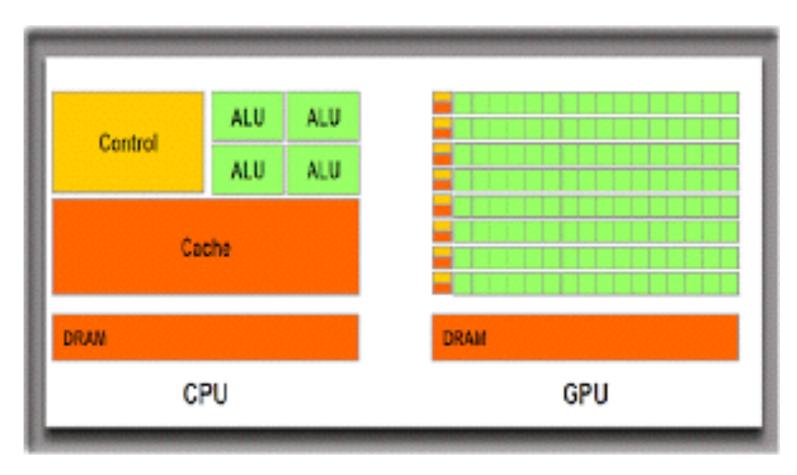
Introduction to Cuda

Olivier Mattelaer UCLouvain CP3 & CISM

Program of this lecture

- Difference between CPU and GPU
 - ➡ Why and when to use a GPU?
- What is CUDA?
 - ➡ When can I use cuda?
- Structure of a GPU program
 - Nomenclature
- First example of CUDA programming
- First step in optimisation of a CUDA program
 - Managing memory transfer

CPU versus GPU



CPU	GPU
Central Processing Unit	Graphics Processing Unit
Several cores	Many cores
Low latency	High throughput
Good for serial processing	Good for parallel processing
Can do a handful of operations at once	Can do thousands of operations at once

Speed versus Latency

- Speed: number of operation per second
- Latency: delay in the first operation
 - $\Rightarrow T = L + vD$
- How amazon transfer data from one cluster to another

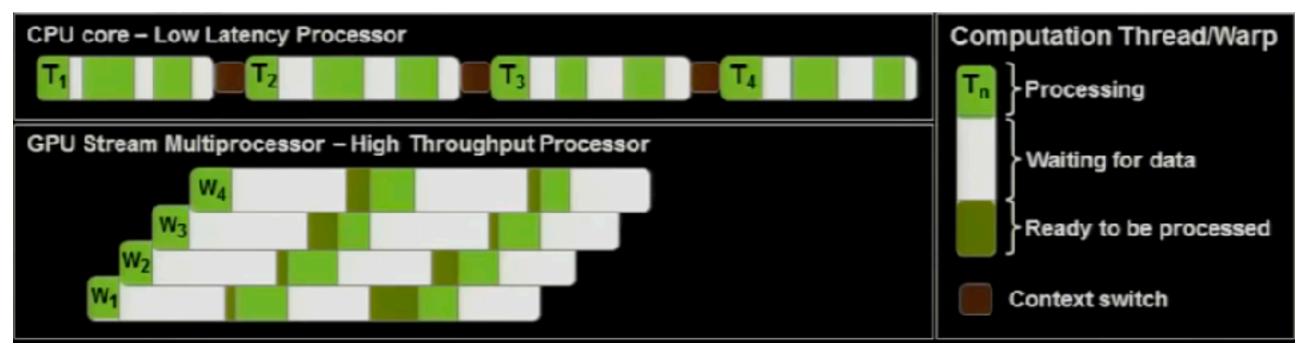


- Speed: Large bandwidth
 - Fiber connection: Gb
- Latency: time of the travel between the two cluster.



GPU versus CPU

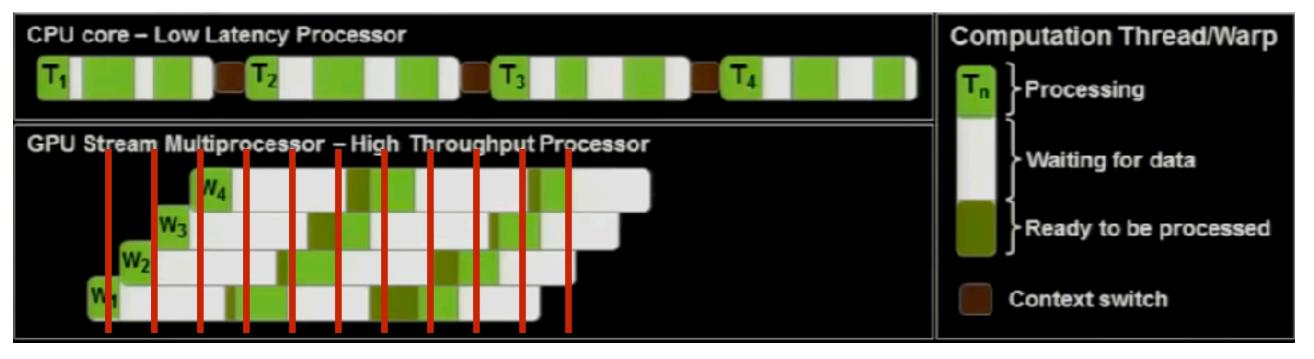
- CPU minimizes latency
- GPU hides latency by overlapping computation





GPU versus CPU

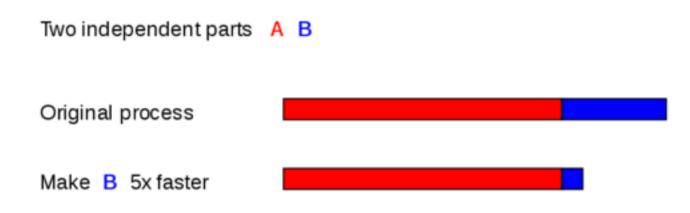
- CPU minimizes latency
- GPU hides latency by overlapping computation





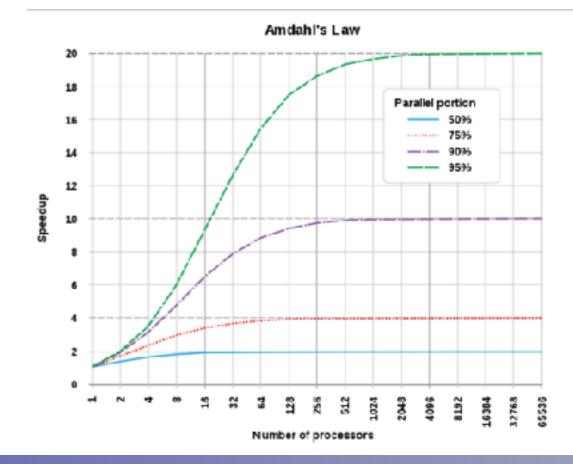
Amdahl's law

- A cpu has 8 core a GPU 2056 core
 - Should my code should be 200 faster?



 It depends which fraction of your code can use parallelism

 This is Amdahl's law given theoretical speed-up of your code

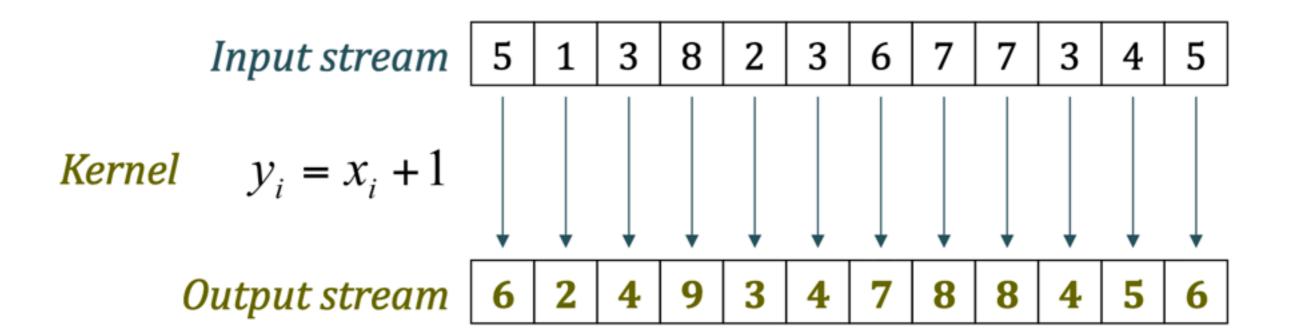


Speed-up in practise

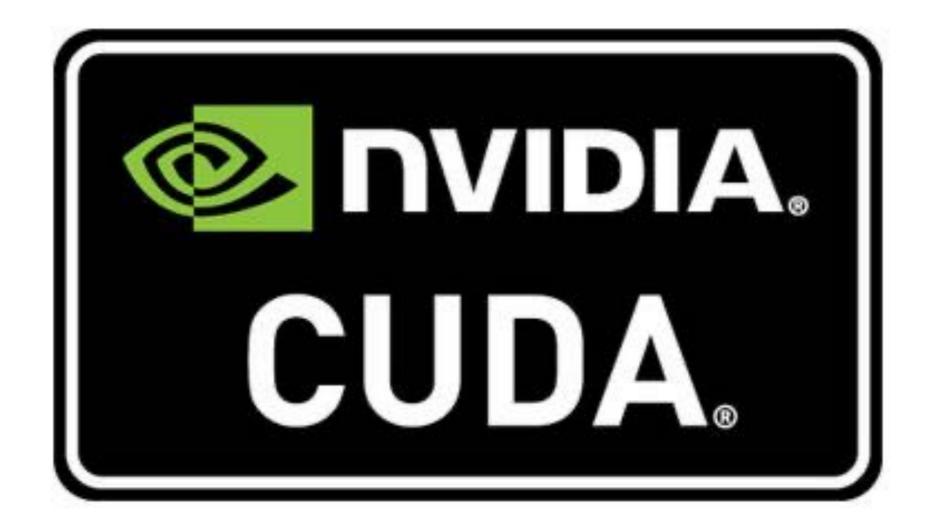
- Comparing speed of code between cpu and gpu are not really fair
 - Cost of the GPU/CPU
 - Huge speed-up typically means "bad" denominator
- A "normal" is around 5-20
 - Much higher number reported in some cases.
- GPU clock is slower than CPU clock
 - ➡ GPU ~ mhz
 - ➡ CPU ~ Ghz

Stream computing

- The idea of GPU are
 - "multiple data"
 - ➡ SAME operation
- Same as vectorisation on CPU (but different scale)



• You can have synchronisation between threads



- As for CPU, you do not want to code at assembler level
- First released in 2006
 - Restricted to nvidia GPU
 - Expose the raw computation power
 - No need of graphical knowledge

GPU availability

- Dragon 2:
 - Two machines with two Nvidia V100
- Manneback (UCL only)
 - Two machines with two Nvidia V100
 - One MI0
 - One K80
- (Future) Lumi European computer (EUROHPC)
 - Not Nvidia GPU machine
 - Cuda code need to be converted to HEAP
 - Alternative: OpenACC, OneAPI, Sycl, kokos,...

