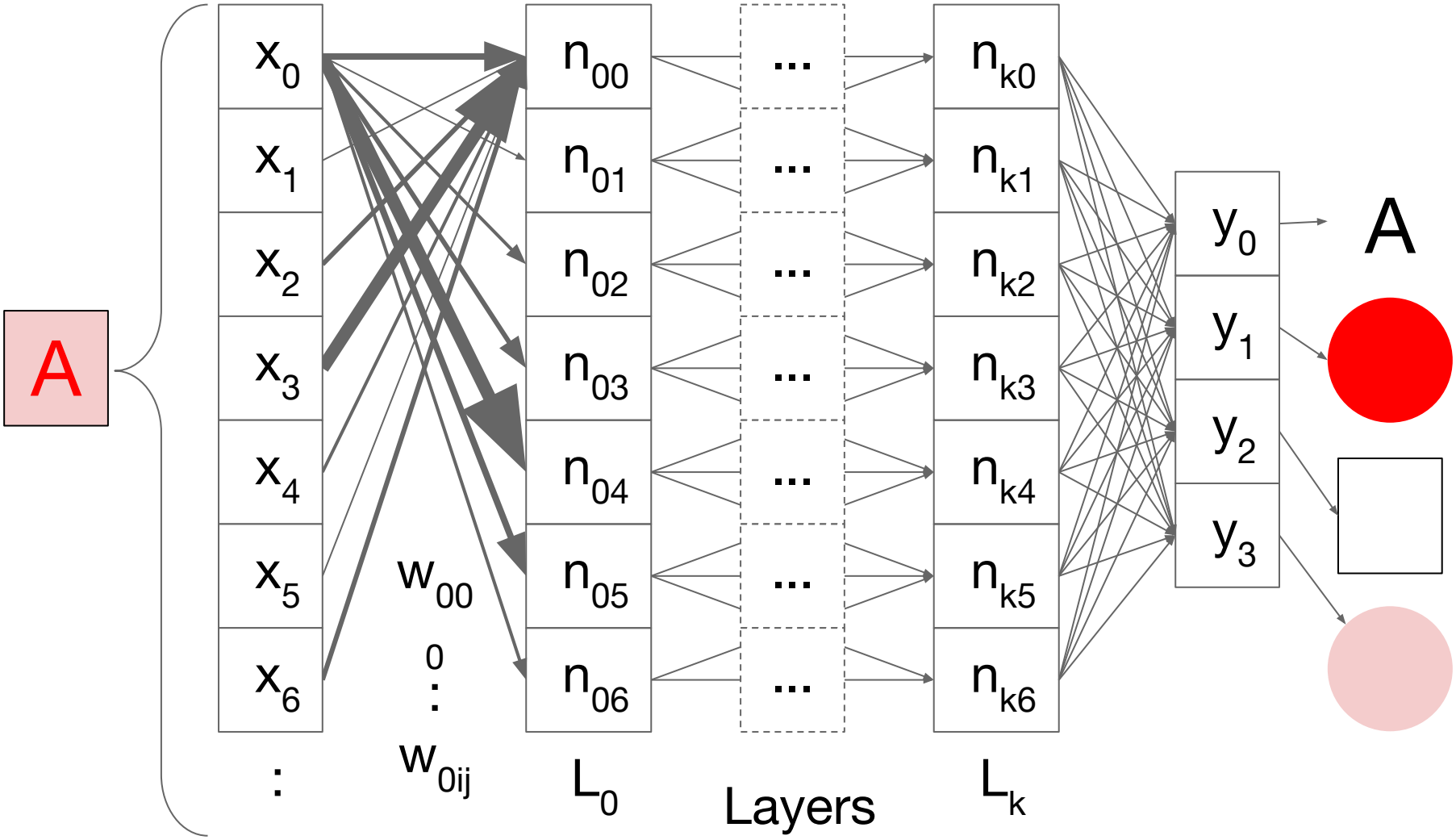
- Two ways of thinking about speedup from parallelism:
  - 1: Compute a fixed-size problem faster
    - Amdahl's law describes diminishing returns from adding more processors
  - 2: Choose larger problems in the time you have
    - Gustafson's law: Problem size can often scale linearly with number of processors





GPUs in Deep Learning

Calcul Québec



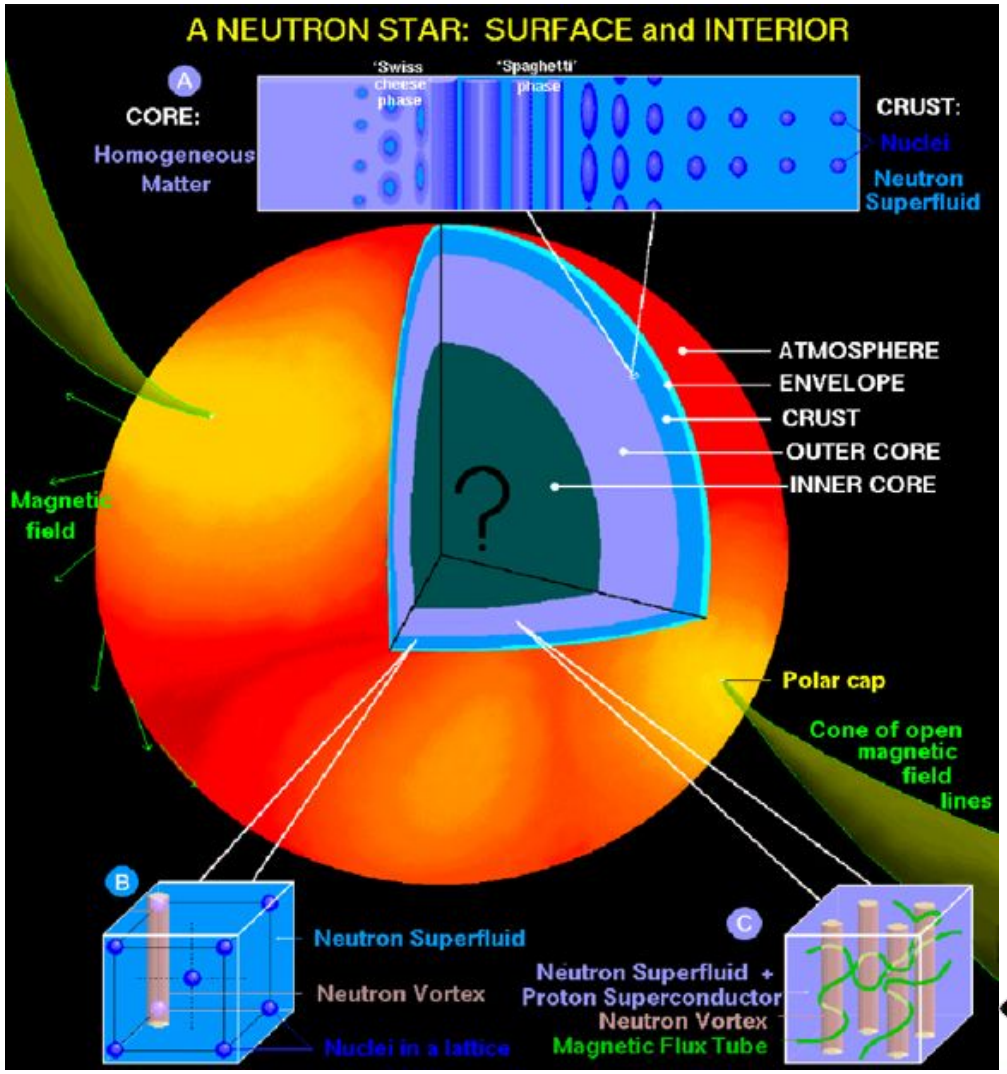
GPUs in Deep Learning
Backpropagation

Calcul Québec

1. Propagation
   a. Compute the output from a new input
   b. Compute intermediate (hidden) values and input values given a solution to the corresponding input
2. Weight update - for each weight:
   a. Evaluate the gradient (delta) of the weight given the difference between the forward and backward propagation
   b. Apply a fraction (percentage) of the gradient of the weight to the weight. The percentage influences how the network *learns*.

GPUs in Theoretical Astrophysics

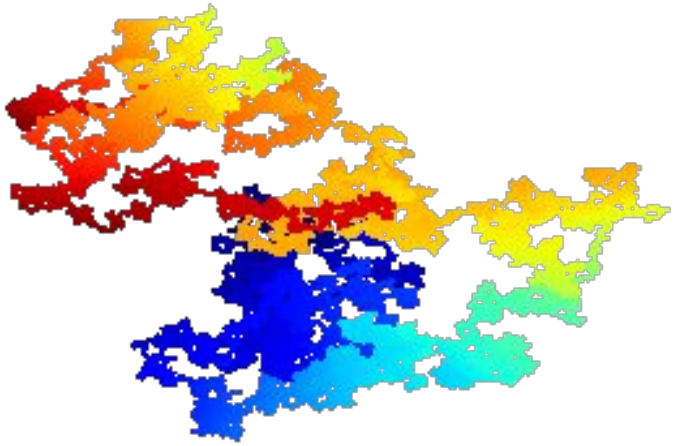
Calcul Québec



Worldline Numerics

Calcul Québec

- What is the non-local Casimir interaction between magnetic flux tubes?
- The energy is given by a path integral over all possible paths a virtual electron-positron pair can take through the flux tubes
- Approximate functional integrals with an average over an ensemble of representative worldlines



GPU Worldline Numerics

Calcul Québec

- Monte Carlo => error bars shrink with sqrt(N)
  - Want to compute thousands of paths or more
- Virtual pairs can interact with many flux tubes
  - Each path is a numerically intensive integral
- GPUs allow thousands of paths to be computed in parallel

Other Applications of GPUs

Calcul Québec

- Hundreds of scientific applications have GPU accelerated versions
- http://www.nvidia.ca/object/gpu-applications.html
- Chemistry, biology, physics, math, machine learning, weather and climate, CFD, finance, ...
- ABAQUS, Amber, GROMACS, LAMMPS, MATLAB, Theano

Ways to use GPUs

- Accelerated application (AMBER, GROMACS, LAMMPS, ABAQUS, MATLAB)
- Libraries (CUBLAS, CUFFT, Thrust)
- Programming directives (OpenACC)
- GPU programming (CUDA-C, CUDA Fortran, pyCUDA, OpenCL)

