

High Performance Computing – Course 4: CUDA – Programming NVIDIA GPUs

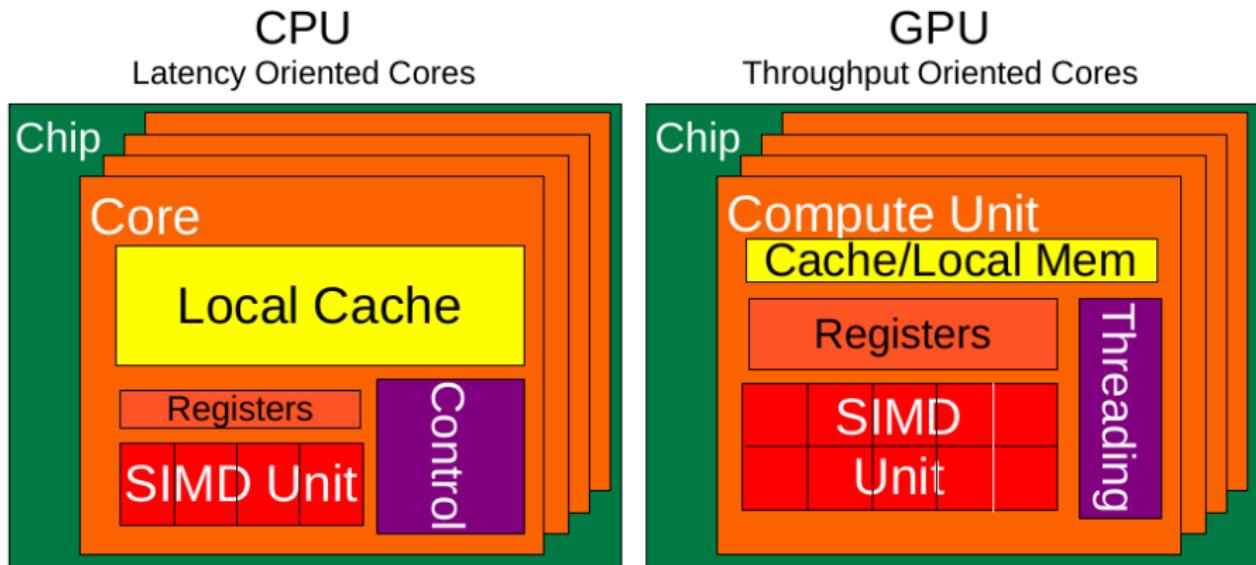
Jonathan Rouzaud-Cornabas

LIRIS / Insa de Lyon – Inria Beagle

References

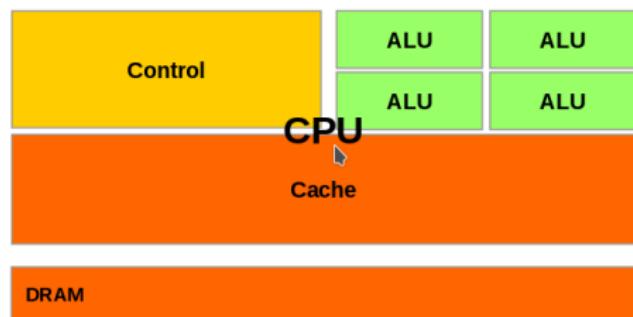
- HPC/GPU courses by Caroline Collange
<https://team.inria.fr/pacap/members/collange/>
- Programming GPU Accelerators with OpenCL by Raymond Namyst and Pierre-André Wacrenier
- NVidia presentation at GTC (2021, 2022)
- HotChips conferences (2020, 2021, 2022)
- PRACE training (NHR@FAU, GENCI, CSCS)
- Satoshi Matsuoka, Modsim Workshop, Seattle, WA, USA

CPU: Latency oriented design



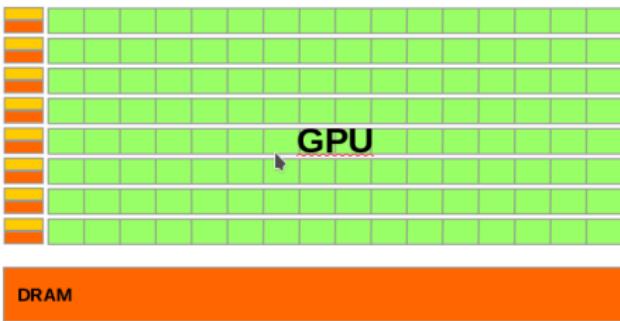
CPU: Latency oriented design

- High clock frequency
- Large caches
 - Convert high latency accesses in memory to low latency accesses in the cache
- Sophisticated control system
 - Branch prediction to reduce latency due to branches
 - Loading data to reduce latency due to data access
- Powerful arithmetic and logic unit (ALU)
 - Reduced operation latency



GPU: Throughput oriented design

- Moderate clock frequency
- Small caches
 - To maximize memory throughput
- Simple control
 - No branch prediction
 - No data loading
- Low consumption arithmetic and logic unit (ALU)
 - Numerous, high latency but strongly pipelined for high throughput
- Requires a very large number of threads for latency to be tolerable



Applications can benefit from both

- CPUs for sequential parts where latency is critical
 - CPUs can be 10+X faster than GPUs for sequential code
- GPUs for parallel parts where throughput is critical
 - GPUs can be 10+X faster than CPUs for parallel code

The winning strategy is to use both

- CPUs for sequential parts where latency is critical
 - CPUs can be 10+X faster than GPUs for sequential code
- GPUs for parallel parts where throughput is critical
 - GPUs can be 10+X faster than CPUs for parallel code

Load distribution

The total execution time of a parallel application is limited by the thread that takes the longest to finish



good



bad!

Global memory bandwidth

Ideal



Reality



Data access conflict: serialization and deadlines

- Massively parallel applications cannot afford serialization
- Contention caused by concurrent access to a critical resource causes serialization



Programming on NVIDIA GPU

Programming the NVIDIA Platform

CPU, GPU, and Network

ACCELERATED STANDARD LANGUAGES

ISO C++, ISO Fortran

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y){ return y + a*x; });

```

```
do concurrent (i = 1:n)
    y(i) = y(i) + a*x(i)
enddo
```

```
import cunumeric as np
...
def saxpy(a, x, y):
    y[:] += a*x
```

INCREMENTAL PORTABLE OPTIMIZATION

OpenACC, OpenMP

```
#pragma acc data copy(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}

#pragma omp target data map(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

PLATFORM SPECIALIZATION

CUDA

```
__global__
void saxpy(int n, float a,
            float *x, float *y) {
    int i = blockIdx.x*blockDim.x +
            threadIdx.x;
    if (i < n) y[i] += a*x[i];
}

int main(void) {
    ...
    cudaMemcpy(d_x, x, ...);
    cudaMemcpy(d_y, y, ...);

    saxpy<<<(N+255)/256,256>>>(...);

    cudaMemcpy(y, d_y, ...);
}
```