SLURM FOR GPU

• Check ressource

➡ sinfo --format="%N %.6D %P %G"

mb-bro080	1 cp3-gpu gpu:TeslaK80:2,localscratch:156
mb-cas101	1 gpu gpu:TeslaV100:2,mps:TeslaV100:100,localscratch:172
mb-cas102	1 gpu gpu:TeslaV100:2,mps:TeslaV100:100,localscratch:411
mb-sab040	1 gpu gpu:TeslaM10:4,localscratch:46

- First run iteratively
 - srun -p gpu --gres=gpu:TeslaV100:1 --pty bash
- Check module on the machine
 - module av
- Check that you have access to the GPU
 - nvidia-smi

SLURM FOR GPU

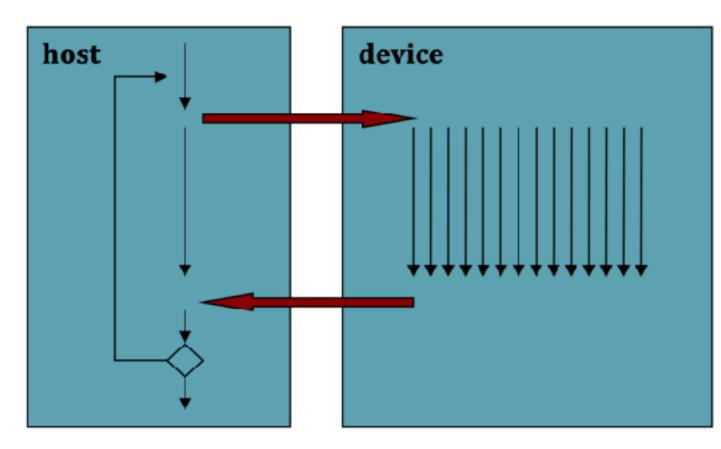
- Check ressource
 - ➡ sinfo --format="%N %.6D %P %G"

NODELIST NOD	ES PARTITI	ON GRES
drg2-w[001-01	17 l	patch* (null)
drg2-w[001-01	[7] 17	long (null)
drg2-w[018-01	.9] 2 g	gpu gpu:2
drg2-w[001-01	.7] 17 d	debug (null)

- First run iteratively
 - srun -p gpu --gres=gpu: I --pty bash
- Check module on the machine
 - module av
- Check that you have access to the GPU
 - nvidia-smi

Cuda Programming model

- A GPU needs to be controlled by a CPU.
 - ➡ All programs start by the CPU
 - Data are prepared on the CPU and moved to the GPU
 - ➡ GPU is crunching data
 - Data moved back to the cpu
 - Programs end



Kernel execution is asynchronous

Asynchronous memory transfers also available

Cuda Programming model

- The cpu is called the "host"
- The gpu is called the **"device"**
 - Viewed as a co-processor
- Function executed on gpu are called kernel
 - Executed in parallel on different data element
- Both the host/device have their own memory
 - Memory management is handle by the host
 - Automatic management is possible

Multi-processor/block/thread



- Main component
 - Memory
 - Streaming Multiprocessor (84 of them here)

Multi-processor/block/thread



- Main component
 - Memory
 - Streaming Multiprocessor (84 of them here)

Multi-processor/block/thread



Tex

- Memory
- Streaming Multiprocessor (84 of them here)

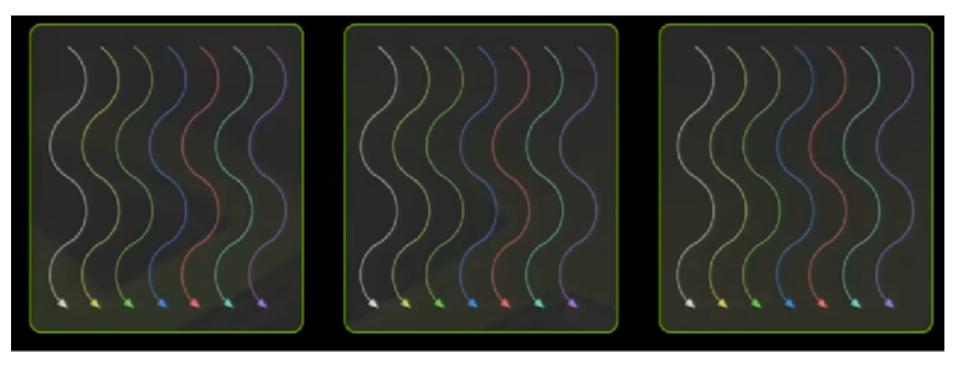
Tex:

192KB L1 Data Cache / Shared Memory

Tex

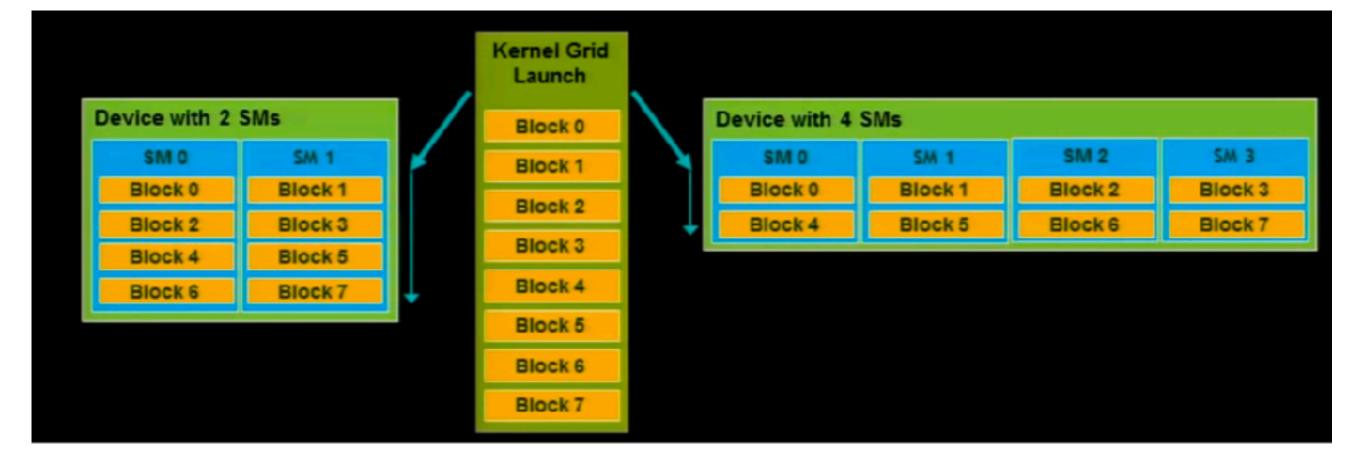
Тен

Block



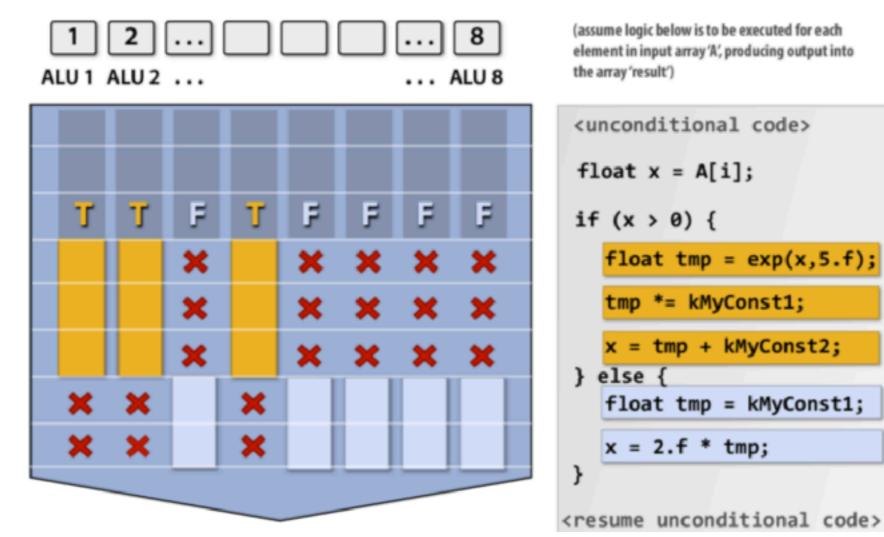
- Thread are grouped by block
 - Collaboration of thread (syncronization, shared memory)
- Up to 2048 thread per block
- Block are fully independent
 - Can be executed in any order
 - Can be executed on different GPU

• Separation into block allow you to adapt to various GPU in an easy way.