Increasing effort

Focus For Today

Calcul Québec

- No - GPU Accelerated Applications and Libraries
  - Incredibly useful for research, easy to use
  - Won't teach you much about GPUs
- Yes - Explicit GPU Programming
  - Will teach you about GPUs
  - Some depth in one language will help you to understand libraries, applications, directives, and other GPU languages
  - We will focus mainly on CUDA-C

GPU Programming Standards

Calcul Québec

- CUDA
  - Nvidia proprietary standard
  - Dependant on Nvidia hardware and software
  - Mature toolkit (debugging, profiling, etc.)
- OpenCL
  - Open Standard
  - Similar programming model to CUDA
- OpenMP 4.0 accelerator offloading
  - Open Standard
  - Higher-level, Pragma based
- OpenACC
  - Open Standard
  - Higher-level, Pragma based

Workshop Files

Calcul Québec

- Please copy the workshop files to your home directory

```
cp -R /software/workshop/gpu/* ~/
```

- Contains:
  - Code for the exercises
  - Submission scripts
  - Solutions to the exercises
  - Some GPU codes to explore

Scheduling GPU Jobs

Calcul Québec

- Submissions to Guillimin should specify `:gpus=1` or `:gpus=2` (per node)
- Workshop jobs run on a single CPU core + single GPU device
  - Example of an interactive job submission:

```
qsub -I -l nodes=1:ppn=1:gpus=1 \
  -l walltime=00:10:00
```

  - Example of a batch job submission:

```
qsub subTrivial.sh
```

Exercise 1: OpenACC

Calcul Québec

- OpenACC lets you easily offload segments of code to the GPU
- It uses pragmas, similar to OpenMP
- Our PGI OpenACC license is available through the **`pgi`** module:

```
module load pgi/14.9
```

- **`gcc`** is also working on OpenACC support (early stages of development)

Exercise 1: OpenACC

Calcul Québec

- Directives: the second easiest way to accelerate your code next to libraries
- Compile and run the C program **`matrix_mul`**:

```
pgcc -fast -o matrix_mul matrix_mul.c
```

```
./matrix_mul
```

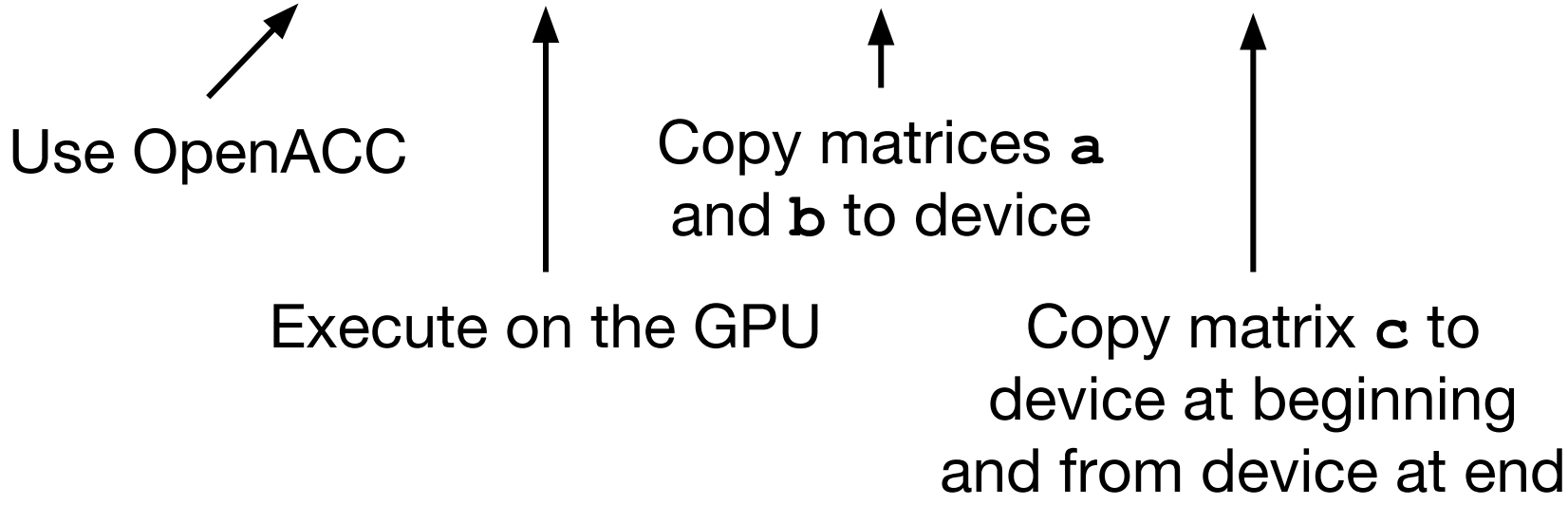
- The code multiplies two 4000x4000 float matrices using the CPU only. It takes about 20 seconds on our Sandy Bridge processors.

Exercise 1: OpenACC

Calcul Québec

- We can more than double the speed of this code with a single line and a special compiler:

```
#pragma acc kernels copyin(a,b) copy(c)
```



Exercise 1: OpenACC

Calcul Québec

- When the serial calculation has finished:
  - Do: **`cp matrix_mul.c matrix_mul_acc.c`**
  - Insert **`pragma`** immediately before the matrix multiplication **`for`** loops in **`matrix_mul_acc.c`**

```
#pragma acc kernels copyin(a,b) copy(c)
```

```
pgcc -acc -fast -Minfo \
  -o matrix_mul_acc matrix_mul_acc.c
./matrix_mul_acc
```

Exercise 1: OpenACC

Calcul Québec

- My results:
  - Serial CPU code: 21.29s
  - With OpenACC: 3.88s
  - Speedup of 5.5x
- What happened?
  - Compiler looked at our code and automatically parallelized it on the gpu device using advice from the **`pragma`**
  - Similar to OpenMP
- Speedup should increase with matrix size until device memory size becomes an issue

OpenACC Documentation

Calcul Québec

- OpenACC is very versatile and has many more features.
- Getting started guide:

```
/software/CentOS-6/compilers/pgi149/linux86-64/14.9/doc/openAcc_gs.pdf
```

First CUDA Code: trivial.cu

Calcul Québec

```
#include <stdio.h>


__global__ void foo()
{
}


int main()
{
    foo<<<1,1>>>();
    printf("CUDA error: %s\n",
        cudaGetErrorString(cudaGetLastError()));
    return 0;
}
```

Kernel (this one does nothing)

Kernel call (compared to function call in C). `<<<1,1>>>` indicates 1 block of 1 thread(s)

CUDA built-in functions (checks for errors, converts to string)

The rest is regular C code

Exercise 2: First CUDA Code

Calcul Québec

```
#include <stdio.h>

__global__ void foo()
{
}

int main()
{
    foo<<<1,1>>>();
    printf("CUDA error: %s\n",
        cudaGetErrorString(cudaGetLastError()));
    return 0;
}
```

Compile and run **trivial.cu**:

```
nvcc trivial.cu -o trivial

./trivial
trivial.out: CUDA error: no
error
```

NVCC Warning (if any)

Calcul Québec

- **`nvcc`** warning : The 'compute_10' and 'sm_10' architectures are deprecated, and may be removed in a future release.
- This warning is harmless. By default, **`nvcc`** compiles for older CUDA architectures
- Add the '**`-arch=compute_35`**' option to nvcc to compile specifically for the K20 architecture (compute capability 3.5)

Device Properties

Calcul Québec

```
cudaSetDevice(dev);


cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);


sprintf(msg, "  Total amount of global
memory: %.0f MBytes \n",
(float)deviceProp.totalGlobalMem /
1048576.0f);
```

Exercise 3: CUDA Samples

Calcul Québec

- Copy the CUDA samples directory to your home directory (warning: ~230MB)

```
cp -R $EBROOTCUDA/samples ./
```

- Contains:
  - Example programs (simple, utilities, graphics, imaging, finance, simulations, advanced, CUDA libraries)
  - Documentation
  - Tools

Exercise 3: CUDA Samples

Calcul Québec

- Run the deviceQuery program:

```
cd samples/1_Utilities/deviceQuery
make
./deviceQuery
cd -
```

- **Check**: CUDA capability, total memory, # CUDA cores, GPU clock rate, shared memory per block, warp size, threads per block (3D), blocks per grid (3D)