ACCELERATION LIBRARIES

Core

Math

Communication

Data Analytics

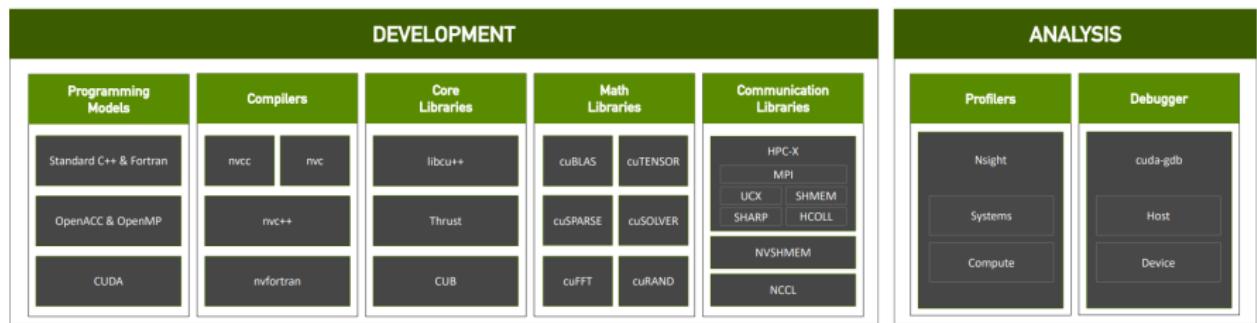
AI

Quantum

Programming on NVIDIA GPU

NVIDIA HPC SDK

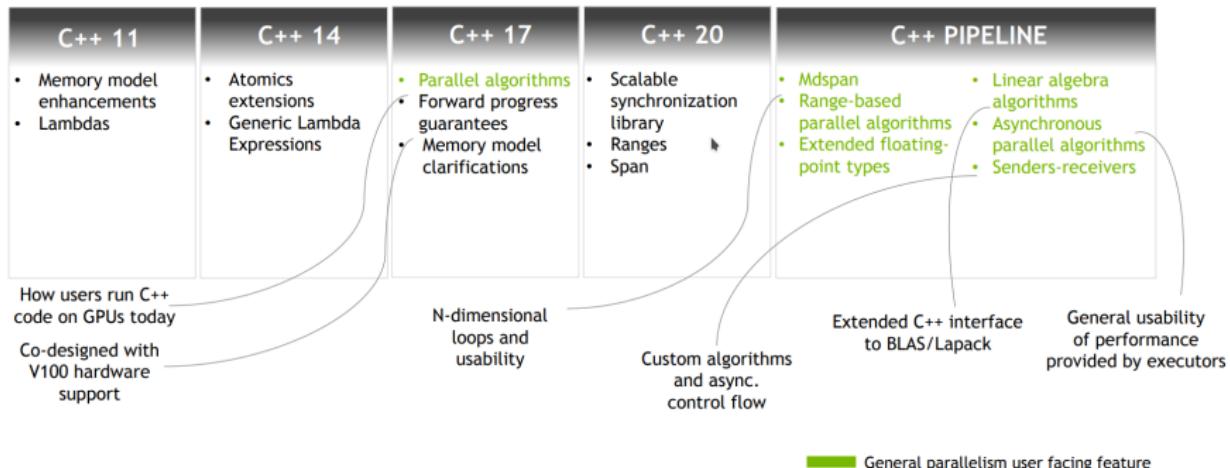
Available at developer.nvidia.com/hpc-sdk, on NGC, via Spack, and in the Cloud



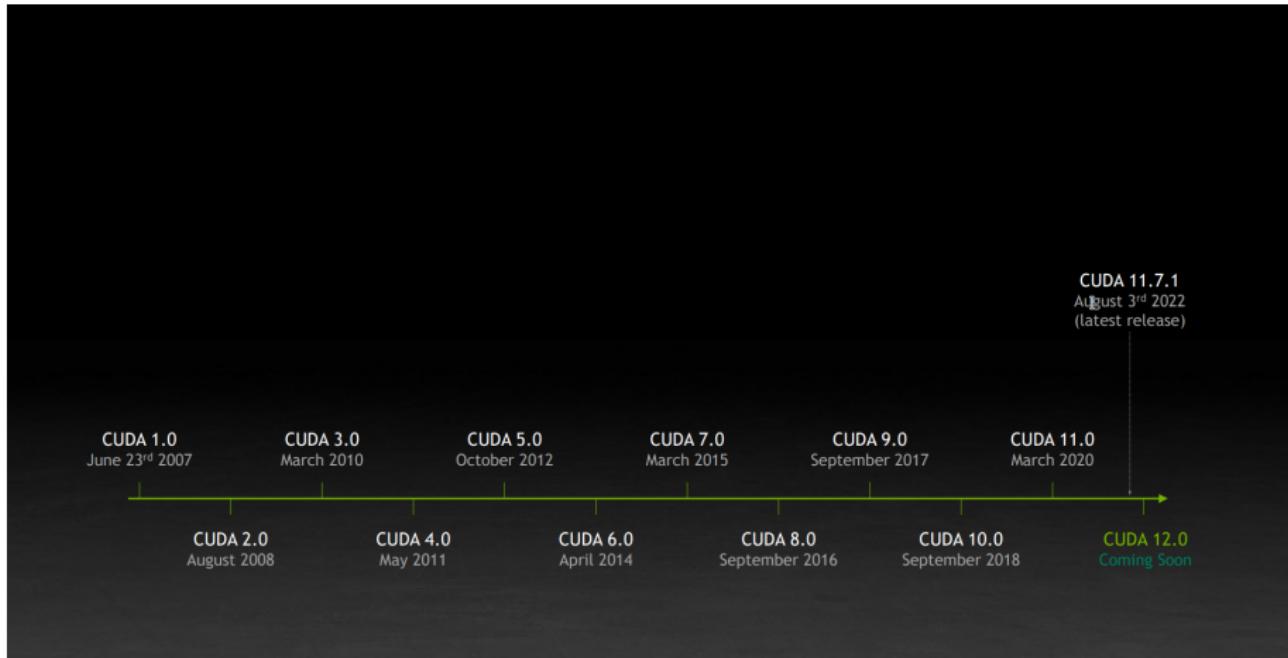
Develop for the NVIDIA Platform: GPU, CPU and Interconnect
 Libraries | Accelerated C++ and Fortran | Directives | CUDA
 x86_64 | Arm | OpenPOWER
 7-8 Releases Per Year | Freely Available