Wrap

- block are organised in wrap of 32 thread
 - Correspond to an hardware configuration
- Those 32 threads are working in lock step
 - Run the same command at the same time
 - If statement slows down the code



CECI training: Cuda

include <stdio.h>

#include <stdlib.h>
#include <algorithm>
#include <cmath>

```
void saxpy_cpu(float* vecX, float* vecY, float alpha, int n)
  for (int i=0; i < n; i++){</pre>
   vecY[i] = alpha * vecX[i] + vecY[i];
  }
int main(){
  int N = 1<<20; // 2^20 = 1,048,576
  float* x;
  float* y;
  x = (float *) malloc(N*sizeof(float));
  y = (float *) malloc(N*sizeof(float));
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
   y[i] = 2.0f;
  }
  saxpy_cpu(x, y, 2.f, N);
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, std::abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);
```

}

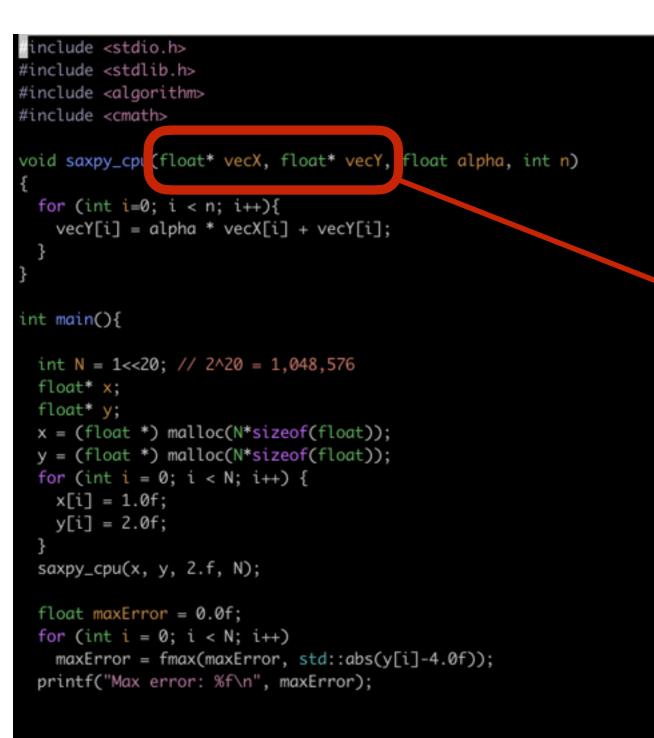
CECI training: Cuda

include <stdio.h>

#include <stdlib.h>
#include <algorithm>
#include <cmath>

```
void saxpy_cpu(float* vecX, float* vecY, float alpha, int n)
  for (int i=0; i < n; i++){</pre>
   vecY[i] = alpha * vecX[i] + vecY[i];
  }
int main(){
  int N = 1<<20; // 2^20 = 1,048,576
  float* x;
  float* y;
  x = (float *) malloc(N*sizeof(float));
  y = (float *) malloc(N*sizeof(float));
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
   y[i] = 2.0f;
  3
  saxpy_cpu(x, y, 2.f, N);
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, std::abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);
```

• $\overrightarrow{y} = a\overrightarrow{x} + \overrightarrow{y}$



• $\overrightarrow{y} = a\overrightarrow{x} + \overrightarrow{y}$

 float* is used here for passing an array

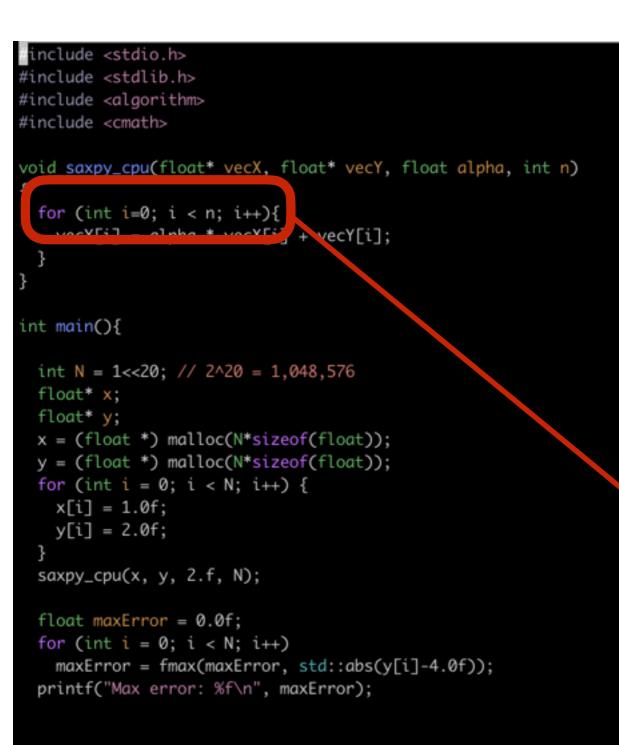
include <stdio.h>

#include <stdlib.h>
#include <algorithm>
#include <cmath>

```
void saxpy_cpu(float* vecX, float* vecY, float alpha, int n)
  for (int i=0; i < n; i++){</pre>
   vecY[i] = alpha * vecX[i] + vecY[i];
int main(){
  int N = 1 < 20; // 2^20 = 1,048,576
  float* x;
  x = (float *) malloc(N*sizeof(float));
  y = (float *) malloc(N*sizeof(float));
                  x[i] = 1.0f;
   y[i] = 2.0f;
  saxpy_cpu(x, y, 2.f, N);
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
   maxError = fmax(maxError, std::abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);
```

• $\overrightarrow{y} = a\overrightarrow{x} + \overrightarrow{y}$

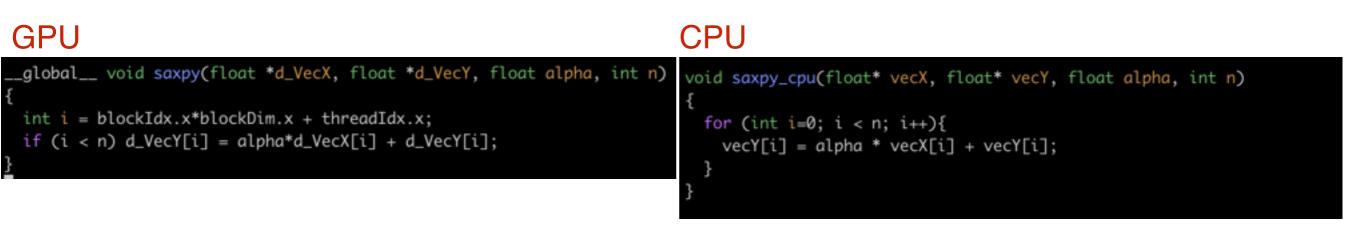
- float* is used here for passing an array
 - that array is assigned dynamically (malloc)



• $\overrightarrow{y} = a\overrightarrow{x} + \overrightarrow{y}$

- float* is used here for passing an array
- that array is assigned dynamically (malloc)
- We explicitly loop over the data element

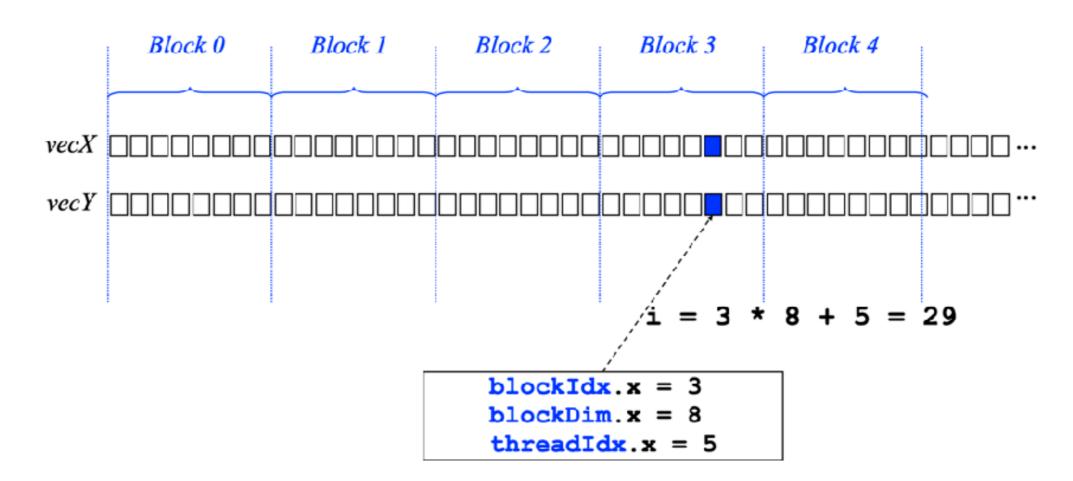
Cuda version: kernel



- No loop anymore !!
 - Each thread will take care of one data
 - Need to compute which element each thread has to handle.
 - Various variable defined for that
 - blockldx.x (.y/ .z if 2D and 3D): id of the current block
 - blockDim.x: number thread in Block (for that dimension)
 - threadIdx.x: id of the current thread inside the block

Index

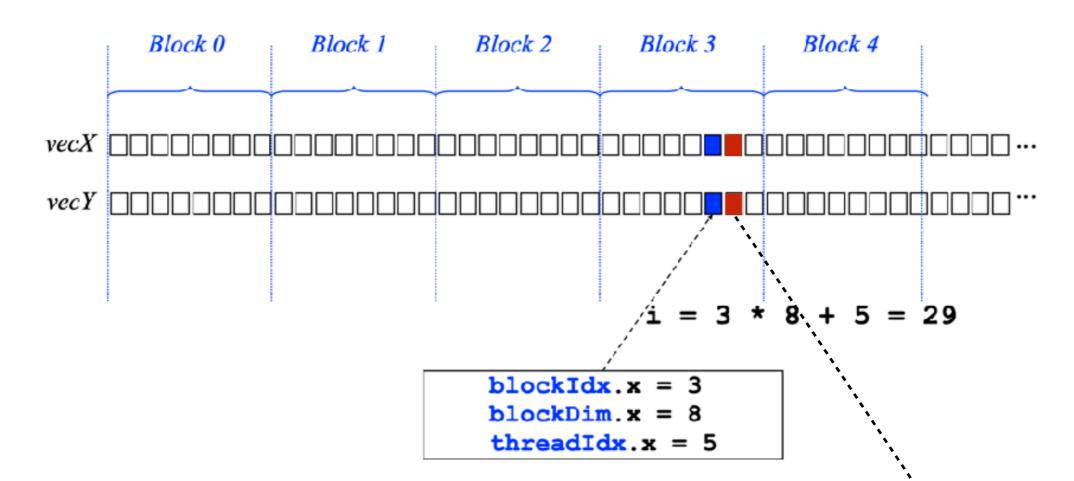
• Let's give an example:



- Super Important coalesced memory:
 - Reading (global) memory should be from adjacent memory address for the threads