Simple CUDA Program

Calcul Québec

Serial CPU code

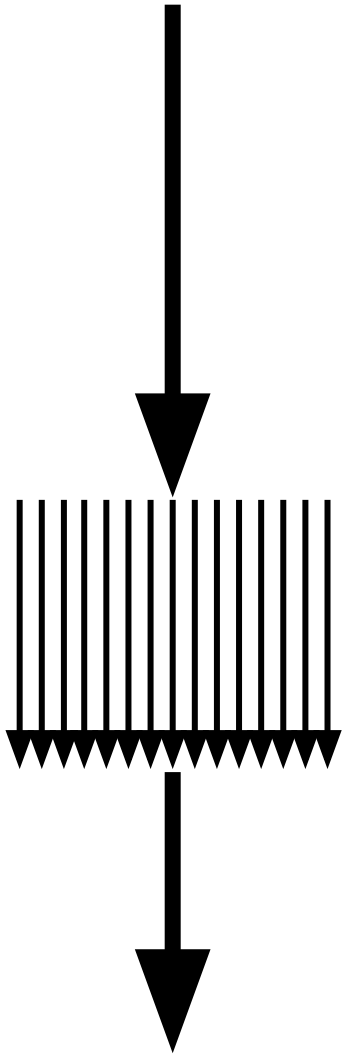
Allocate Memory on Device

Copy data from host to device

Perform parallel computations on data

Copy results from device to host

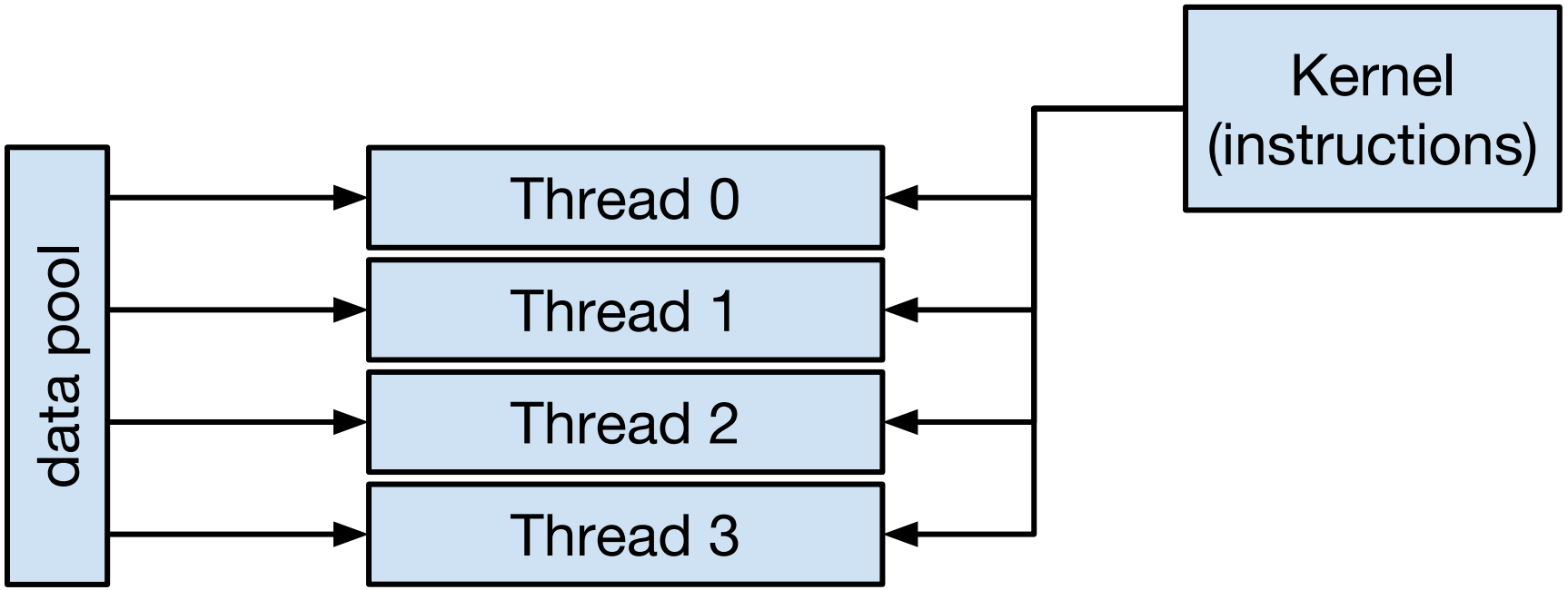
Free device memory



Simple CUDA Kernel

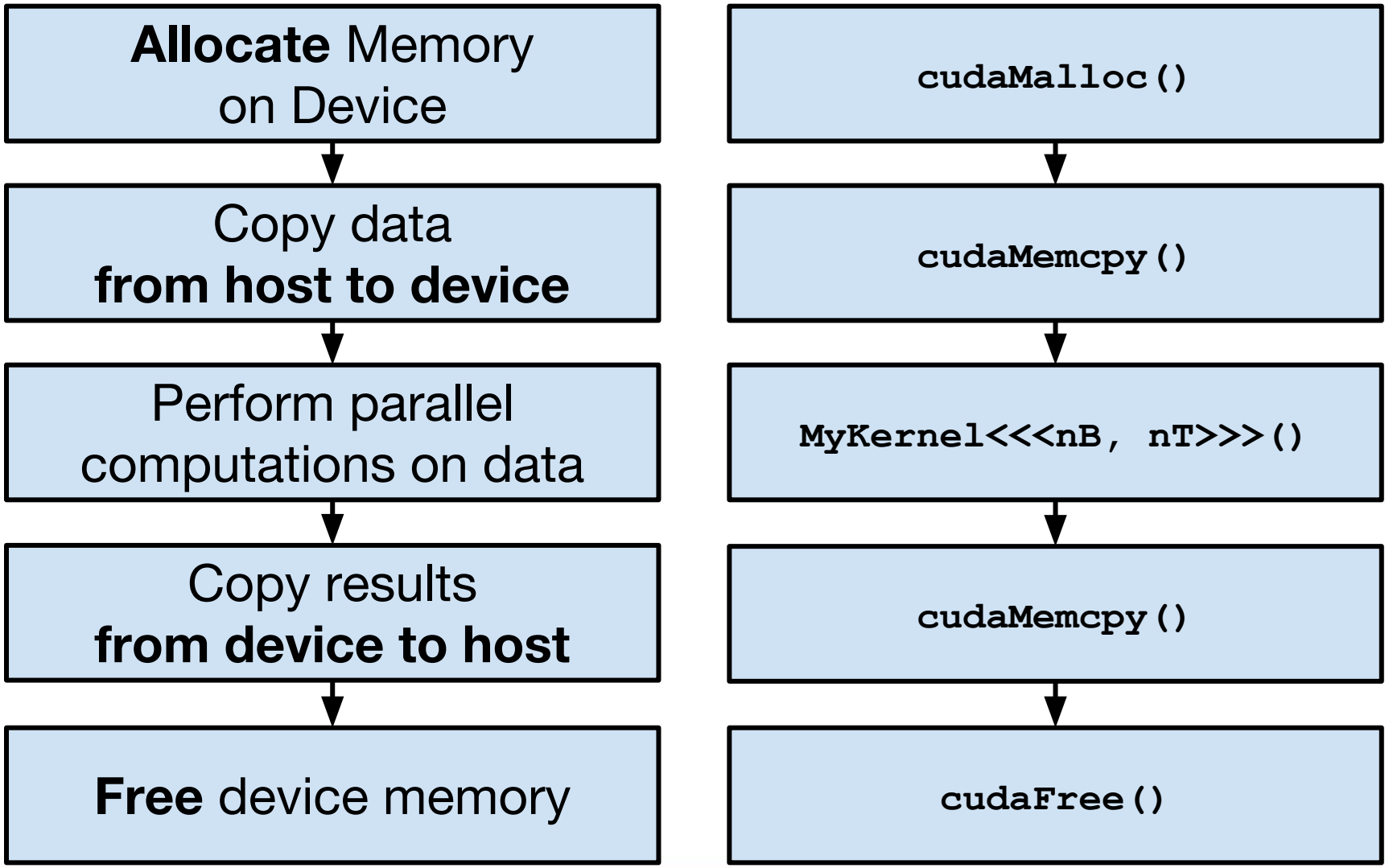
Calcul Québec

Perform parallel computations on data



"Single Instruction, Multiple Data" (SIMD) or
"Single Instruction, Multiple Thread" (SIMT) model

Simple CUDA Program



CUDA-C Reference Sheet

Calcul Québec

- Allocating/freeing memory
- Copying Data
- Kernels
- Error Checking
- Timing
- Unified Memory (CUDA 6, CC >= 3.0)

Question #2 - addnumbers.cu
What is the output?

Calcul Québec

```
__global__ void add2(int *a)
{
     *a = *a + 2;
}


int main( void )
{
     int *data_h, *data_d;
     cudaMalloc((void**)&data_d, sizeof(int));
     data_h = (int*)malloc(sizeof(int));


     *data_h = 5;
     cudaMemcpy( data_d, data_h, sizeof(int), cudaMemcpyHostToDevice );
     add2<<<1,1>>>(data_d);
     cudaMemcpy( data_h, data_d, sizeof(int), cudaMemcpyDeviceToHost );


     printf("data: %d\n", *data_h);
     free(data_h); cudaFree(data_d);
     return 0;
}
```

Question #3 - addnumbers.cu
What is the output?

Calcul Québec

```
__global__ void add2(int *a)
{
     *a = *a + 2;
}


int main( void )
{
     int *data_h, *data_d;
     cudaMalloc((void**)&data_d, sizeof(int));
     data_h = (int*)malloc(sizeof(int));


     *data_h = 5;
     //cudaMemcpy( data_d, data_h, sizeof(int), cudaMemcpyHostToDevice );
     add2<<<1,1>>>(data_d);
     cudaMemcpy( data_h, data_d, sizeof(int), cudaMemcpyDeviceToHost );


     printf("data: %d\n", *data_h);
     free(data_h); cudaFree(data_d);
     return 0;
}
```

Question #4 - addnumbers.cu
What is the output?

Calcul Québec

```
__global__ void add2(int *a)
{
     *a = *a + 2;
}


int main( void )
{
     int *data_h, *data_d;
     cudaMalloc((void**)&data_d, sizeof(int));
     data_h = (int*)malloc(sizeof(int));


     *data_h = 5;
     cudaMemcpy( data_d, data_h, sizeof(int), cudaMemcpyHostToDevice );
     add2<<<1,1>>>(data_d);
     cudaMemcpy( data_h, data_d, sizeof(int), cudaMemcpyDeviceToHost );


     printf("data: %d\n", *data_d);
     free(data_h); cudaFree(data_d);
     return 0;
}
```

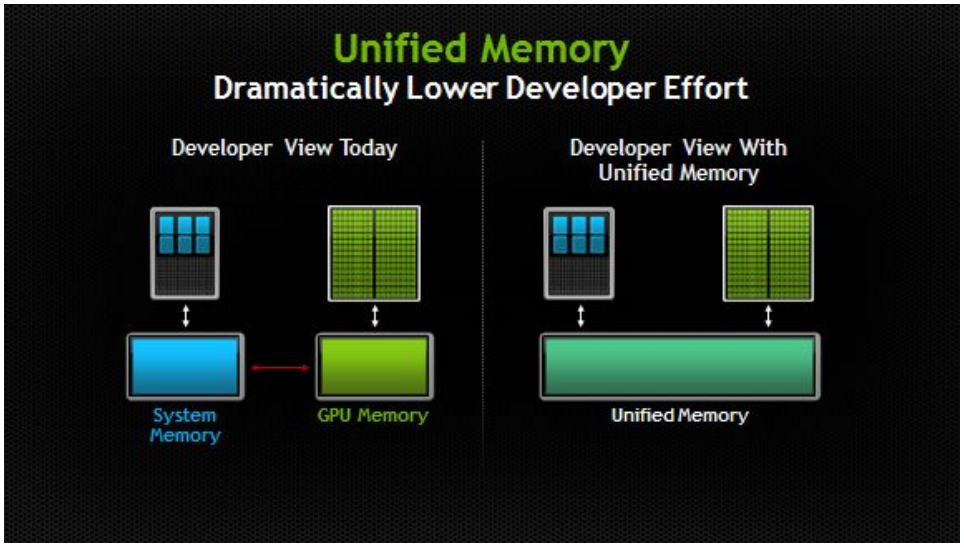
Important Points about Memory

Calcul Québec

- Device memory is different than host memory
  - Allocated with `cudaMalloc()`
  - Freed with `cudaFree()`
  - Device memory not accessible to host code
  - Host memory not accessible to device code
- Data is copied to and from the device using `cudaMemcpy()`
- Use `_d` and `_h` suffixes on variable names