Programming on NVIDIA GPU

Parallelism in C++ Roadmap



A bit of CUDA history



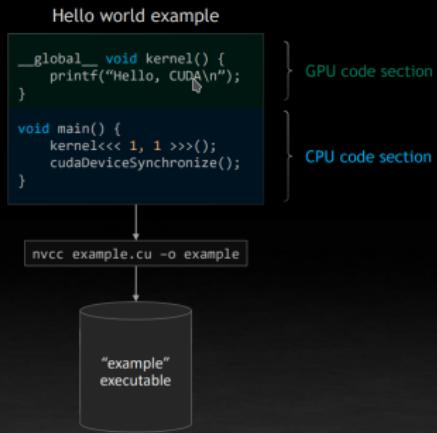
A bit of CUDA history

CUDA MAINTAINS BACKWARDS COMPATIBILITY



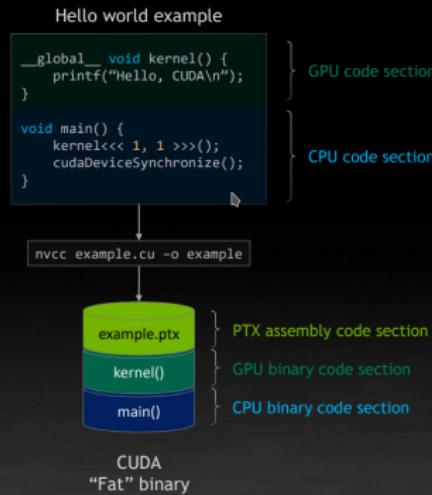
How it works: Quick introduction

ANATOMY OF A CUDA BINARY

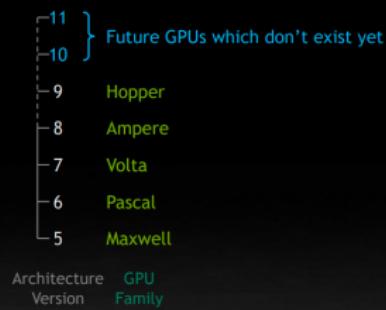


How it works: Quick introduction

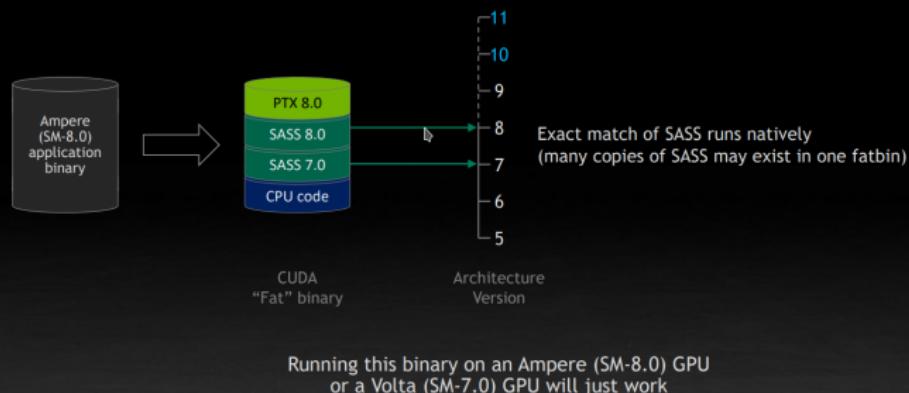
ANATOMY OF A CUDA BINARY



How it works: Quick introduction

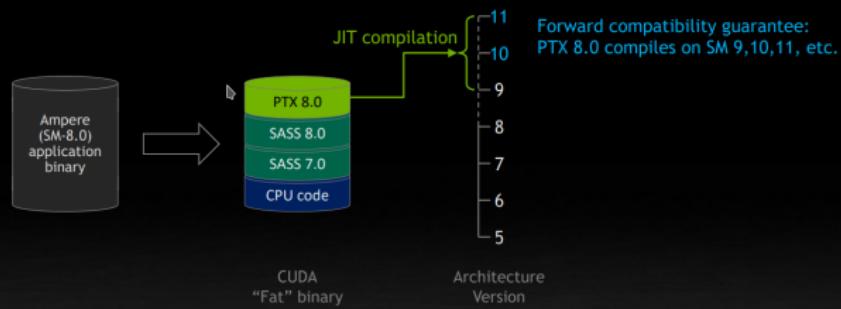


How it works: Quick introduction



How it works: Quick introduction

PORTABILITY VIA PTX JUST-IN-TIME COMPILEMENT



When running on a GPU for which SASS does not exist,
CUDA's built-in PTX compiler recompiles for the new GPU

How it works: Quick introduction

SEMANTIC VERSIONING - STARTING WITH 11.0

Version number MAJOR.MINOR.PATCH

Rules of Semantic Versioning:

1. MAJOR version changes when you make incompatible API changes
2. MINOR version changes when you add functionality in a backwards compatible manner
3. PATCH version changes when you make backwards compatible bug fixes



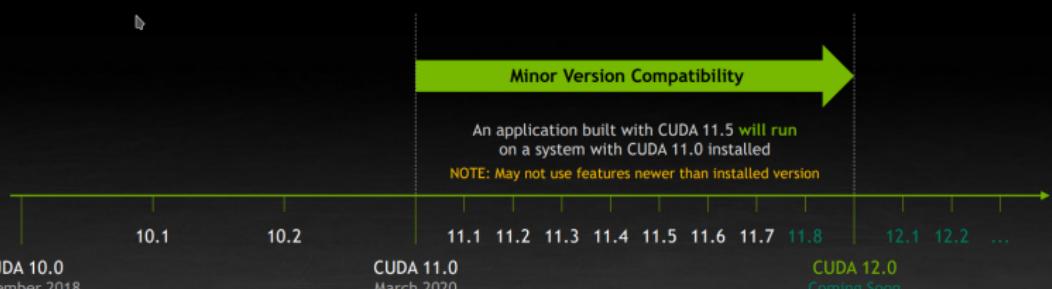
How it works: Quick introduction

SEMANTIC VERSIONING - STARTING WITH 11.0

Version number MAJOR.MINOR.PATCH

Rules of Semantic Versioning:

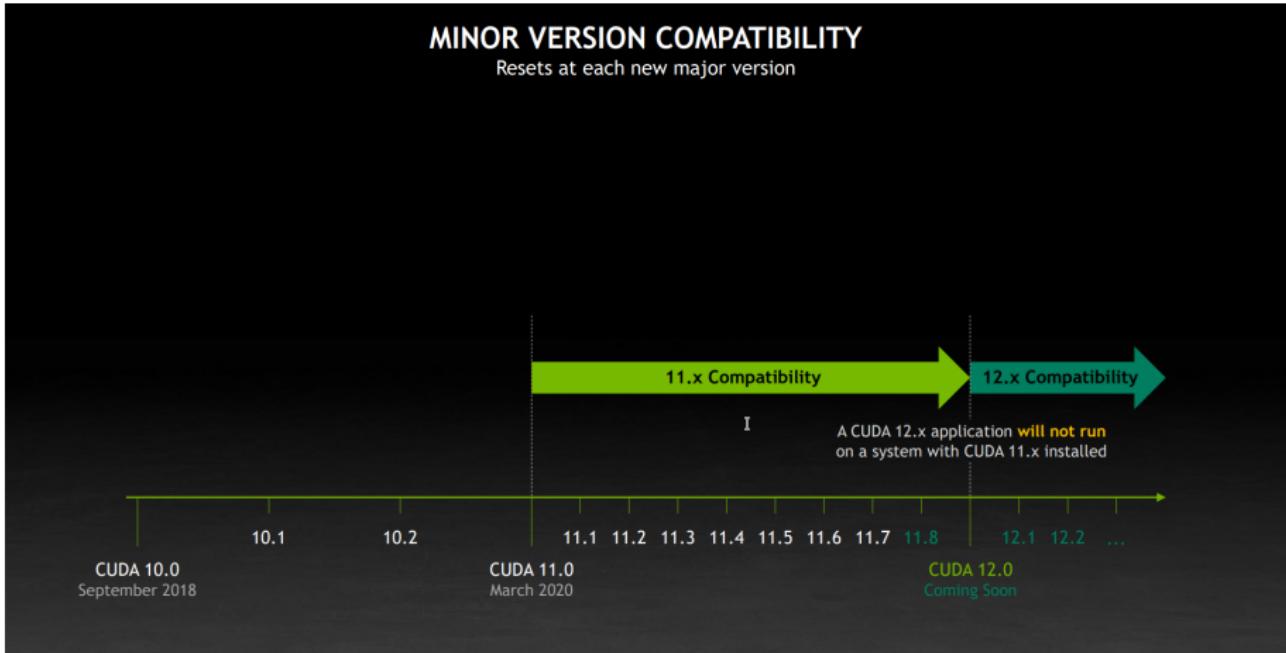
1. MAJOR version changes when you make incompatible API changes
2. MINOR version changes when you add functionality in a backwards compatible manner
3. PATCH version changes when you make backwards compatible bug fixes