Index

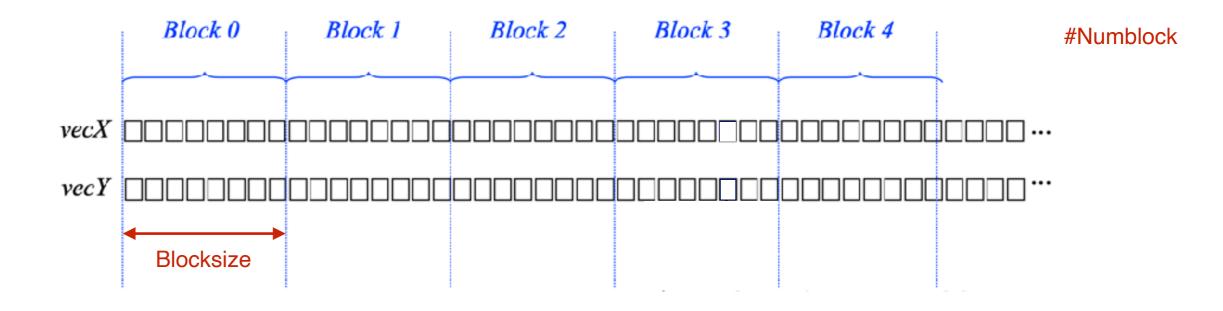
• Let's give an example:



- Super Important coalesced memory:
 - Reading (global) memory should be from adjacent memory address for the threads

Kernel call

- How do you call a kernel?
 - saxpy<<<numblock, blocksize>>>(d_x, d_y, a, n)



- blocksize: number of thread in a block
 - Should be multiple of 32 (due to wrap)
 - Maximum of 2048
 - depends of the GPU capabilities

- Code steps in more details:
 - I. Initialise GPU
 - 2. Initialise variable on the host (cpu)
 - 3. Allocate memory on the device (gpu)
 - 4. Move data from host to device
 - 5. Execute kernel on device
 - 6. Move back results
 - 7. Clean up (deallocation)

- I. Initialise GPU
- 2. Initialise variable on the host (cpu)



- culnit(0) is NOT required for the code to work
 - Will be called automatically at first cuda function call
 - Nice to use for profiling
 - Otherwise first call much slower than expected

3. Allocate memory on the device (gpu)

float *d_x, *d_y; cudaMalloc(&d_x, N*sizeof(float)); cudaMalloc(&d_y, N*sizeof(float));

- cudaMalloc does NOT follow the exact same syntax as a malloc:
 x = (float*)malloc(N*sizeof(float));
 - The cuda rule for any function is to return an error code
 - So the cuda malloc does not return a pointer but has one more argument (pointer of pointer)
- Here we use "d_" prefix to indicated device pointer.
 - Useful convention for code clarity

4. Move data from host to device

cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice); cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

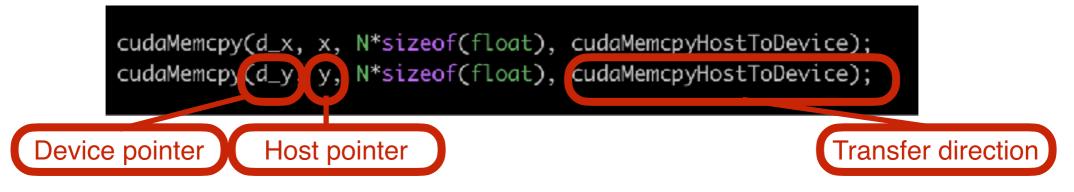
4. Move data from host to device

cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice); cudaMemcpy(d_y) y, N*sizeof(float), cudaMemcpyHostToDevice); Device pointer

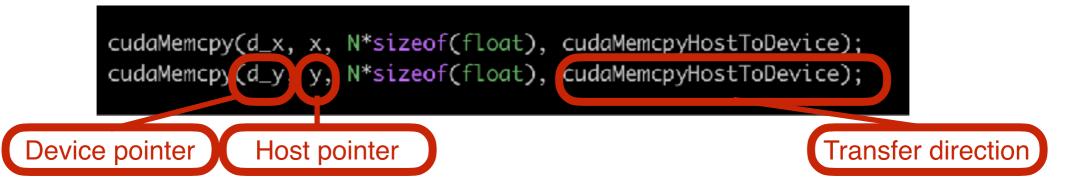
4. Move data from host to device

cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice); cudaMemcpy(d_y y, N*sizeof(float), cudaMemcpyHostToDevice); Device pointer Host pointer

4. Move data from host to device



4. Move data from host to device



- Quite slow transfer but 2 tricks:
 - I. For simple initialisation/value
 - cudaMemSet(d_x, 0, N*sizeof(xxxxx))
 - 2. Used hosted pinned memory for host
 - cudaMallocHost(&&x_host, size)
 - Slower allocation on host

5. Execute kernel on device

```
// Perform SAXPY on 1M elements
int blocksize = 512;
int nblock = N/blocksize + ( n % blocksize > 0 ? 1: 0 );
saxpy<<<nblock, blocksize>>>( d_x, d_y, 2.0f, N);
```

5. Execute kernel on device

```
// Perform SAXPY on 1M elements
int blocksize = 512;
int nblock = N/blocksize + ( n % blocksize > 0 ? 1: 0 );
saxpy<<<nblock, blocksize>>>( d_x, d_y, 2.0f, N);
```

- Computing the number of block needed
 - Special <<<A, B, C, D >>> syntax
 - A: number of block
 - B: number of thread per block
 - C: dynamically allocated shared memory
 - D: which stream to use

5. Execute kernel on device

```
// Perform SAXPY on 1M elements
int blocksize = 512;
int nblock = N/blocksize + ( n % blocksize > 0 ? 1: 0 );
saxpy<<<nblock, blocksize>>>( d_x, d_y, 2.0f, N);
```

- Computing the number of block needed
 - Special <<<A, B, C, D >>> syntax
 - A: number of block
 - B: number of thread per block
 - C: dynamically allocated shared memory
 - D: which stream to use

__global__ void saxpy(float *d_VecX, float *d_VecY, float alpha, int n)

- __global___ to use for kernel called from the host
- __device___ for GPU function call from a kernel

- 6. Move back results
- 7. Clean up (deallocation)

```
cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
float maxError = 0.0f;
for (int i = 0; i < N; i++)
  maxError = max(maxError, abs(y[i]-4.0f));
printf("Max error: %f\n", maxError);
cudaFree(d_x);
cudaFree(d_y);
free(x);
free(y);
```

Full code

```
#include <stdio.h>
__global__ void saxpy(float *d_VecX, float *d_VecY, float alpha, int n)
 int i = blockIdx.x*blockDim.x + threadIdx.x;
 if (i < n) d_VecY[i] = alpha*d_VecX[i] + d_VecY[i];
int main(void)
 int N = 1 << 20;
 float *x, *y;
 x = (float*)malloc(N*sizeof(float));
 y = (float*)malloc(N*sizeof(float));
  cuInit(0);
  float *d_x, *d_y;
  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));
 for (int i = 0; i < N; i++) {
   x[i] = 1.0f;
   y[i] = 2.0f;
 }
 cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
 // Perform SAXPY on 1M elements
     blocksize = 512;
  int
  int nblock = N/blocksize + ( n % blocksize > 0 ? 1: 0 );
  saxpy<<<nblock, blocksize>>>( d_x, d_y, 2.0f, N);
  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
   maxError = max(maxError, abs(y[i]-4.0f));
 printf("Max error: %f\n", maxError);
  cudaFree(d_x);
```

noold v

• How to compile it?

Compilation of cuda code

- Module load CUDA
- nvcc -arch=sm_70 saxpy.cu -o saxpy
 - You can have additional flags for C++ par of the code (library linking, -O3,...)
 - Arch allows to have a minimum target gpu
 - No dedicated flag for additional GPU optimisation
 - ➡ GPU does support multiple file source code
 - But seriously limit optimisation
 - Cudal I starts supports for that but still limited.

Is GPU always faster?

• GPU

CPU

	@mb-cas102 omatt]\$ time ./saxpy ror: 0.000000
real	0m2.401s
user	0m0.752s
sys	0m0.555s

[omatt@mb-cas102 omatt]\$ time ./saxpy_cpu										
max erro	or: 0.00000	00								
real	0m0.803s									
user	0m0.704s									
sys	0m0.097s									

• You need to have a lot of work to do on the GPU to hide the latency, the initialisation, ...

• You have to manage memory: Plenty of type of memory on the GPU

LD Instruction Cache Warp Scheduler (52 thread/clk) Dispatch Unit (52 thread/clk) Register File (16,384 x 32-bit) Dispatch Unit (52 thread/clk) NT32 NT32 NT32 FF32 FF32 FF24 FF44 NT32 NT32 NT32 FF32 FF32 FF24 FF44 NT32 NT32 NT32 FF32 FF32 FF24 FF44 NT32 NT32 NT32 FF32 FF32 FF24 FF24 FF44 NT32 NT32 NT32 FF32 FF32 FF24 FF24 FF44 NT32 NT32 NT32 FF32 FF32 FF34 TENSOR CORE NT32 NT32 FF32 FF32 FF34 TENSOR CORE </th <th colspan="13">M L1 Instruction Cashe</th>	M L1 Instruction Cashe												
Listential FP22 FP24 FP24 FP24 FP22 FP24 FP22 FP24 FP22 FP24 FP22 FP24 FP22 FP24 FP22 FP24 FP23 FP24 FP23 FP24 FP23 FP24 FP24 FP24 FP23 FP24 FP33 FF34 FF33 FF34 FF34 FF34 FF33 FF34 FF34 FF34 FF33 FF34 FF34 FF34 FF34 FF34 FF33 FF34 FF34 FF34 FF34 FF34 FF34 FF34 FF33 FF34 FF34 FF34 FF34 FF34 FF33	Warp Scheduler (32 ti	hæsd. ⁽ clk)	Warp Sche	duler (32 ti	hread/clk)								
INT32 INT32 <td< th=""><th>Register File (18,384</th><th>4 x 32-bit)</th><th>Register F</th><th>ile (16,384</th><th>4 x 32-bit)</th></td<>	Register File (18,384	4 x 32-bit)	Register F	ile (16,384	4 x 32-bit)								
L0 Instruction Cache Warp 3chedwier (32 thread/clk) Dispatch Unit (32 thread/clk) Register File (16,384 x 32-bit) Int32 IN132 FP32 FP64 Int32 IN132 FP32 FP64 Int32 IN132 FP32 FP64 Int32 IN13	INT32 INT32 ITT32 ITT32 ITT32		INT32 INT32 FP32 FP32 INT32 INT32 FP32 FP32										
INT32 IF932 IF932 IF942 IF944 INT32 INT32 IF932 IF932 IF944 INT32 INT32 IF932 IF932 IF944 INT32 INT32 IF932 IF92 IF944 INT32 INT32 IF932 IF932 IF944 INT32 INT32 IF932 IF932 IF944 INT32 INT32 IF932 IF932 IF942 INT32 INT32 IF932 IF932 IF944 INT32	ST ST ST ST ST ST	ST ST STO	L0 Instruction Cache Warp Scheduler (32 thread/clk) Dispatch Unit (32 thread/clk)										
	Warp Scheduler (32 ti Dispatch Un 1 (32 the	hneadlicik) read/cik)	Warp Sche Dispetch	duler (32 t Unit (32 th	hread/clk) read/clk)								
192ND L1 Data Cache / Shareo Memory	Warp Scheduler (32 B) Dispatch Unit [32 Ihr Register File (15,384 INT32 INT32	heead/clk) read/clk(# x 32-bft)	Warp Schw Dispatch Register P INT32 INT32 FP32 FP32 INT32 INT32 FP32 FP32	Elefer (32 f Unit (32 th File (16, 38- FP64 FP64 FP64 FP64 FP64 FP64 FP64	hread/clk) read/clk) 4 x 32-bit)								
Tex Tex Tex Tex	Warp Scheduler (32 U Dispatch Unit [32 Ihr Register File (15,384 INT32 INT32 IN	TENSOR CORE	Warp Schw Dispatch Register F INT32 INT32 FP32 FP32 INT32 INT32 IP32 FP32 INT32 INT32 INT32 IP32 IP32 INT32 INT32 INT32 INT32 INT32	defer (32 f Unit (32 th File (16,38- FP64 FP64 FP64 FP64 FP64 FP64 FP64 FP64	hread/clk) read/clk) 4 x 32-bit) TENSOR CORE								