Unified Memory

- Unified memory is new in CUDA 6
  - CUDA runtime manages memory transfers
  - Easier to use complex data structures
  - Allows pass by value and pass by reference
  - Note: only supported by compute capability >= 3.0 on 64-bit Linux or Windows




```
int *data;
cudaMallocManaged(&data,
                  sizeof(int));

*data = 3;
printf("data = %d\n", *data);
myKernel<<<1,1>>>(data);
cudaDeviceSynchronize();
printf("data = %d\n", *data);
cudaFree(data);
```

Unified Memory

- Here is the unified memory version of **`addnumbers.cu`**
  - See: **`addnumbers_unified.cu`**
- One data pointer instead of two
- No explicit cudaMemcpy calls

```
#include<stdio.h>

__global__ void add2(int *a)
{
    *a = *a + 2;
}

int main( void )
{
    int *data;
    cudaMallocManaged(&data,
        sizeof(int));
    *data = 5;
    add2<<<1,1>>>(data);
    cudaDeviceSynchronize();
    printf("data: %d\n",
        *data);
    cudaFree(data);
    return 0;
}
```

Using more than one thread

Calcul Québec

- Programmer defines the number of blocks and the number of threads per block
- Kernel is aware of the block ID, the block size, and the thread ID within the block

```
MyKernel<<<3, 6>>>()
```

block ID	0						1						2					
thread ID	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5

- **Question #5**: how to compute a unique index number?

Block and Thread Syntax

Calcul Québec

Thread ID within a block	`threadIdx.x`
Block size	`blockDim.x`
Block ID	`blockIdx.x`

idx = (thread ID) + (block size) * (block ID)

```
idx = threadIdx.x + blockDim.x * blockIdx.x
```

Q: Why the `.x`?

A: CUDA allows for (up to) 3-dimensional `threadIdx`'s and `blockIdx`'s: `threadIdx.{x,y,z}`, `blockIdx.{x,y,z}`

Exercise 4: CUDA Syntax

Calcul Québec

- Fill in the missing CUDA commands in the file **`matrixmul.cu`**
- Refer to your reference sheet for examples
- Choose a difficulty level depending on your experience level with C:
  - `matrixmul.cu`
  - **`matrixmul_med.cu`**
  - `matrixmul_adv.cu`
- (Optional) See CUDA fortran implementation, **`matrix_mul.cuf`**

```
pgfortran -fast -o matrix_mul matrix_mul.cuf
```

  - The **`.cuf`** extension or the **`-Mcuda`** compiler option indicate CUDA Fortran

Exercise 5: Write a kernel

Calcul Québec

- In the program **`integers.c`**, host code fills in an array with consecutive integers
- Modify this program so that the array is filled in parallel in a CUDA kernel function
- Don't forget:
  - The CUDA version should have extension **`.cu`**

    `cp integers.c integers.cu`
  - You may either use unified memory, or create separate host and device pointers
  - Copy results from the device to the host
- **`integers_adv.c`**: no hint in the code!

Exercise 6: Error Checking

Calcul Québec

- Compile and run **`errorcheck.cu`**. Notice that no errors are reported to the terminal window.
- Actually, there is more than one error in the program.
- Please use the error checking information on your CUDA reference sheet to add error checking to the program.
  - Advanced: Minimize the amount of added code and make the resulting code look clean
- Run the program and observe the CUDA errors present.
  - Solve any errors you discover and check that the output is 'data = 7' with no CUDA errors reported.
  - Discover the errors though CUDAs error reporting instead of through code inspection.

Exercise 7: Dot Product

Calcul Québec

- In small groups, plan a kernel function for computing a vector dot product (10 minutes, max)

$$C = a_0b_0 + a_1b_1 + a_2b_2 + \ldots + a_{N-1}b_{N-1}$$

- Pseudocode is fine
- Assume vector length is a power of two

```
dot<<<nBlocks, nThreads>>>(a_d, b_d, c_d);

__global__ void dot(float *a, float *b, float *c)
{
    …
}
```

One possible solution:

Calcul Québec

```
__global__ void dot(float *a, float *b, float *c)
{
    if (threadIdx.x + blockDim.x * blockIdx.x == 0)
    {
        for (int i = 0; i < N; i++)
        {
            *c += a[i] * b[i];
        }
    }
}
```

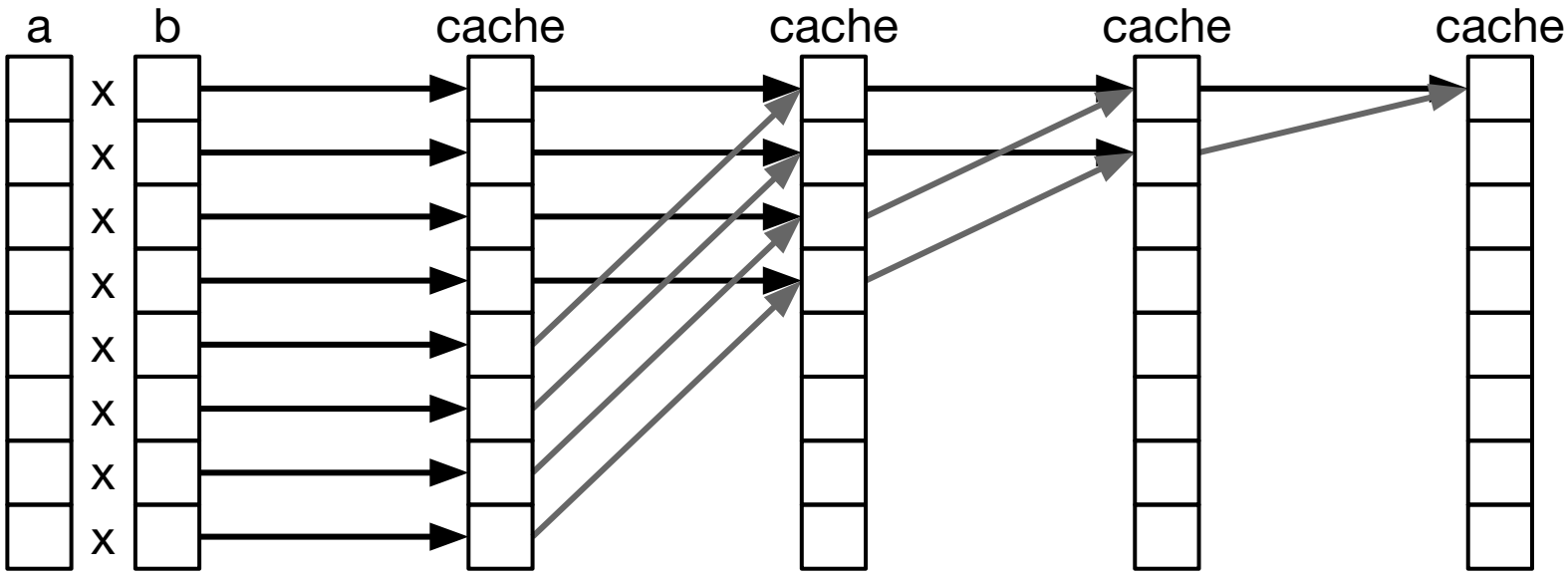
A 'correct' solution, but makes NO use of parallelism!

What problems do we encounter?

Calcul Québec

- How can we make good use of the threads and parallelism?
- How should we store intermediate values?
- How can we perform the sum?

Dot Product Strategy



Note: parallel reduction requires some cooperation between threads

Question #6 - Which memory type is the best choice?

Calcul Québec

- Want to store an array of $a_i * b_i$ that we can add through parallel reduction using many threads

Option	Description	Bandwidth	Accessible to...	Notes	Example
A)	Global Memory	Slow, high latency	All threads	same as cudaMalloc(), ~4.8GB	`__device__ float data[N];`
B)	Constant Memory	Slow, cached	All Threads	read-only	`__constant__ float data[N];`
C)	Shared Memory	150x faster than global	Threads in same block	Lifetime of block, ~50KB/block	`__shared__ float data[N];`
D)	Register Memory	Even faster, no latency	Local thread only	very limited resource, lifetime of thread	`float data[N];`

Let's use Shared Memory

Calcul Québec

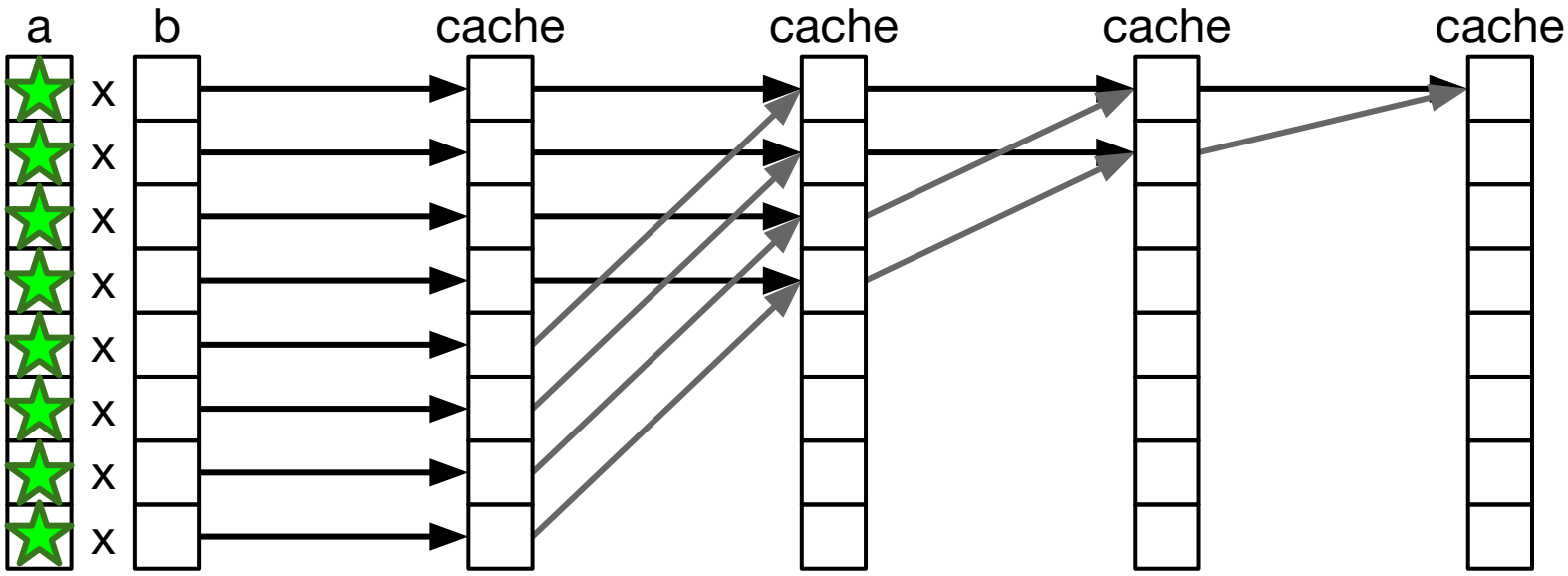
Pros:

- 150x faster than global memory
- Allows cooperation between threads (necessary for parallel reduction)

Cons:

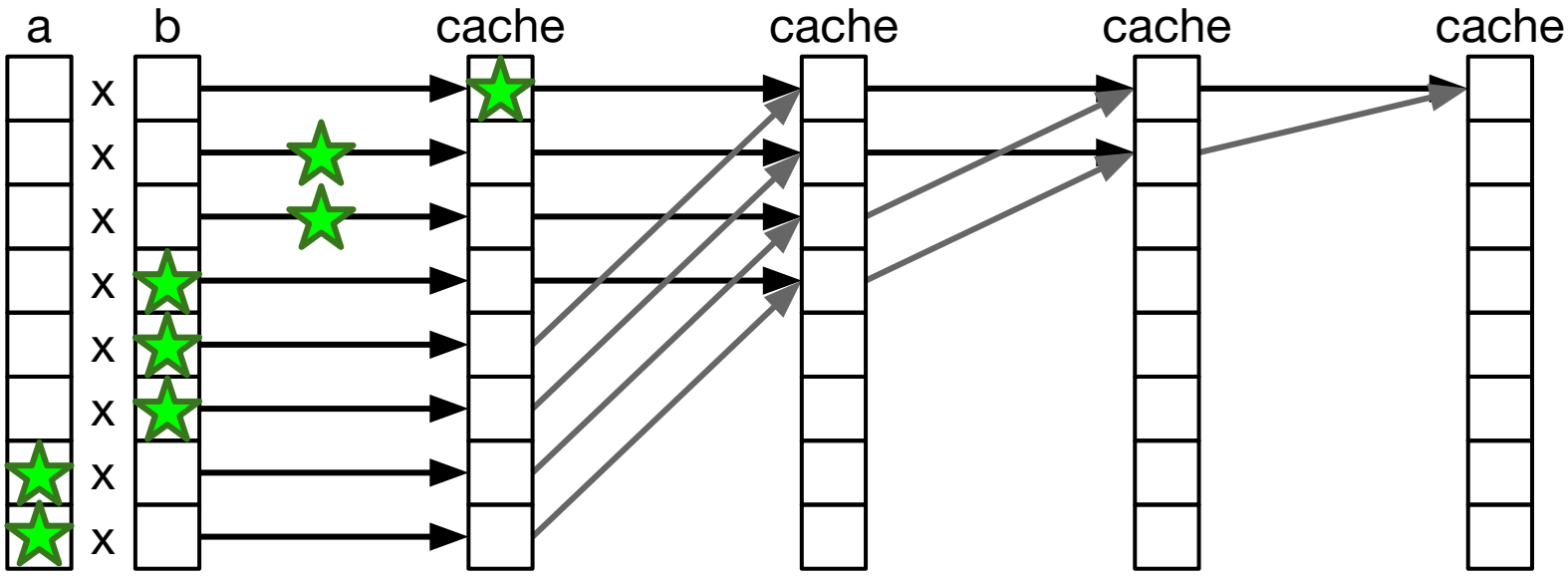
- Only ~50KB/block
- Only allows cooperation within a block (partial parallel reduction)
- Lifetime of the block

Thread Race



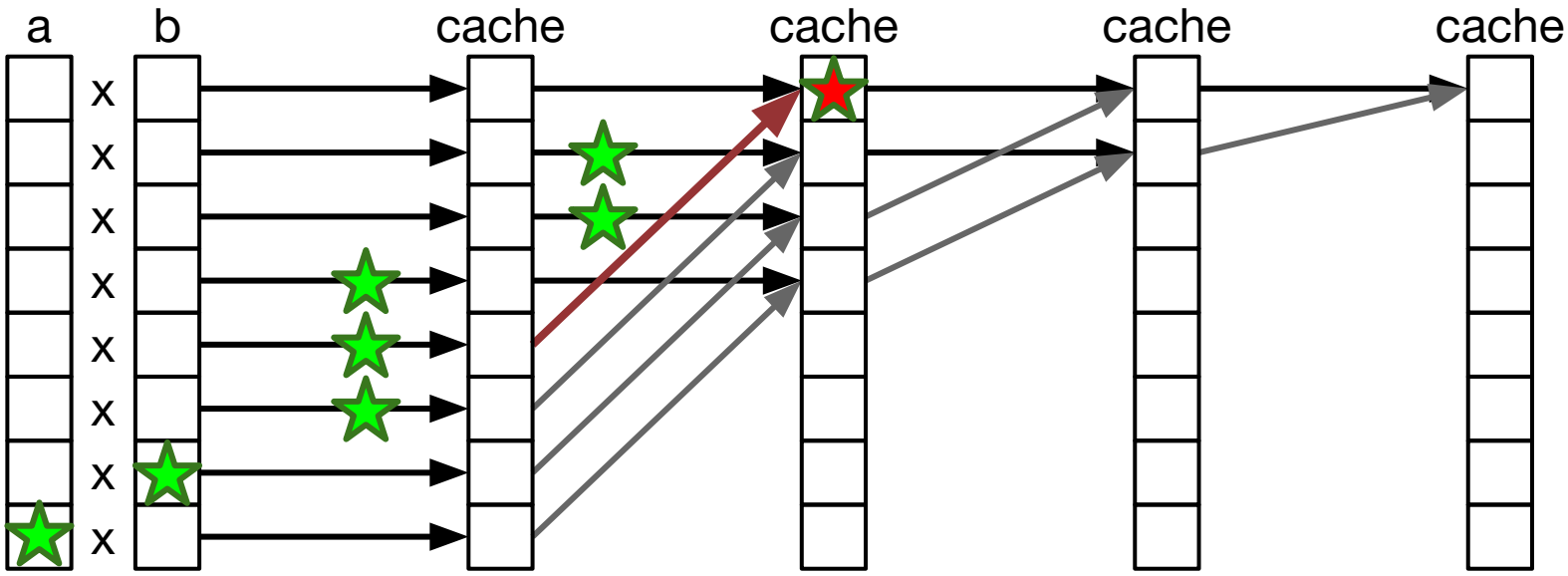
Thread 0 is the first one to cache its product

Thread Race



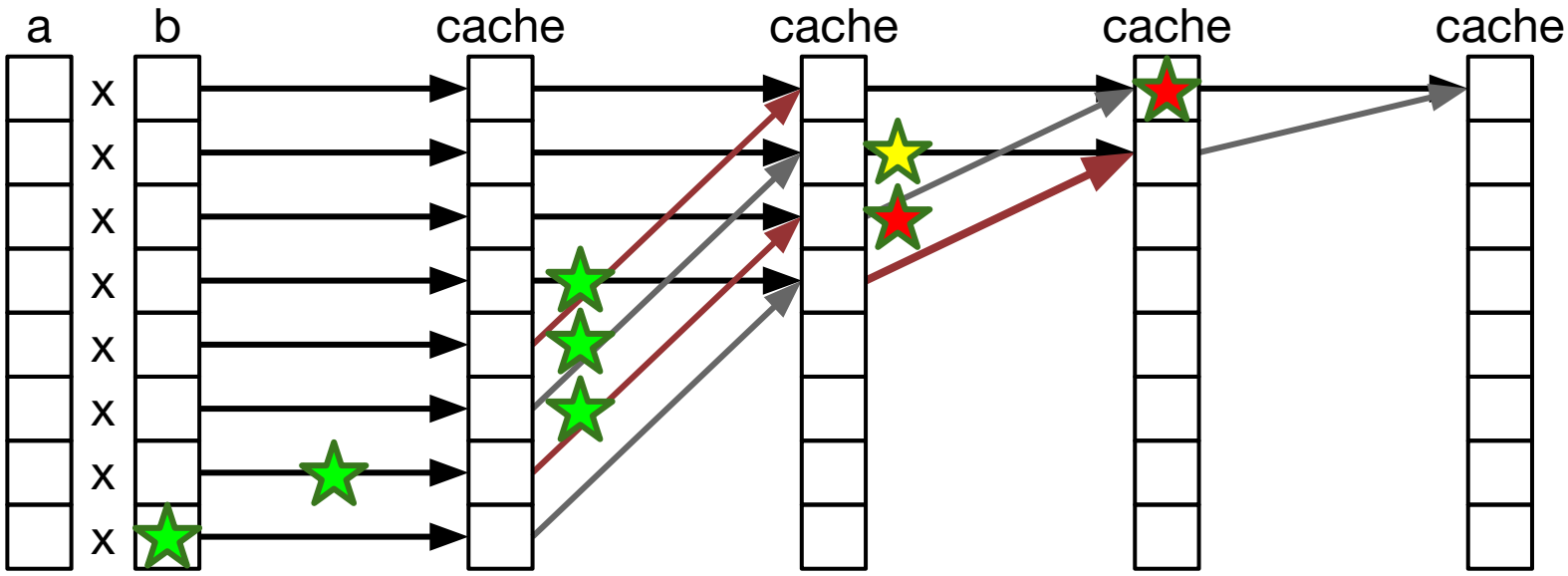
Thread 0 is the first one to cache its product