How it works: Quick introduction

MINOR VERSION COMPATIBILITY

Resets at each new major version



How it works: Quick introduction

Hello World in CUDA

- CPU “host” code + GPU “device” code

hello.cu:

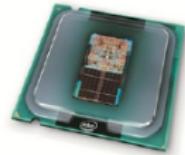
```
_global_ void hello() {  
    printf("Hello World!\n");  
}
```

} Device code

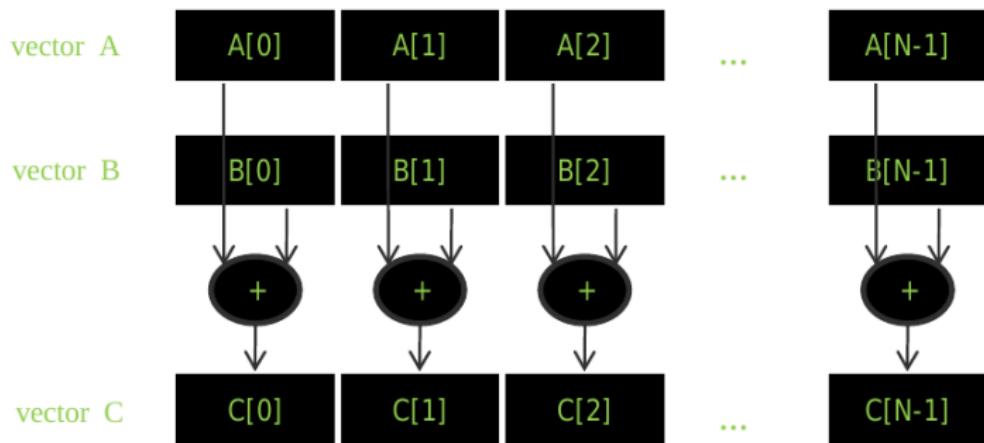


```
int main() {  
    hello<<<1,1>>>();  
    return 0;  
}
```

} Host code



Data parallelism: Addition of vectors

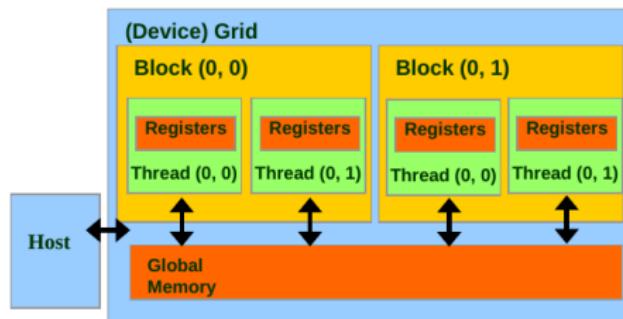


Addition of vectors with C

```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

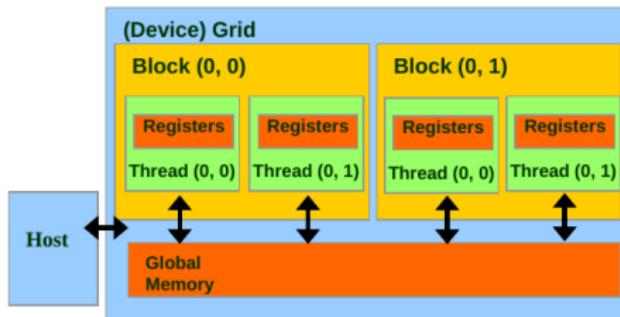
int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```

The basic memories / caches in CUDA



- The device code can:
 - read / write registers (by threads)
 - read / write the whole global memory
- Host code can
 - Transfer data from / to global GPU memory

API of the global memory management in CUDA



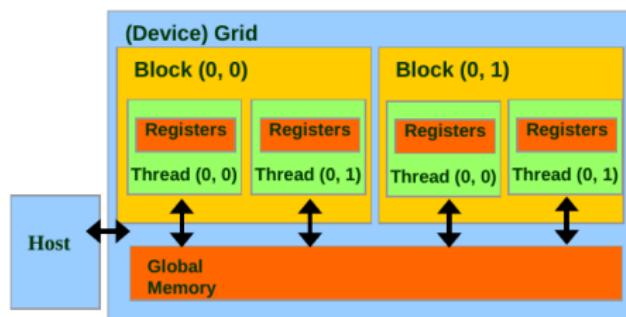
- `cudaMalloc()`

- Allocate an object in the global memory of the GPU
- Two parameters
 - Address of the pointer where the object will be allocated
 - Size (in bytes) of the object to allocate

- `cudaFree()`

- Free the object in the global memory
- A parameter
 - The pointer to the object to be released

Memory transfer API $CPU <> GPU$ in CUDA



`cudaMemcpy()`

- Data transfer to / from GPU memory
- Requires 4 parameters
 - Destination pointer
 - Source pointer
 - Number of bytes to copy
 - Type / Direction of transfer
- ATTENTION: The transfer is asynchronous

Host code: Addition of vectors in CUDA

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);
    // Kernel invocation code - to be shown later
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Remember to check the error return codes

```
cudaError_t err = cudaMalloc((void **) &d_A, size);
if (err != cudaSuccess) {
    printf("%s in %s at line %d\n",
           cudaGetErrorString(err), __FILE__, __LINE__);
    exit(EXIT_FAILURE);
}
```

Example: Device addition vector code

- Compute the sum of 2 vectors $C = A + B$
- Each thread performs a peer addition

```
--global--
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x+blockDim.x*blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

Example: Vector addition host code

- The *ceil* function makes sure that it has enough threads for all the elements of the vectors

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    // d_A, d_B, d_C allocations and copies omitted
    // Run ceil(n/256.0) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n);
}
```

Example: Vector addition host code

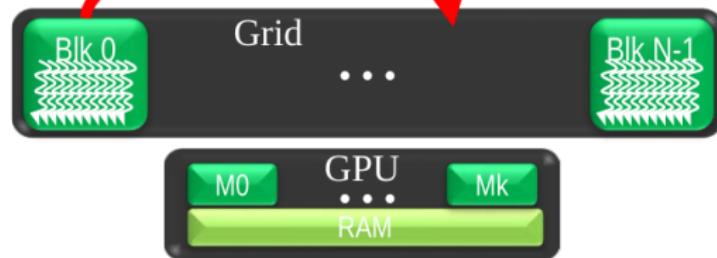
- Another method of calling the kernel

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    dim3 DimGrid((n-1)/256 + 1, 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

Kernel execution in brief