Inside each SM:

CECI training: Cuda

• You have to manage memory: Plenty of type of memory on the GPU

M L1 Instruction Cashe											
L0 Instruction Cach Warp Scheduler (92 three			struction C eduler (32 t								
Register File (18,384 x	. 32-bit)	Register I	File (16,384	4 x 32-bit)							
INT32 INT32 FP32 FP32 FP64 INT32 INT32 FP32 FP32 FP64 INT32 INT32 FP32 FP62 FP64 INT32 INT32 FP32 FP62 FP64 INT32 INT32 FP32 FP52 FP64		INTAX INTAX <th< th=""><th>FP64 FP64 FP64 FP64 FP64 FP64 FP64 FP64</th><th>TENSOR CORE</th></th<>	FP64 FP64 FP64 FP64 FP64 FP64 FP64 FP64	TENSOR CORE							
L0 Instruction Cash Warp Scheduler (32 three Dispetch Up 4 121 three Register File (15,384 x	ead/clk)	L0 in Warp 9ch	struction C edular (32 f File (16,38-	iacha Ihread/clk)							
L0 Instruction Caef Warp Scheduler (32 three Register File (15,384 x http://within.com/ h	TENSOR CORE	L0 Im Warp 3ch Nagister I NT32 INT32 FP32 FP32 INT32 INT32 FP32 FP32	File (16, 38) File (16, 38) FP64 FP64 FP64 FP64 FP64 FP64 FP64	techa Ihread/clk) 4 x 32-bit) TENSOR CORE							
L0 Instruction Cach Warp 3c heduler (32 three Register File (15,384 x http://within.com/ http://withic.com/	TENSOR CORE	L0 In Warp 3ch Register 1 INT32 INT52 FP32 FP32 INT32 INT52 FP32 FP32	File (16, 38 File (16, 38 FP64 FP64 FP64 FP64 FP64 FP64	iacha Ihread/clk) 4 x 32-bit)							

Inside each SM:

- Register
 - Fastest memory
 - Thread specific
 - Very limited amount
 - Overflow goes to L1/RAM

CECI training: Cuda

• You have to manage memory: Plenty of type of memory on the GPU

1122 1123	LO In	istruction Ca	iche	L.	Instruction (lache					
Register File (18,384 x 32-bit) Register File (18,384 x 32-bit) Register File (18,384 x 32-bit) NT32 NT32 FF32 FF32 FF64 NT32 NT32 FF32 FF32 FF74 NT32 NT32 FF32 FF32 FF34 NT32 NT32 FF32 FF34 NT32 NT32 FF32 FF34 NT32 NT32 FF32 FF34 NT32 NT32 FF32 FF34	Warp Sch	eduler (32 ti	nead/cik)	Warp St	heduler (32)	thread/clk)					
INTEGE INTEGE<	Dispatel	Un 1 (32 1hr	cad/cllk)	Dispat	ch Unit (32 tr	nread/elk)					
HT22 HT22 HT22 FF24	Register	File (16,384	x 32-bit)	Registe	r File (16,38	i4 x 32-bit)					
Kit 32 Kit 32<	INTAZ INTAZ FP32 FP32	FP64		INTAZ INTAZ FP32 FP3	2 FP64						
INT22 INT22 IP32 IP34 IP32 IP34 IP34	INT32 INT32 FP32 FP32	FP64		INTEE INTEE FPB2 FPB	FT64						
No.0 No.0 TENSOR CORE INT.2 INT.2 <thint.2< th=""> INT.2 INT.2</thint.2<>	INT32 INT32 FP32 FP32	FP64		INTS2 INTS2 FP32 FP3	2 FP64						
IKT32 IF932 IF932 IF932 IF932 IF932 IF932 IF932 IF944 IKT32 IKT32 <td< td=""><td>INTAZ INTO2 FP32 FP32</td><td>FP04</td><td>TENSOR CORE</td><td>INTAS INTOS FPAS FPA</td><td>2 FP64</td><td>TENSOR CORE</td></td<>	INTAZ INTO2 FP32 FP32	FP04	TENSOR CORE	INTAS INTOS FPAS FPA	2 FP64	TENSOR CORE					
INT22 FF22 FF24	INT32 INT32 FP32 FP32	FP84		INTER INTER FREE FRE	2 FP61	IL ISON CONL					
KT22 FP32	INTAZ INTAZ FP32 FP32	FP64		INTAZ INTAZ EP32 EP3	PP64						
Light Light <th< td=""><td>INT32 INT32 FP32 FP32</td><td>FP64</td><td></td><td>INT82 INT82 FP82 FP8</td><td>2 FP64</td><td></td></th<>	INT32 INT32 FP32 FP32	FP64		INT82 INT82 FP82 FP8	2 FP64						
Image: Im	INTAR INTAR FP32 FP32	FP04		INTRE INTRE FPRE FPR	FP64						
Warp 3cheduler (32 thread/clk) Warp 3cheduler (32 thread/clk) Uspateh Un 1 (32 thread/clk) Dispatch Un 1 (32 thread/clk) Dispatch Un 1 (32 thread/clk) Dispatch Unit (32 thread/clk) NT32 NT32 F932 F964 INT32 INT32 F932 F964	LD/ LD/ LD/ LD/ ST ST ST ST	LDV LDV ST ST	ST ST SFU	LDV LDV LDV LDV ST ST ST ST	LD' LD' ST ST	LD' LD' SFU					
Dispatch Unit [32 thread/clk]	L0 I	istruction Ca	ache		Instruction (lacha					
Krizz Nrizz Prizz Priz Prizz Prizz <thp< td=""><td></td><td></td><td></td><td colspan="8">10</td></thp<>				10							
INT32 INT32 <th< td=""><td>Charpet C</td><td>101111 DE 111</td><td></td><td>Dispat</td><td>nih Limit #23.4</td><td></td></th<>	Charpet C	101111 DE 111		Dispat	nih Limit #23.4						
INT32 INT32 <th< td=""><td>Register</td><td></td><td></td><td></td><td></td><td>nreadl'eik)</td></th<>	Register					nreadl'eik)					
INT32 INT32 <th< td=""><td></td><td>File (16,384</td><td></td><td>Registe</td><td>r File (16,38</td><td>nreadl'eik)</td></th<>		File (16,384		Registe	r File (16,38	nreadl'eik)					
INT32 INT32 <th< td=""><td>INT32 INT32 FP32 FP32</td><td>File (16,384 FP64</td><td></td><td>Registe</td><td>r File (16,38 2 FP64</td><td>nreadl'eik)</td></th<>	INT32 INT32 FP32 FP32	File (16,384 FP64		Registe	r File (16,38 2 FP64	nreadl'eik)					
INT32 INT32 FP32 FP32 FP64 FP32 FP32 FP64 FP32 FP32 FP32 FP32 FP32 FP32 FP34 FP54 FP32 FP32 FP32 FP34 FP54 FP32 FP32 FP32 FP32 FP34 FP54 FP32 FP32 FP32 FP32 FP32 FP34 FP54 FP32 FP32 FP32 FP32 FP34 FP54 FP54 FP32 FP32 FP32 FP32 FP34 FP54 FP54 FP54 FP54 FP54 FP54 FP54 FP5	INT32 INT32 FP32 FP32 INT32 INT32 FP32 FP32	File (16,384 FP64 FP64		Registe INT32 INT32 FP32 FP3 INT32 INT32 FP32 FP3	r File (16,38 2 FP64 2 FP64	nreadl'eik)					
INT32 INT32 INT32 FP32 FP32 FP64 INT32 INT32 INT32 INT32 INT32 FP32 FP32 FP64	INT32 INT32 FP32 FP32 INT32 INT32 FP32 FP32 INT32 INT32 FP32 FP32	File (15,384 FP64 FP64 FP64	x 32-bit)	Registe INT32 INT32 FP32 FP3 INT32 INT32 FP32 FP3 INT32 INT32 FP32 FP3	r File (16,38 2 FP64 2 FP64 2 FP61	nread'elk) 14 x 32-bit)					
INT32 INT32 FP32 FP32 FP64	INT32 INT32 FP32 FP32 INT32 INT32 FP32 FP32 INT32 INT32 FP32 FP32 INT32 INT32 FP32 FP32	Filo (15,384 FP64 FP64 FP64 FP64	x 32-bit)	Registe INT32 INT32 FP32 FP3 INT32 INT32 FP32 FP3 INT32 INT32 FP32 FP3 INT32 INT32 FP32 FP3	r File (16,38 2 FP64 2 FP64 4 FP64 2 FP64	nread'elk) 14 x 32-bit)					
	INT32 INT32 FP32 FP32 INT32 INT32 FP32 FP32	Filo (15,384 FP64 FP64 FP64 FP64 FP64	x 32-bit)	Register INT32 INT32 FP32 FP32 INT32 INT32 IP32 FP32 INT32 INT32 INT32 FP32 FP33	r File (16,38 2 FP64 2 FP64 2 FP64 2 FP64 2 FP64 2 FP64	nread'elk) 14 x 32-bit)					
LEY	INT32 INT32 FP32 FP32 INT32 INT32 FP32 FP32	Filo (15,384 FP64 FP64 FP64 FP64 FP64 FP64	x 32-bit)	INT32 INT32 FP32 FP33 INT32 INT32 FP32 FP33 FP33 INT32 INT32 INT32 FP32 FP33 INT33 INT32 INT32 FP32 FP33	r File (16,38 2 FP64 2 FP64 2 FP61 2 FP64 2 FP64 2 FP64 2 FP64	nread'elk) 14 x 32-bit)					
	INT32 INT32 FP32 FP32 INT32 INT32 FP32 FP32	Filo (15,384 FP64 FP64 FP64 FP64 FP64 FP64 FP64	x 32-bit)	INT32 INT32 FP32 FP33 INT32 INT32 INT32 FP32 FP33	r File (16,38 2 FP64 2 FP64 2 FP64 2 FP64 2 FP64 2 FP64 2 FP64 2 FP64	nread'elk) 14 x 32-bit)					
	NT32 NT32 FP32 FP32 NT32 NT32 NT32 FP32	Filo (15,384 FP64 FP64 FP64 FP64 FP64 FP64 FP64 FP6		Register INT32 INT32 FP32 FP32 INT32 INT32 IP32 FP33 INT32 INT32 IP32 FP32 INT32 INT32 IP32 FP33 INT32 INT32 IP32 FP33 INT32 INT32 IP32 FP33	r File (16,38 2 FP64 2 FP64 2 FP64 2 FP64 2 FP64 2 FP64 2 FP64 2 FP64	rread'elk) 14 x 32-bit) TENSOR CORE					