Thread Race



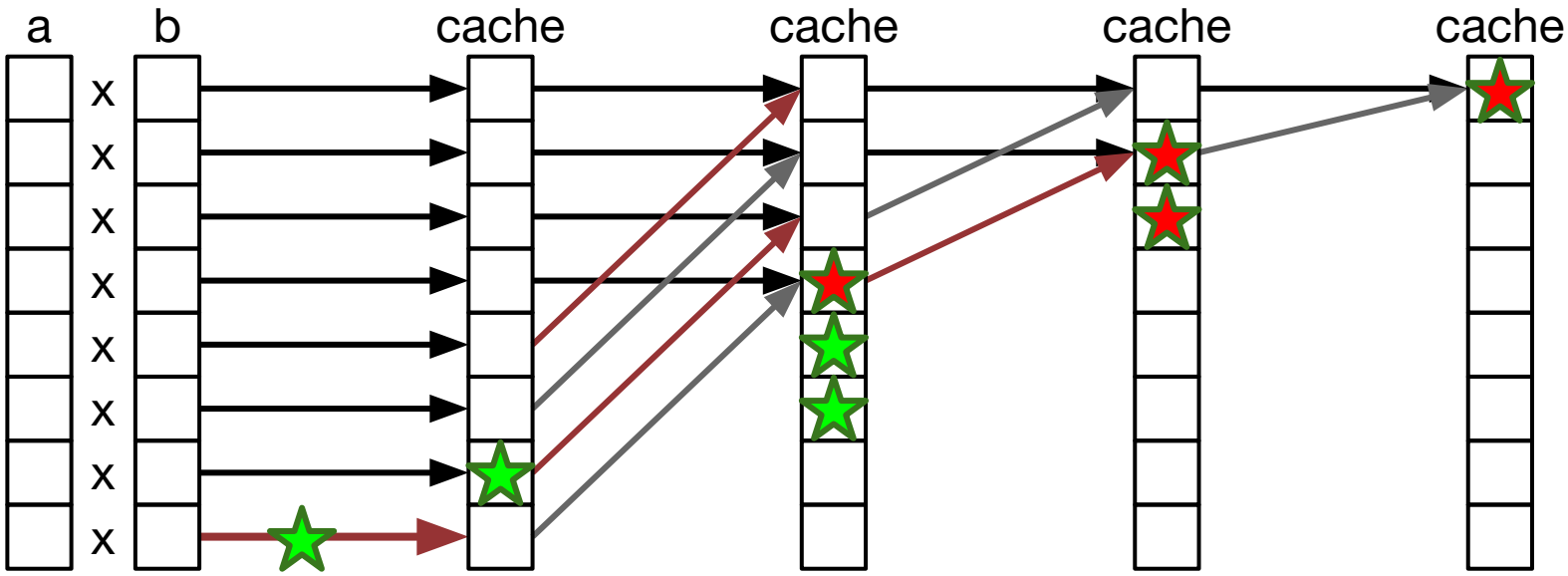
Thread 0 has used the value of thread 4, but this thread still hasn't cached its product! The result is undefined

Thread Race



Thread 1 is going to sum the value of thread 3,
but that value is not yet updated

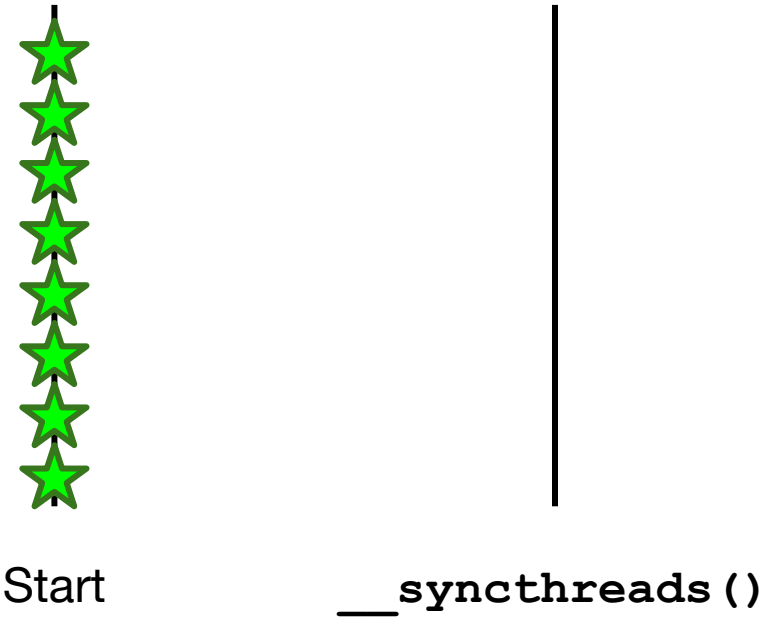
Thread Race



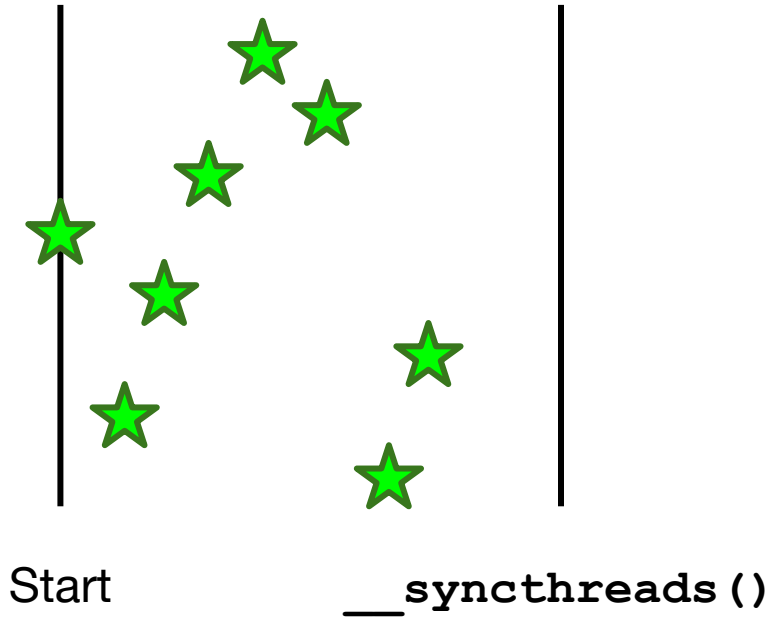
Thread 0 thinks we are done, but thread 7 still hasn't cached its product! The result is undefined

Our algorithm has 'race conditions'

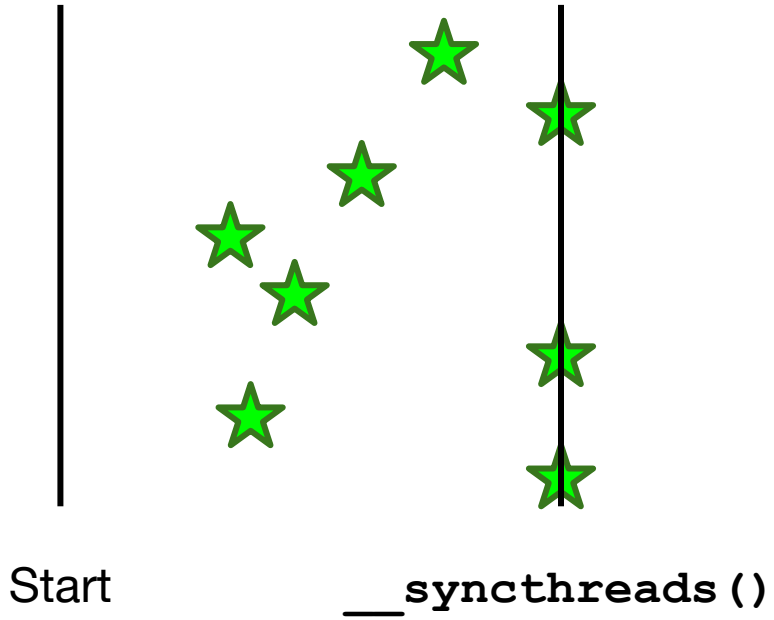
__syncthreads()



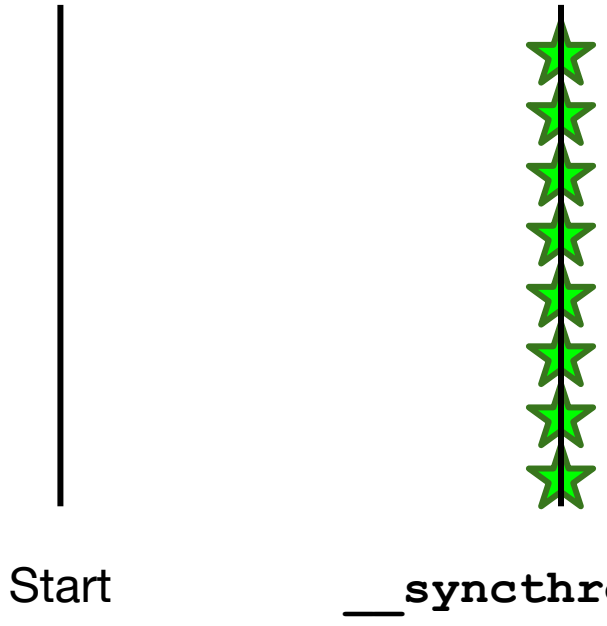
__syncthreads()