```
__host__
void vecAdd(...)
{
    dim3 DimGrid(ceil(n/256.0),1,1);
    dim3 DimBlock(256,1,1);
    vecAddKernel<<<DimGrid,DimBlock>>>(d_A,d_B
    ,d_C,n);
}
```

```
__global__
void vecAddKernel(float *A,
                  float *B, float *C, int n)
{
    int i = blockIdx.x * blockDim.x
           + threadIdx.x;
    if( i < n ) C[i] = A[i]+B[i];
}
```



Déclarations de fonctions CUDA

	runs on	is called from
<code>--device__ float DeviceFunc()</code>	device	device
<code>--global__ void KernelFunc()</code>	device	host
<code>--host__ float HostFunc()</code>	host	host

- `--global__` defines a kernel, must return void
- `-- device__` and `-- host__` can be used together
- `-- host__` is optional if used alone

Threads and Blocks

CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
<code>__device__ __local__ int LocalVar;</code>	local	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ I int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

- **`__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`**
- **Automatic variables without any qualifier reside in `register`**

BSP Programming Model

PRAM limitations

- PRAM model proposed in 1978
 - ◆ Inspired by SIMD machines of the time
- Assumptions
 - ◆ All processors synchronized every instruction
 - ◆ Negligible communication latency
- Useful as a theoretical model, but far from modern computers



ILLIAC-IV, an early SIMD machine

BSP Programming Model

Modern architectures

- Modern supercomputers are clusters of computers
 - ◆ Global synchronization costs millions of cycles
 - ◆ Memory is distributed
- Inside each node
 - ◆ Multi-core CPUs, GPUs
 - ◆ Non-uniform memory access (NUMA) memory
- **Synchronization** cost at all levels

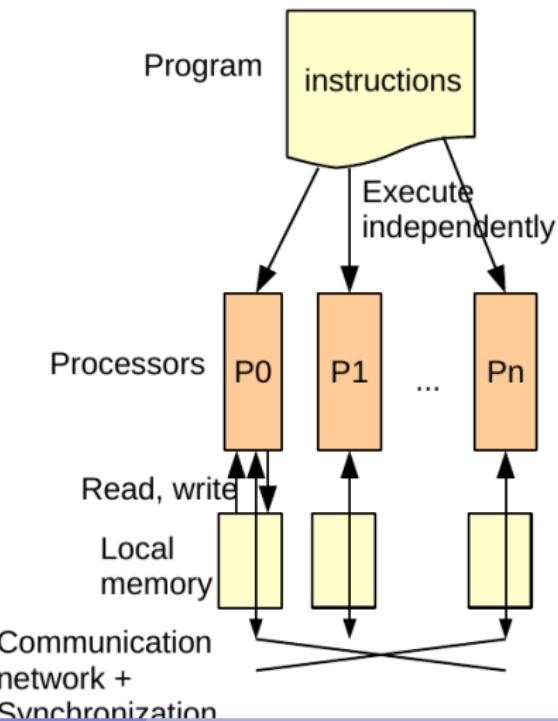


Mare Nostrum, a modern distributed memory machine

BSP Programming Model

Bulk-Synchronous Parallel (BSP) model

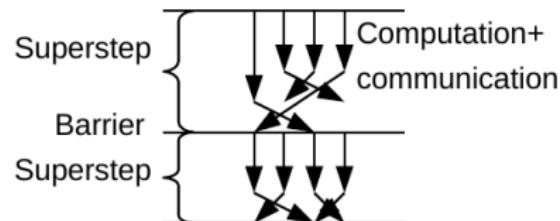
- Assumes distributed memory
 - ◆ But also works with shared memory
 - ◆ Good fit for GPUs too, with a few adaptations