Inside each SM:

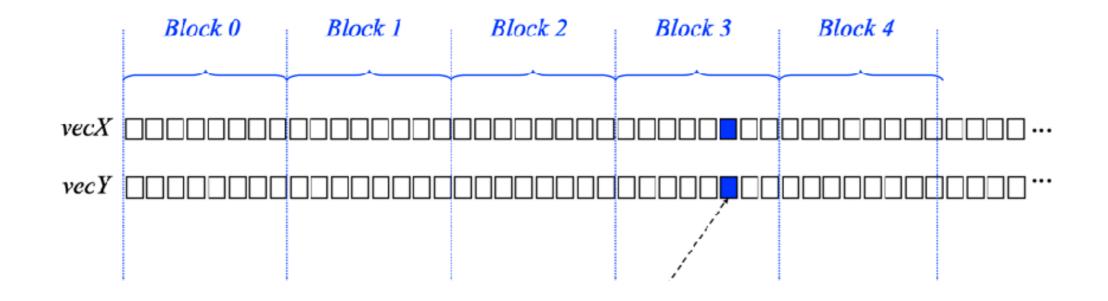
- Register
 - Fastest memory
 - Thread specific
 - Very limited amount
 - Overflow goes to L1/RAM
- Shared memory
 - Limited amount
 - Block wide memory
 - __shared_

CECI training: Cuda

- You have to manage memory: Plenty of type of memory on the GPU
- Outside the SM
 - Global memory
 - High bandwidth (900Gb/s) but High latency
 - High number of thread need to hide this latency
 - Default memory for cpu/gpu pointer

Index

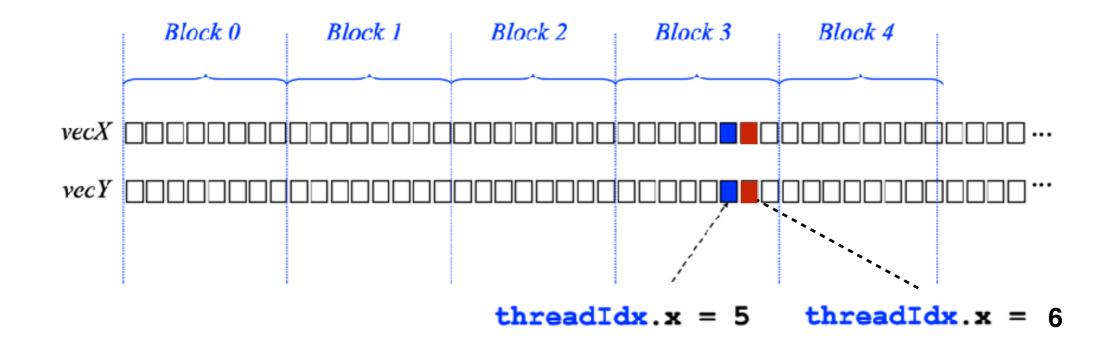
• This is how the memory should be read/write by the various thread



• You need to be careful with 2D array to be sure that you follow that pattern

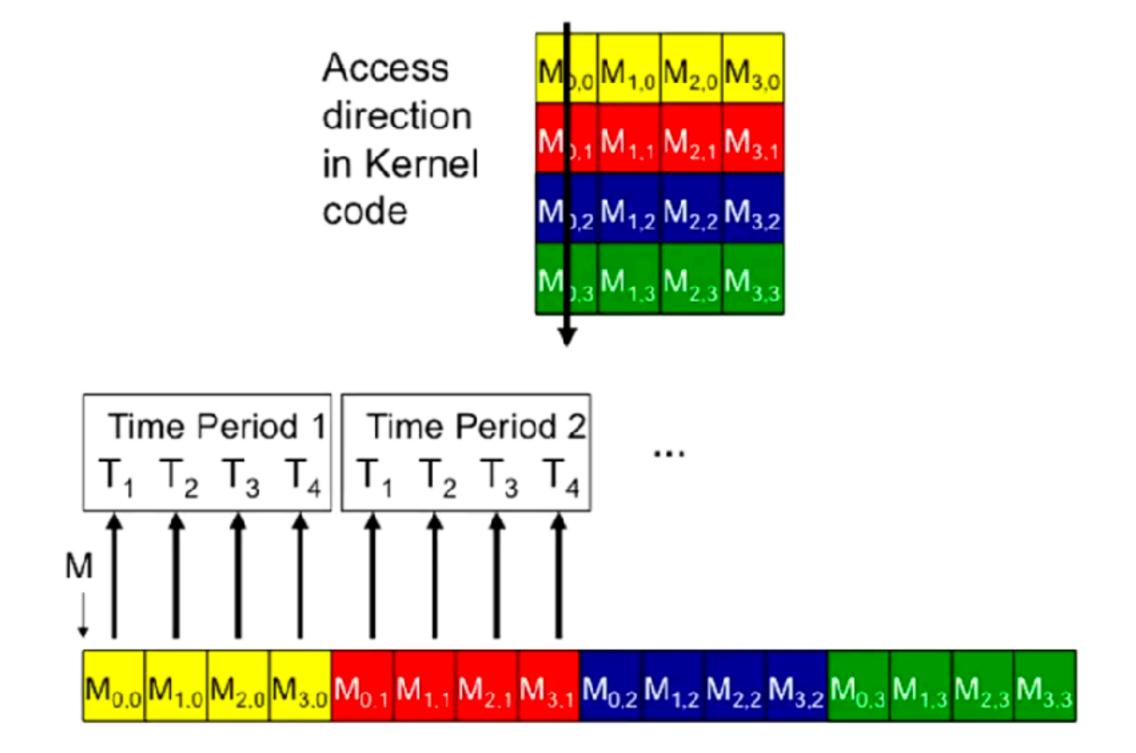
Index

• This is how the memory should be read/write by the various thread



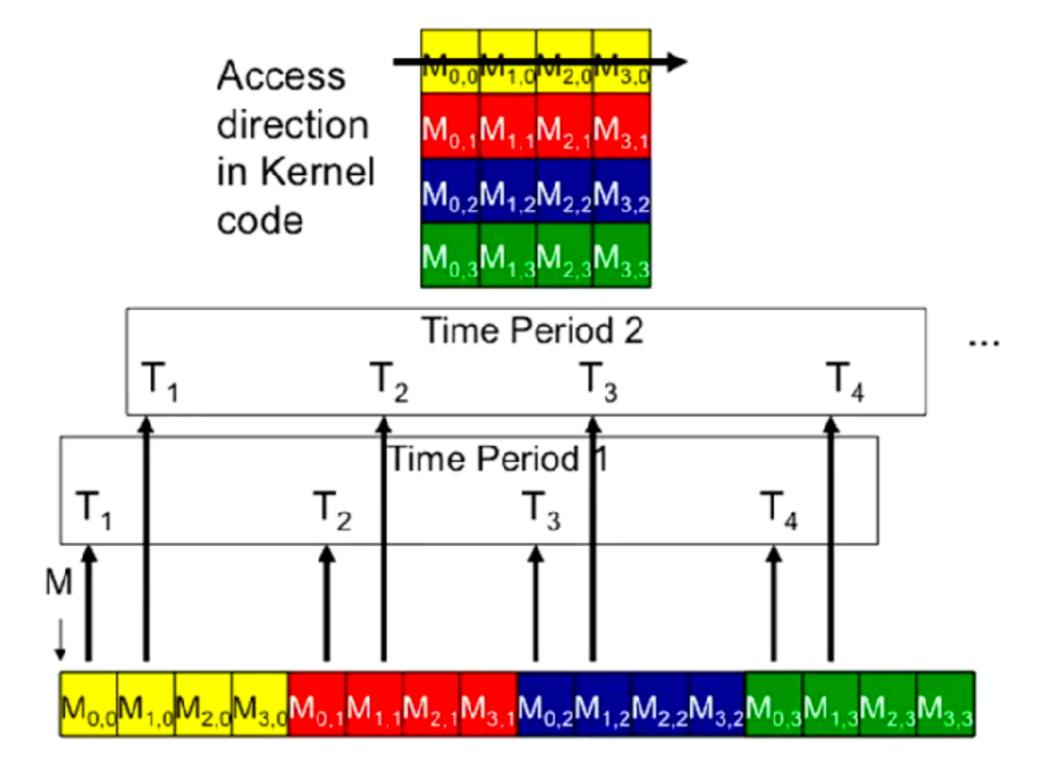
• You need to be careful with 2D array to be sure that you follow that pattern