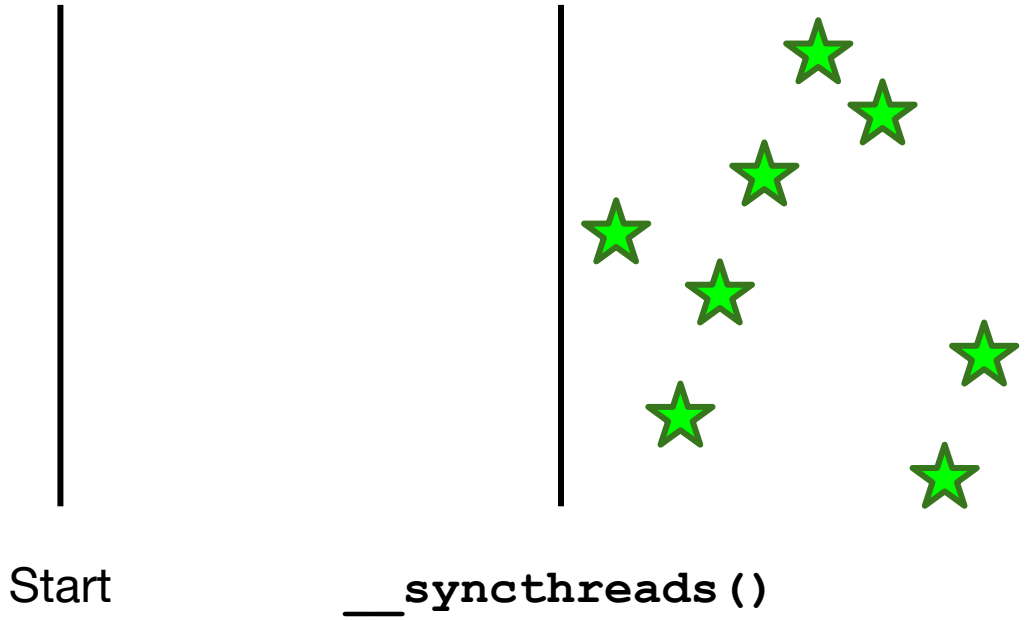
__syncthreads()



__syncthreads()



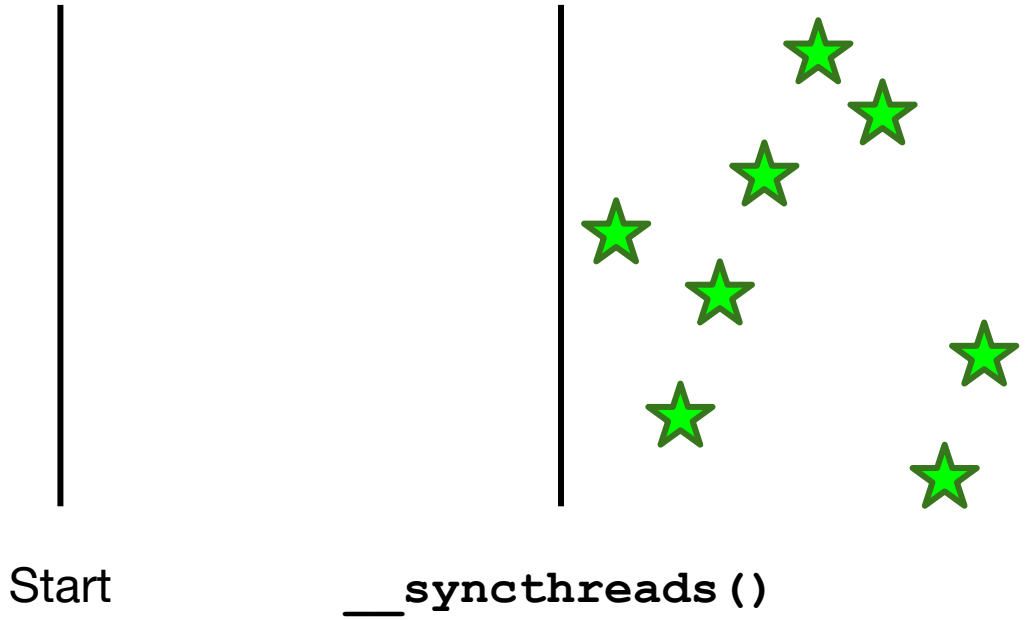
__syncthreads()



Start

`__syncthreads()`

__syncthreads()



Warning: only syncs threads within a block!

Question 7: Where should we syncthreads?

Calcul Québec

```
__global__ void dot( float *a, float *b, float *c ) {
  declarations (including cache)
  Compute products
  // __syncthreads()        // 1) Here?
  Store products into cache
  // __syncthreads()        // 2) Here?
  while (parallel reduction) {
    if (first half of cache) {
      add values from second half of cache
      // __syncthreads()    // 3) Here?
    } //end if
    // __syncthreads()      // 4) Here?
  } //end while
} //end kernel
```

__syncthreads() ensures that no threads in a block advance until they all reach the barrier

Q. 8 - Which is the correct way to store products in the cache?

Calcul Québec

```
__global__ void dot( float *a, float *b, float *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

A	`__shared__ float cache[threadsPerGrid];` `cache[tid] = a[tid] * b[tid];`
B	`__shared__ float cache[threadsPerBlock];` `cache[threadIdx.x] = a[tid] * b[tid];`
C	`__shared__ float cache[threadsPerBlock];` `cache[blockIdx.x] = a[tid] * b[tid];`
D	`__shared__ float cache[blocksPerGrid];` `cache[threadIdx.x] = a[tid] * b[tid];`
E	`__shared__ float cache[blocksPerGrid];` `cache[blockIdx.x] = a[tid] * b[tid];`

```
    c[blockIdx.x] = parallelReduceSum(cache);
}
```

Shared Memory Example

Calcul Québec

- You will implement the dot product kernel shortly
- First, consider an example: reversing an array
- At first, we will assume there is only one block to avoid algebra problems

Shared Memory Example

Here is the kernel using global memory with no shared memory cache:

```
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    int idx = threadIdx.x;

    int outidx = (blockDim.x - 1 - idx);

    d_out[outidx] = d_in[idx];
}
```

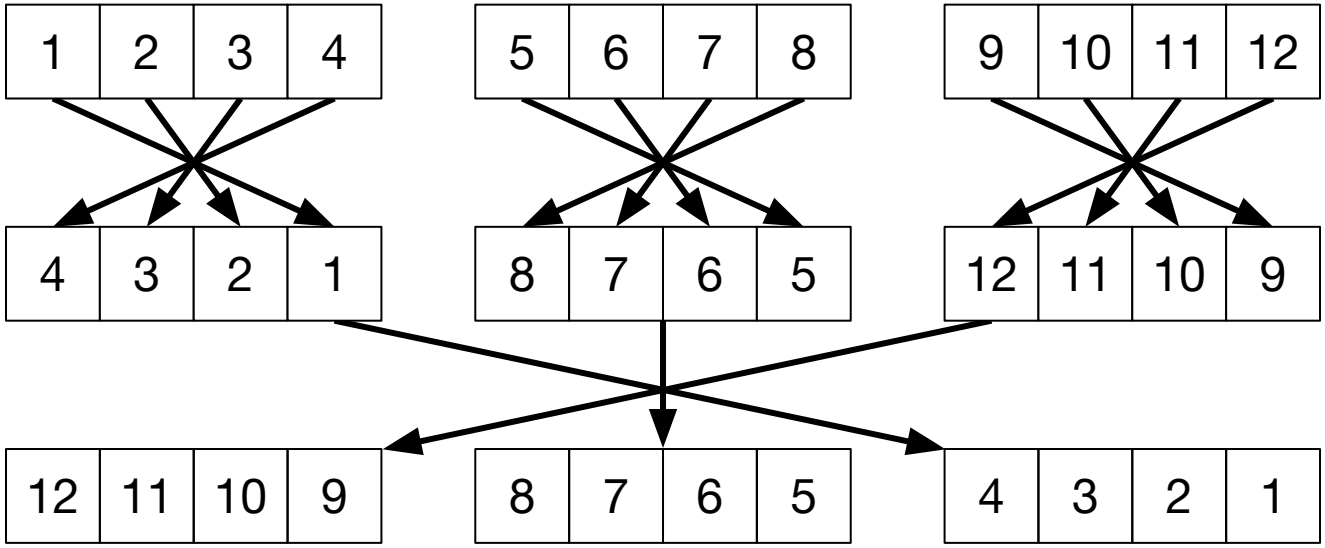
Shared Memory Example

Calcul Québec

```
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    __shared__ int s_data[blockDim.x];

    int idx  = threadIdx.x;

    // Load one element per thread and store it
    // *in reversed order* into temporary shared memory
    int outidx = (blockDim.x - 1 - idx);
    s_data[outidx] = d_in[idx];

    // Block until all threads in the block have
    // written their data to shared mem
    __syncthreads();

    // write the data from shared memory in forward order,
    // but to the reversed block offset as before
    d_out[idx] = s_data[idx];
}
```

Shared Memory Example

- This example illustrates using shared memory in one block
- Normally, there are many blocks



Reverse within blocks using shared mem.

AND

Reverse the blocks using global mem.

Shared Memory Example

Calcul Québec

```
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    __shared__ int s_data[blockDim.x];

    int inOffset  = blockDim.x * blockIdx.x;
    int idx  = inOffset + threadIdx.x;

    // Load one element per thread and store it
    // *in reversed order* into temporary shared memory
    s_data[blockDim.x - 1 - threadIdx.x] = d_in[idx];

    // Block until all threads in the block have
    // written their data to shared mem
    __syncthreads();

    // write the data from shared memory to global,
    // reversing the order of the blocks
    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
    int out = outOffset + threadIdx.x;
    d_out[out] = s_data[threadIdx.x];
}
```

DotProduct Pseudocode

Calcul Québec

```
__global__ void dotproduct(float *a, float *b, float *c, int N) {
    //Declare shared memory cache

    //Compute global thread index, idx
    //Cache a[idx]*b[idx] into shared memory

    //Synchronize threads

    for (int offset = block_size / 2 ; offset > 0 ; offset /= 2) {
        if (thread_id < offset) {
            cache[thread_id] += cache[offset + thread_id];
        }