- Processors execute instructions independently
- Communications between processors are **explicit**
- Processors need to **synchronize** with each other



BSP Programming Model

Superstep

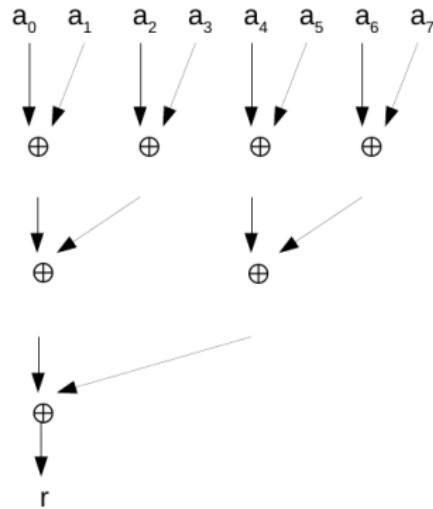
- A program is a sequence of supersteps
- Superstep: each processor
 - ◆ Computes
 - ◆ Sends result
 - ◆ Receive data
- Barrier: wait until all processors have finished their superstep
- Next superstep: can use data received in previous step



BSP Programming Model

Example: reduction in BSP

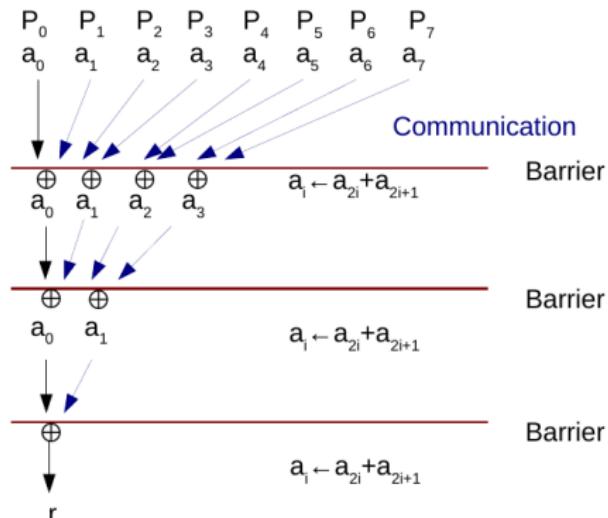
- Start from dependency graph



BSP Programming Model

Reduction: BSP

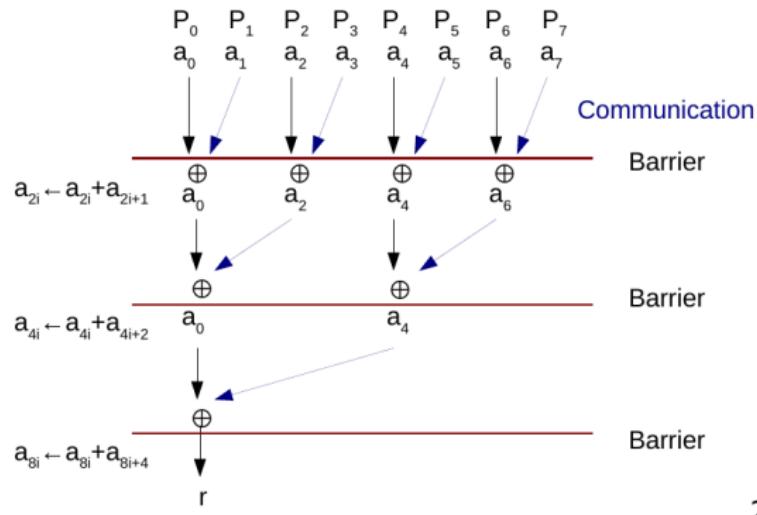
- Add barriers



BSP Programming Model

Reducing communication

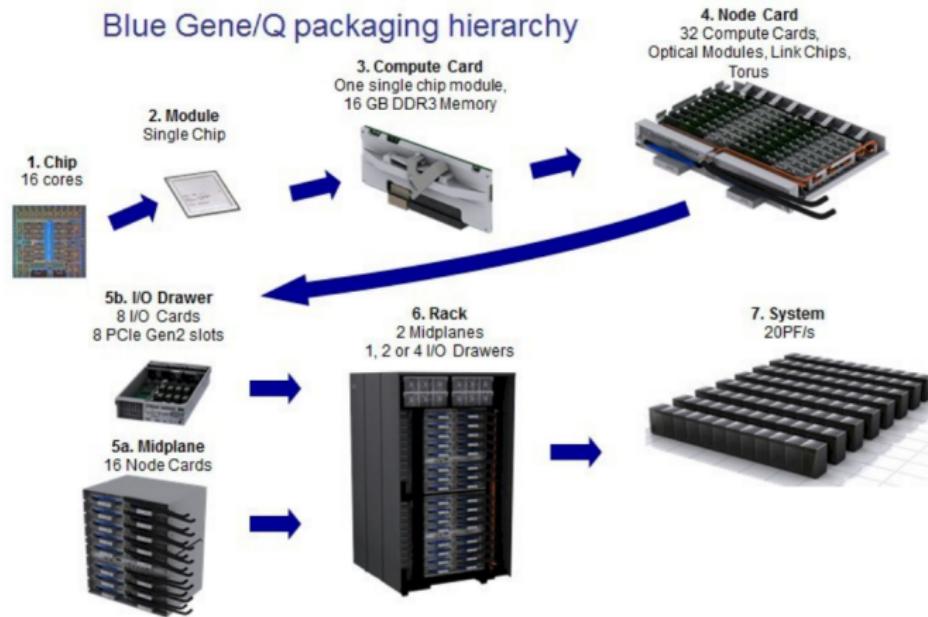
- Data placement matters in BSP
- Optimization: keep left-side operand local



BSP Programming Model

The world is not flat

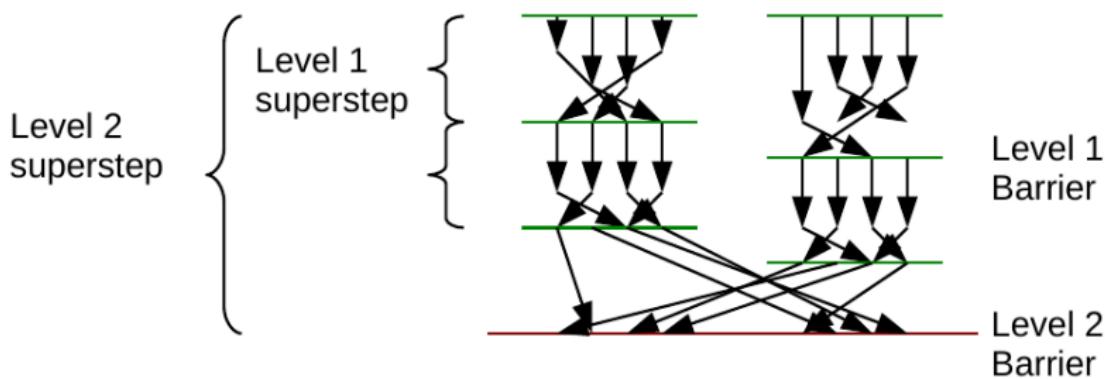
It is hierarchical !



BSP Programming Model

Multi-BSP model

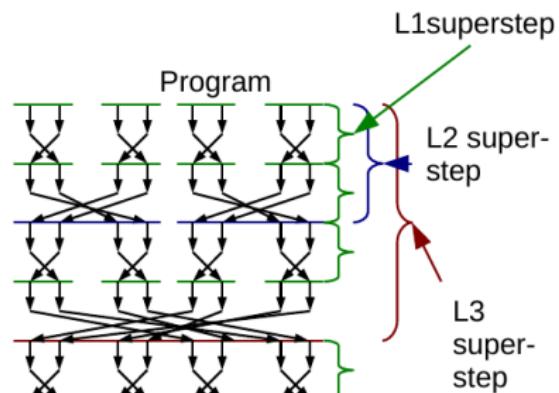
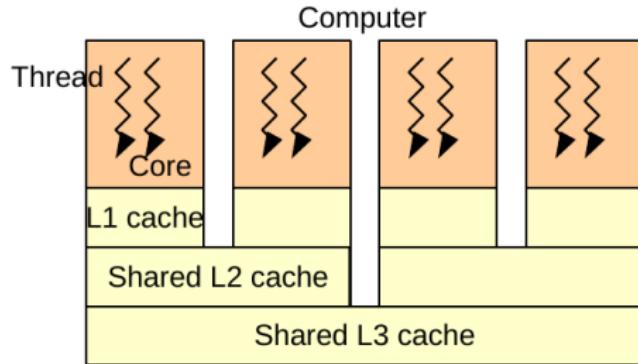
- Multi-BSP: BSP generalization with groups of processors in multiple nested levels



BSP Programming Model

Multi-BSP and multi-core

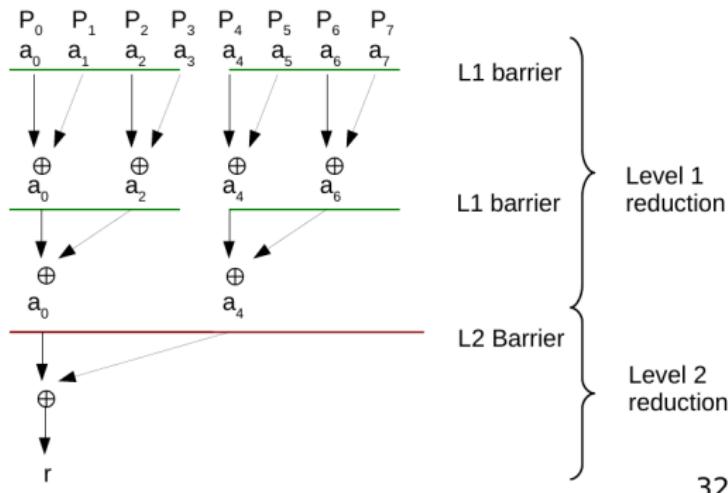
- Minimize communication cost on hierarchical platforms
 - Make parallel program hierarchical too
 - Take thread *affinity* into account



BSP Programming Model

Reduction: multi-BSP

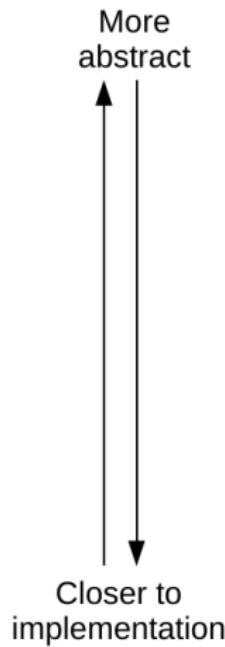
- Break into 2 levels



BSP Programming Model

Recap

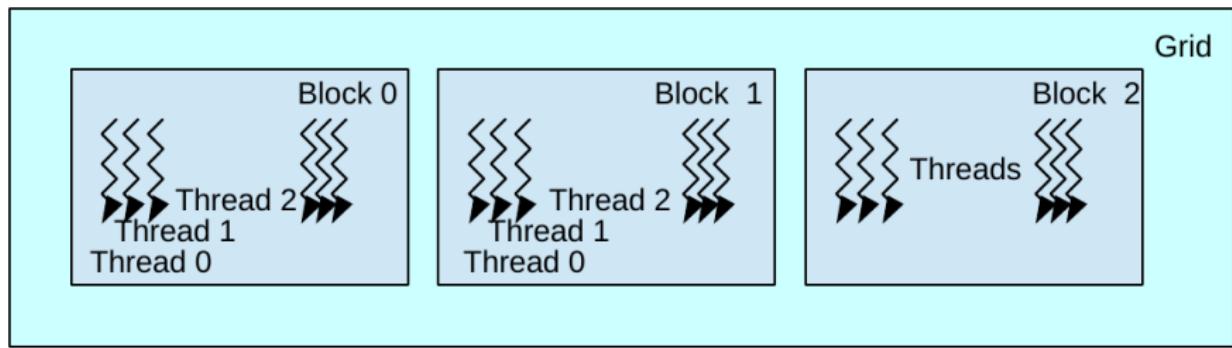
- PRAM
 - ◆ Single shared memory
 - ◆ Many processors in lockstep
- BSP
 - ◆ Distributed memory, message passing
 - ◆ Synchronization with barriers
- Multi-BSP
 - ◆ BSP with multiple scales



Threads and Blocks

Workload: logical organization

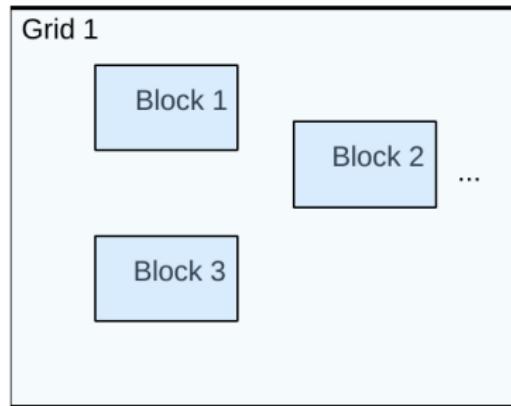
- A kernel is launch on a grid: `my_kernel<<<blocks, threads>>>(...)`
- Two nested levels
 - ◆ Blocks
 - ◆ Threads



Threads and Blocks