Coalesced memory



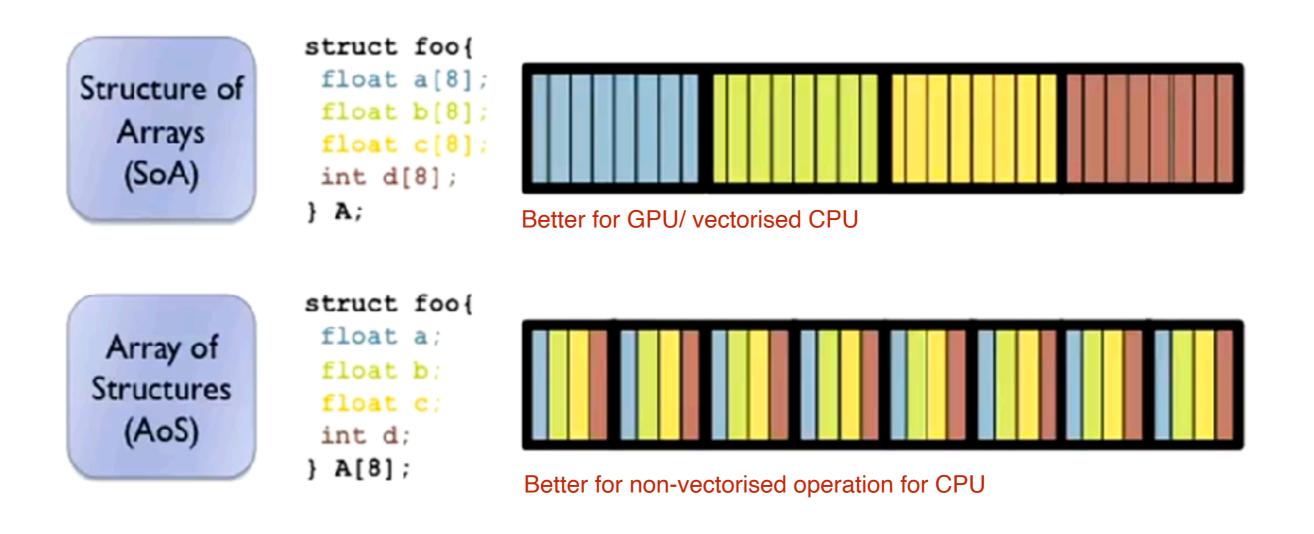
Slide credit: Hwu & Kirk

Uncoalesced Memory



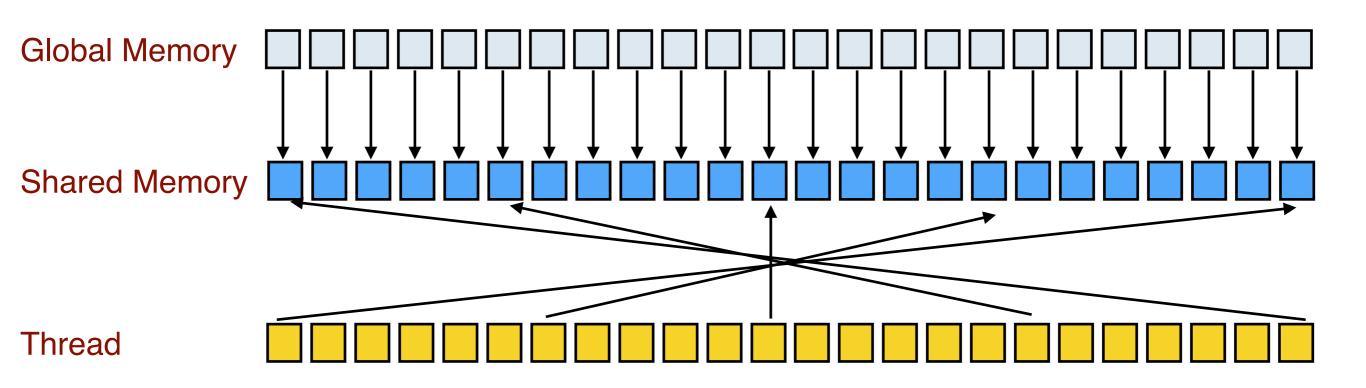
Structure of array

AoS vs. SoA



Coalesced access

- Coalesced access not possible?
 - ➡ Use shared memory as a cache

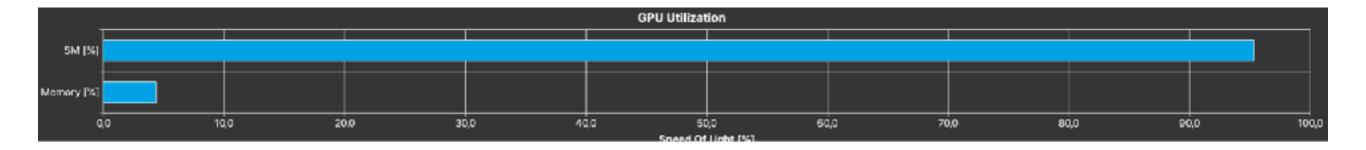


- Constant memory
 - Will be put in cache (same speed as shared memory)
- Texture memory
 - Related to graphics
- Unified Memory
 - Special type of global memory
 - Accessible both on cpu and gpu
 - cudaMallocManaged(&&x, size)
 - Pointer available both on device and on cpu

CUDA profiler

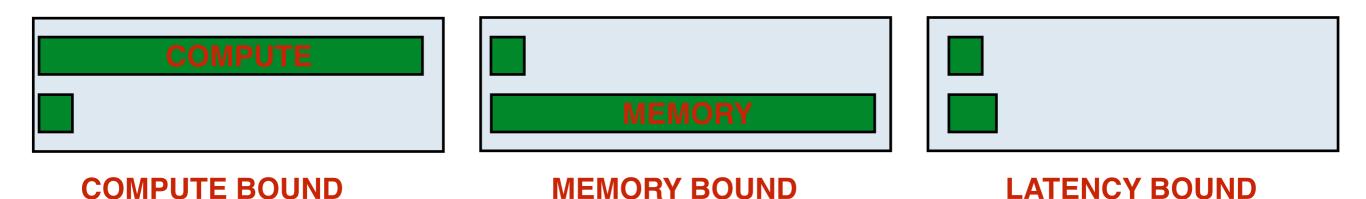
- nv-nsight-cu-cli -o profile --target-processes all ./ saxpy
 - Executable is also sometimes "ncu"
 - The more convenient is to download back that profile on your laptop and use "nsight compute" to visualise the data (do not need a GPU on that machine)
- On cluster mode, you need to be sudo to run those command. Contact us if needed.

What is the limitation of your kernel?

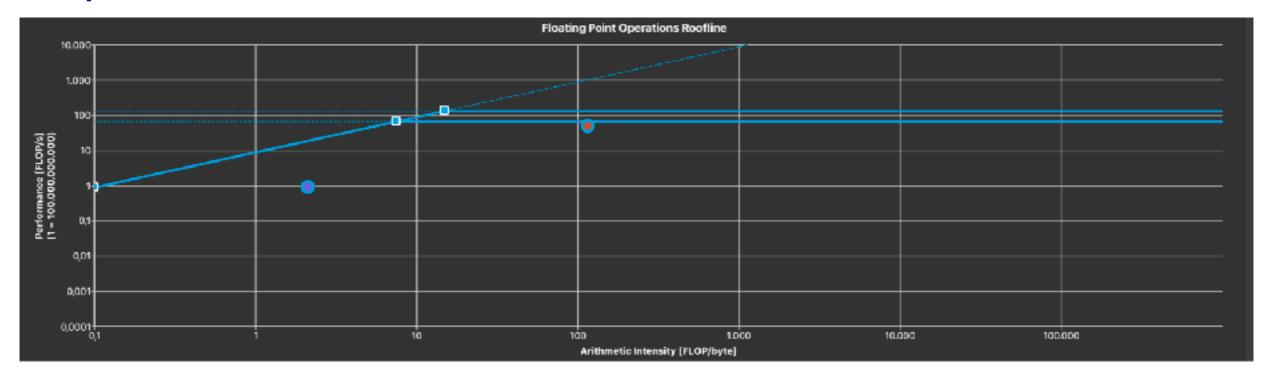


- Here two metric
 - How much the code compute (here more than 90%)
 - ➡ How much memory you use (here 5%)
- This indicates what limit your computation
 - Here we are Compute bound
 - But many memory available
 - We can try to cache computation

What is the limitation of your kernel?

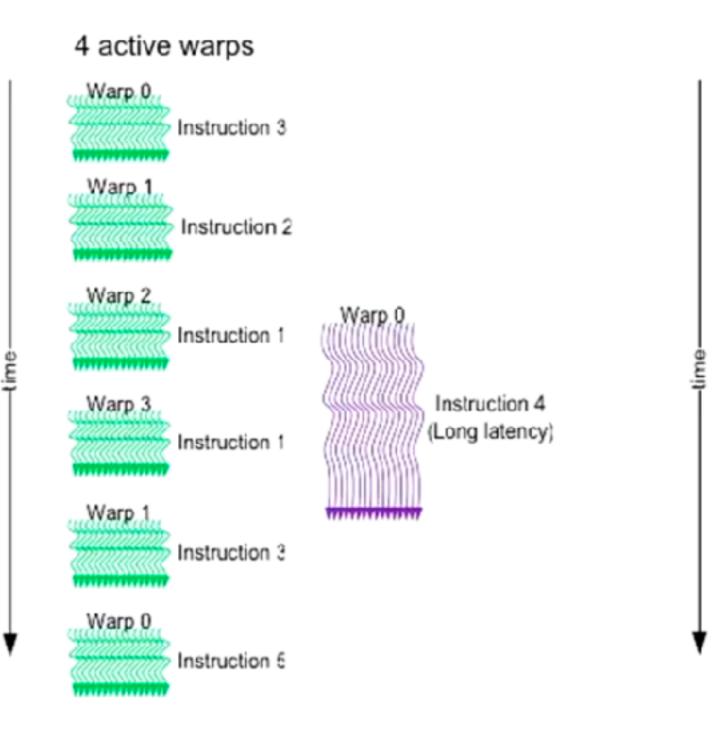


- Ideal case: compute AND memory bound
- If you are latency bound you need to allow more parallelism