        //Synchronize threads
    }

    if (thread_id == 0) {
        //Store reduction result in c[block_id]
    }
}
```

Exercise 8: Implement dot product kernel

Calcul Québec

- The program **`dotproduct.cu`** is missing code from the kernel function
  - Please complete the kernel
- Don't forget:
  - Use shared memory for the cache (may start with global and add shared after)
  - Use `__syncthreads()` where necessary
  - Use parallel reduction to sum all products associated with the current block

Choosing number of threads per block

Calcul Québec

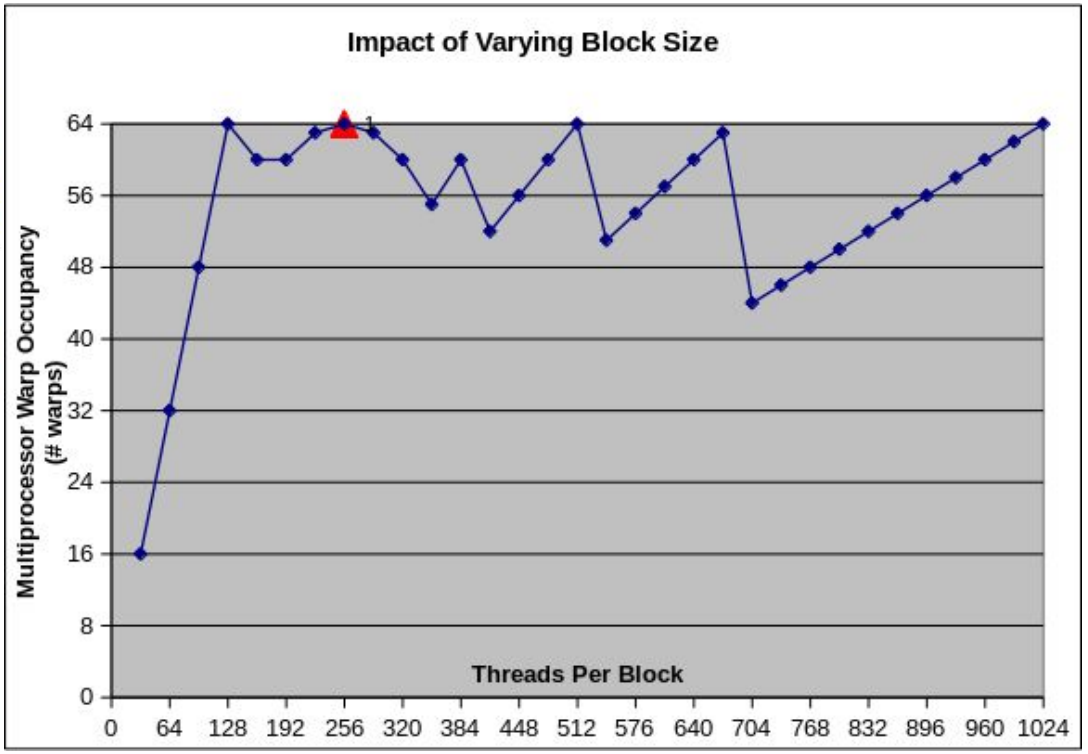
- The resources used per kernel and the limitations of your device determine the concurrency: number of threads that can run simultaneously
  - Threads per block
  - Registers per thread
  - Shared memory per block
- Use the `--ptxas-options=-v` nvcc option to get a report
- Nvidia provides a spreadsheet for planning your kernels to achieve high occupancy
  - CUDA GPU Occupancy Calculator

CUDA Occupancy Calculator

Calcul Québec

```
$ nvcc -o dotprod dotprod_soln.cu --ptxas-options=-v
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z3dotPfS_S_' for 'sm_20'
ptxas info    : Function properties for _Z3dotPfS_S_
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 11 registers, 1024 bytes smem, 56 bytes cmem[0]
```

4	Just follow steps 1, 2, and 3 below! (or click here for help)	
6	1.) Select Compute Capability (click):	3.5
7	1.b) Select Shared Memory Size Config (bytes)	49152
9	2.) Enter your resource usage:	
10	Threads Per Block	256
11	Registers Per Thread	11
12	Shared Memory Per Block (bytes)	1024
14	(Don't edit anything below this line)	
16	3.) GPU Occupancy Data is displayed here and in the graphs:	
17	Active Threads per Multiprocessor	2048
18	Active Warps per Multiprocessor	64
19	Active Thread Blocks per Multiprocessor	8
20	Occupancy of each Multiprocessor	100%
23	Physical Limits for GPU Compute Capability:	3.5
24	Threads per Warp	32
25	Warps per Multiprocessor	64
26	Threads per Multiprocessor	2048
27	Thread Blocks per Multiprocessor	16
28	Total # of 32-bit registers per Multiprocessor	65536
29	Register allocation unit size	256
30	Register allocation granularity	warp
31	Registers per Thread	255
32	Shared Memory per Multiprocessor (bytes)	49152
33	Shared Memory Allocation unit size	256
34	Warp allocation granularity	4
35	Maximum Thread Block Size	1024



Identifying GPU Algorithms

Calcul Québec

- SIMD Parallelizability
  - Number of concurrent threads (need 1000s)
  - Minimize conditionals and divergences
- Operations performed per datum transferred to device (FLOPs/GB)
  - Data transfer is overhead
  - Keep data on device and reuse it

Question 9 - GPU performance boost?

Calcul Québec

Which algorithm gives the most GPU performance boost?

Put the following in order from least work per datum to most:

1. matrix-vector multiplication
2. matrix-matrix multiplication
3. matrix trace (sum of diagonal elements)

Big data case study: K-means clustering

Calcul Québec

- GPUs have limited memory resources
  - 5GB global memory
  - 64kB constant memory
  - 48kB shared memory / block
- For data size > global memory
  - Can still achieve performance boost with GPUs
  - Good performance requires advanced techniques to achieve GPU computation + CPU computation + PCIe memory transfers to occur simultaneously
  - This discussion is to show what is possible, not a learning-goal of this workshop
- Case study based on Ren Wu, Bin Zhang, Meichun Hsu, “GPU-Accelerated Large Scale Analytics”, 2009, Hewlett-Packard Laboratories
  - K-means clustering

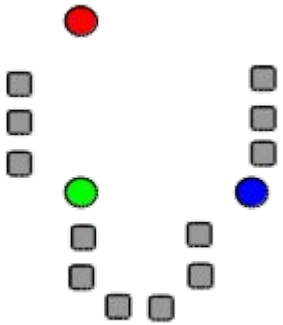
K-means clustering

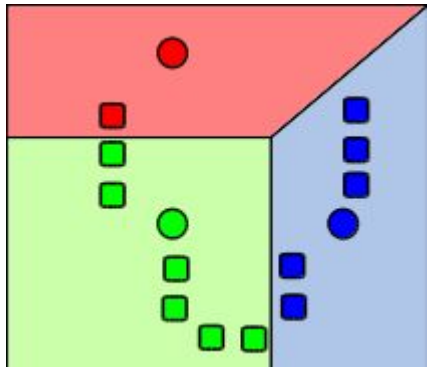
Calcul Québec

- Unsupervised machine learning
- Divide a data set into k different categories based on the features of that data set
- E.g. Clothing manufacturer: based on customer's height and weight data, divide them into 3 or more size categories

K-means clustering

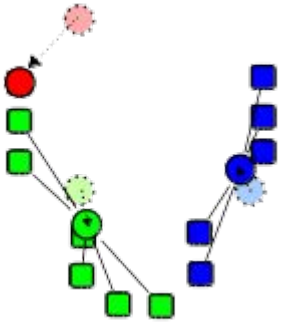
Calcul Québec

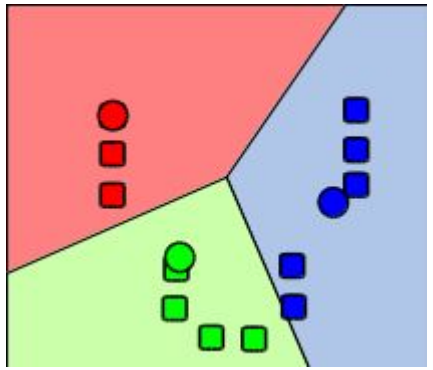




Step 1: Randomly generate K locations (circles)

Step 2: Group data points by proximity to locations





Step 3: Update locations to the centroid of each group

Iterate over steps 2 and 3

Images: I, Weston.pace

“Small data” GPU implementation

Calcul Québec

- Computational hotspot: computing distances between each cluster centroid and each data point, O(n*k)
- Parallelization: Each thread loops over cluster centroids and computes the distances for a single data point
- Memory management:
  - Copy transposed data set to device (Must be transposed to get coalesced reads for memory bandwidth)
  - Copy cluster centroids to constant memory
  - Compute distances and assign clusters in parallel
  - Copy cluster assignments back to host

Big data challenges

- What if data does not fit into global memory?
  - Can the problem be solved in smaller pieces?
  - If so, we must transfer data for each piece over PCIe.
    - How to manage all of the data transfers?
  - When and how to transpose?

CUDA Streams and Data Partitioning

Calcul Québec

- Suppose we need to process the data in separate chunks because of memory limitations
- The GPU has a mechanism for task-based parallelism (in addition to the data parallelism we have worked with today)
  - CUDA Streams: A queue of events to be executed in order on a device
- By using two or more CUDA streams, we can perform the memory copies in parallel with kernel executions
- Can transpose our chunks directly on the device using a kernel

CUDA Streams

Calcul Québec

Stream 0	Stream 1
Copy chunk of data to device (memcpyAsync)	
Transpose chunk of data on device (kernel)	Copy chunk of data to device (memcpyAsync)
Compute distances, assign clusters (kernel)	Transpose chunk of data on device (kernel)
Copy cluster assignments from device (memcpyAsync)	Compute distances, assign clusters (kernel)
	Copy cluster assignments from device (memcpyAsync)

Time

K-means GPU speedup

Calcul Québec

- Dual quad-core Intel Xeon 5345 2.33 GHz CPUs
- Nvidia GeForce GTX 280 GPU (1GB memory)
- The paper reports for big data problems
  - 200x-400x faster than single-core CPU
  - 20x-40x faster than 8-core CPU version

Review

- We learned how to:
  - Use OpenACC pragmas to easily accelerate loops
  - Allocate and free device memory
  - Copy data between host and device
  - Implement programs with SIMD parallelism
  - Identify and fix errors based on CUDA error messages
  - use shared memory for cooperation between threads
  - Compile CUDA programs using nvcc
  - Identify whether an algorithm is a good candidate for ‘easy’ performance gains on a GPU
  - Recognize some advanced techniques for big data processing

Keep Learning...

Calcul Québec

- Documentation:
  - http://docs.nvidia.com
- Tutorials:
  - http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/207200659
  - https://developer.nvidia.com/cuda/cuda-education-training
  - https://developer.nvidia.com/cuda-training
  - https://nvidia.qwiklab.com/ (interactive iPython notebooks on AWS)
- Examples:
  - http://developer.nvidia.com/cuda/cuda-downloads
- Courses:
  - http://code.google.com/p/stanford-cs193g-sp2010/
  - https://www.coursera.org/course/hetero
- Questions:
  - http://stackoverflow.com
  - https://forums.geforce.com/

What Questions Do You Have?

guillimin@calculquebec.ca
support@calculquebec.ca