Outer level: grid of blocks

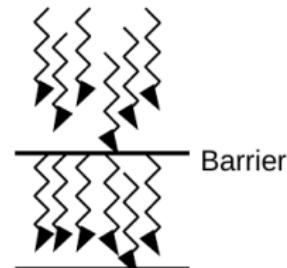
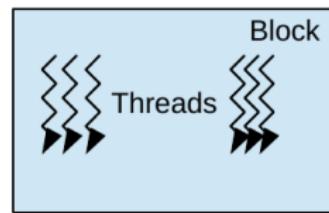
- Blocks also named Concurrent Thread Arrays (CTAs)
- **No communication** between blocks of the same grid
- Practically **unlimited number** of blocks / grid



Threads and Blocks

Inner level: threads

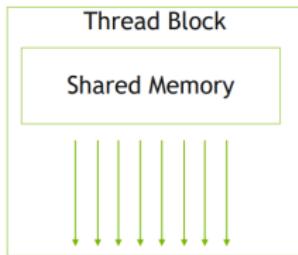
- Blocks contain threads
- All threads in a block
 - ◆ Run on the same SM: they can **communicate**
 - ◆ Run in parallel: they can **synchronize**
- Constraints on **number of threads / block**
 - ◆ Maximum: 512 to 1024 depending on arch
 - ◆ Recommended: at least 64 threads for good performance
 - ◆ Recommended: multiple of the warp size (32)



Threads and Blocks

Thread and Memory Hierarchy

CUDA Thread & Memory Hierarchy pre-Hopper

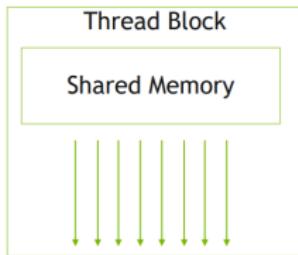


- All threads in a thread block can collaborate using **shared memory**.
- All threads in a thread block **are guaranteed to be co-scheduled** on a Streaming Multiprocessor (SM).

Threads and Blocks

Thread and Memory Hierarchy

CUDA Thread & Memory Hierarchy pre-Hopper

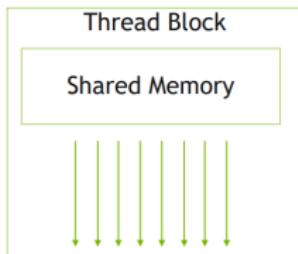


- All threads in a thread block can collaborate using `shared memory`.
- All threads in a thread block `are guaranteed to be co-scheduled` on a Streaming Multiprocessor (SM).
- Threads in a thread block can synchronize / communicate data using
 - `cooperative_groups::this_thread_block.sync();`
OR
`_syncthreads();`
 - `cuda::barrier<thread_scope_block>::arrive()` and `::wait()`

Threads and Blocks

Thread and Memory Hierarchy

CUDA Thread & Memory Hierarchy pre-Hopper

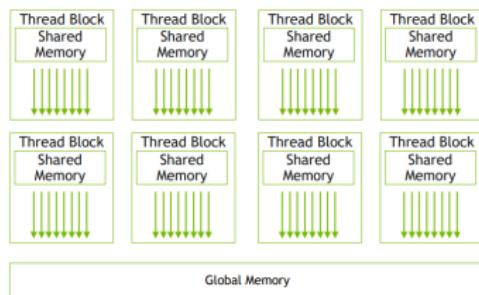


- All threads in a thread block can collaborate using `shared` memory.
- All threads in a thread block are guaranteed to be co-scheduled on a Streaming Multiprocessor (SM).
- Threads in a thread block can synchronize / communicate data using
 - `cooperative_groups::this_thread_block.sync();`
OR
`_syncthreads();`
 - `cuda::barrier<thread_scope_block>::arrive()` and `::wait()`