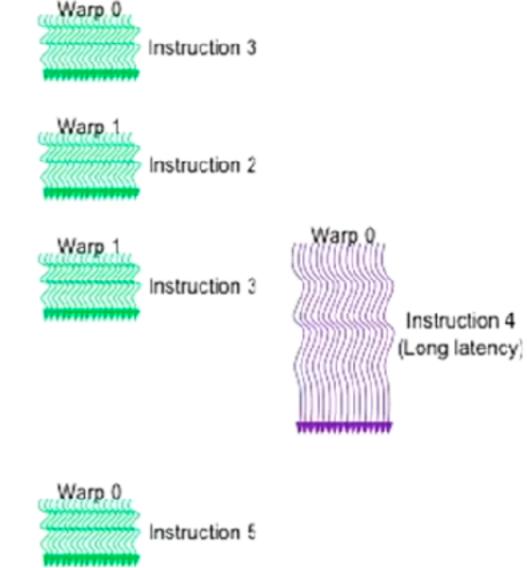


Effect of occupancy

• Hide latency with other wrap

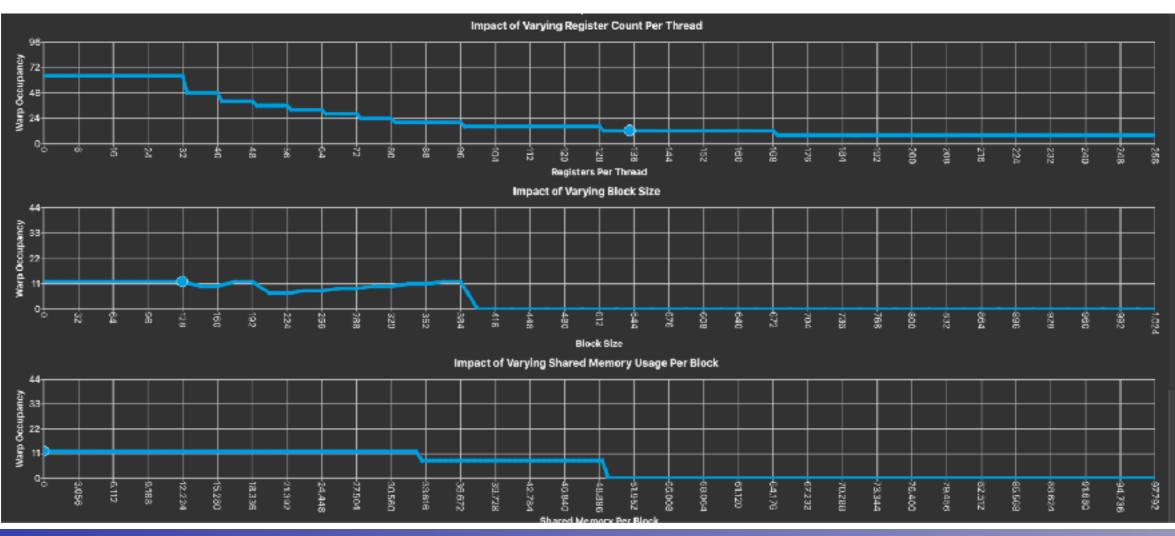


2 active warps



Occupancy

- Occupancy is limited
 - Each SM has limited ressources
 - Maximum number of wrap (64)
 - Maximum number of block (32)
 - Register usage (256Kb)
 - Shared memory usage (64Kb)

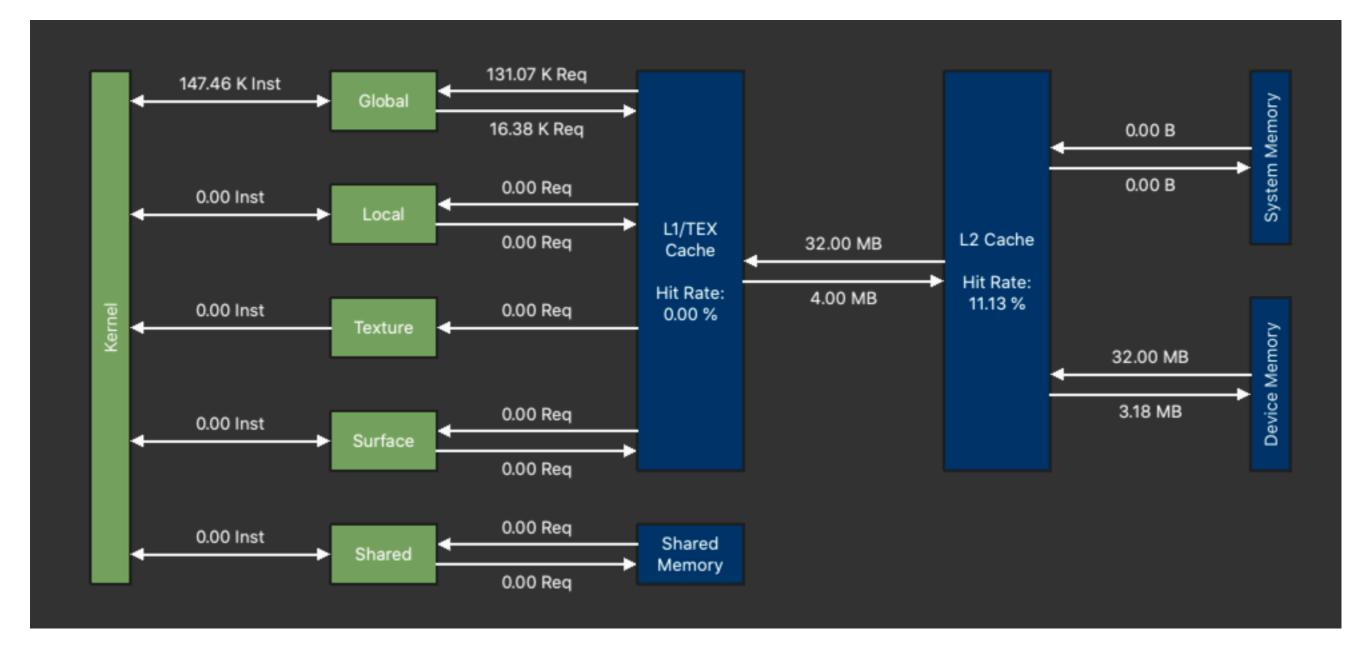


CECI training: Cuda

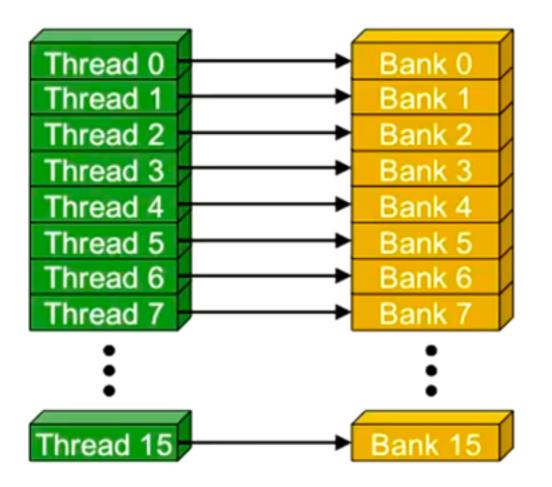
2020

Checking memory

• You should also check where your memory bottleneck are



Memory bank

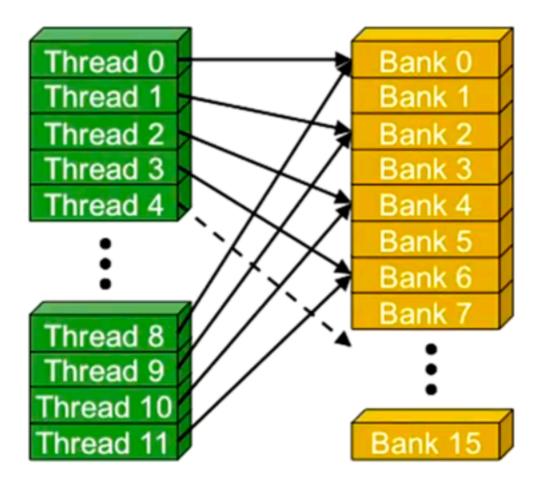


Linear addressing: stride = 1

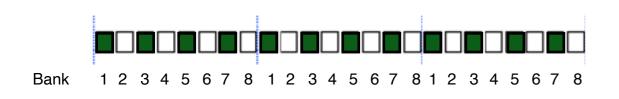
Bank	1 2	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8

- Shared memory use a "Bank" system
 - Each wrap has 32 bank that can provide one data element per cycle
 - The shared memory is dispatched between all those bank
 - A given data can only be given by ONE bank
 - ♦ (&x) % 32
 - So better to use coalesced memory as well

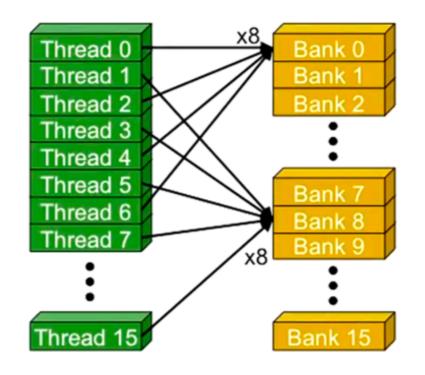
Memory bank



2-way bank conflict: stride = 2



- If you do some striding
 - Bank conflict
 - Increase latency
 - A bank can provide only one value per cycle



8-way bank conflict: stride = 8

Optimization Wrap

- If statement
- Coalesced memory
- Shared memory
 - Memory bank
- Ressource limitation
- Single file

Nice series of tutorial:

- <u>https://developer.nvidia.com/blog/even-easier-introduction-cuda/</u>
 - At the bottom of the page you have a list of quite progressive tutorial
- Nice video on Cuda 5: https://www.youtube.com/watch? v=irvhW7oSNeQ&list=PLGvfHSgImk4aAt3R3XKvUMIv_RFOzSnWz&index=2
- Nice presentation: https://cac-staff.github.io/ summer-school-2018/files/ cuda_day1_summer_school_2018.pdf

Conclusion

- GPU is a high throughput
 - High latency
- Various level of parralelism
 - thread/wrap/block
- Various type of memory
 - register/shared memory/global memory
- Optimization for the hardware is key
 - Coalesced memory ->Array of structure
 - Shared memory