- Threads in thread block can also perform collectives like `cooperative_groups::reduce()`

Threads and Blocks

Thread and Memory Hierarchy

CUDA Thread & Memory Hierarchy pre-Hopper

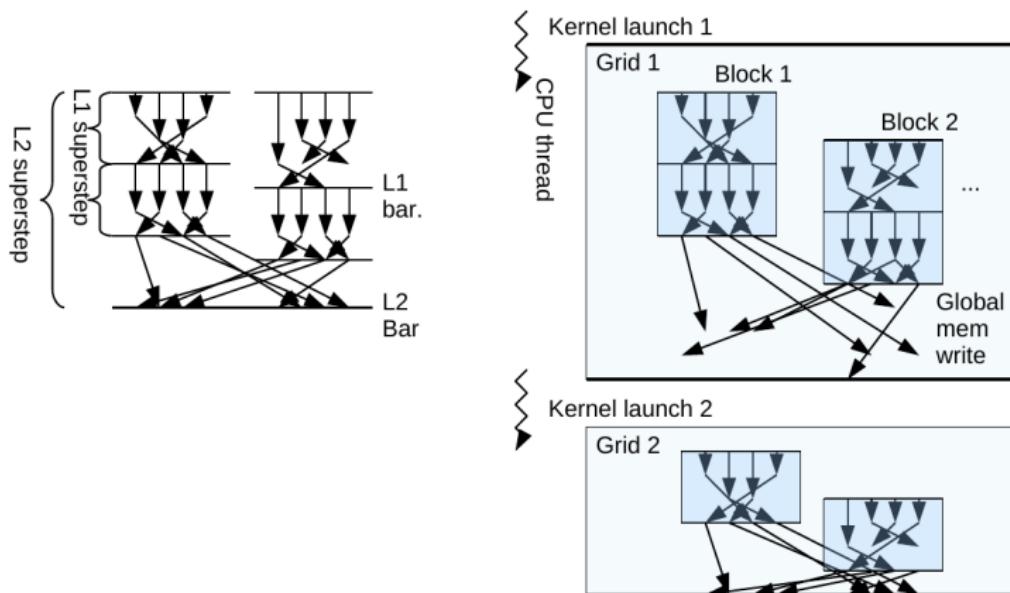


- All thread blocks form the CUDA grid or CUDA kernel
- All thread blocks share global memory to collaborate
- Independent thread blocks can be scheduled out of order to improve occupancy, and hence GPU utilization



Threads and Blocks

Multi-BSP and CUDA

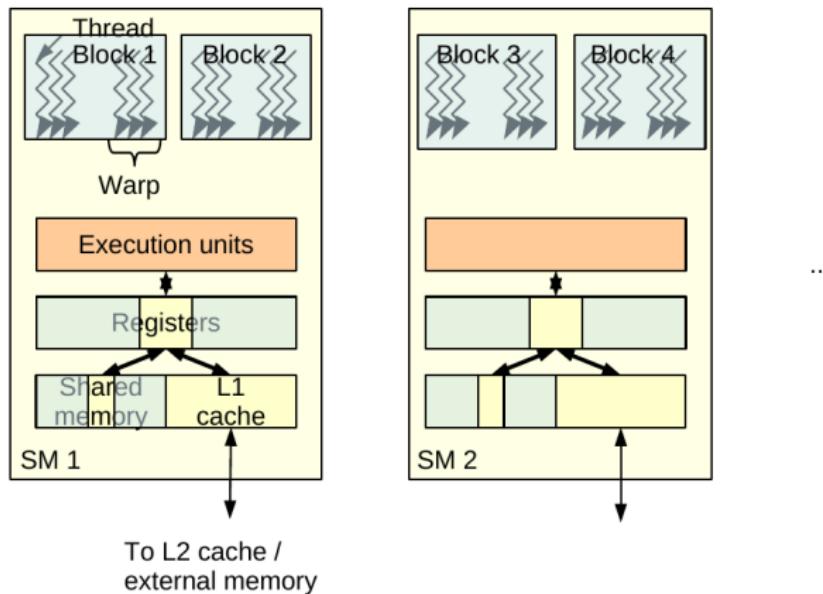


Minor difference: BSP is based on message passing, CUDA on shared memory

38

Threads and Blocks

Mapping blocks to hardware resources

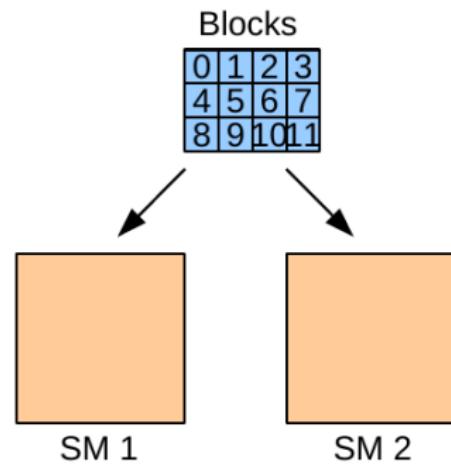


- SM resources are partitioned across blocks

Threads and Blocks

Block scheduling

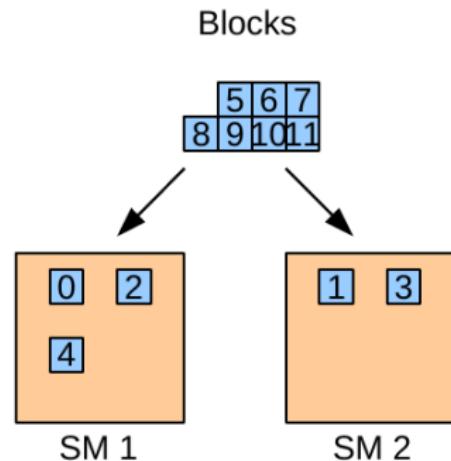
- Blocks may
 - ◆ Run serially or in parallel
 - ◆ Run on the same or different SM
 - ◆ Run in order or out of order
- Should not assume anything on execution order of blocks



Threads and Blocks

Block scheduling

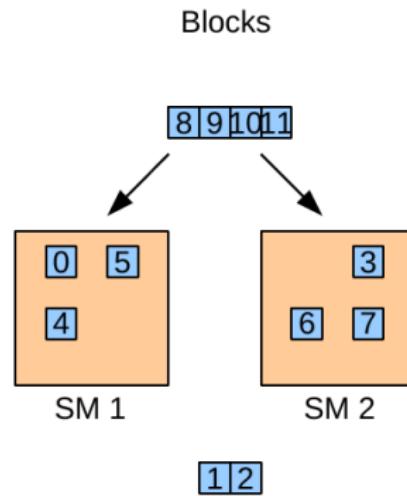
- Blocks may
 - ◆ Run serially or in parallel
 - ◆ Run on the same or different SM
 - ◆ Run in order or out of order
- Should not assume anything on execution order of blocks



Threads and Blocks

Block scheduling

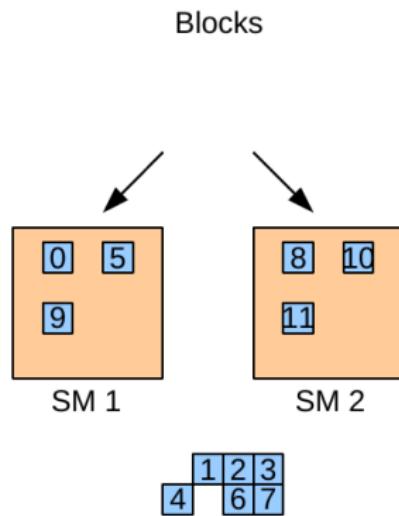
- Blocks may
 - ◆ Run serially or in parallel
 - ◆ Run on the same or different SM
 - ◆ Run in order or out of order
- Should not assume anything on execution order of blocks



Threads and Blocks

Block scheduling

- Blocks may
 - ◆ Run serially or in parallel
 - ◆ Run on the same or different SM
 - ◆ Run in order or out of order
- Should not assume anything on execution order of blocks



Threads and Blocks

Example: vector addition

- Addition example: only 1 thread
 - ◆ Now let's run a parallel computation
- Start with multiple blocks, 1 thread/block
 - ◆ Independent computations in each block
- No communication/synchronization needed

Threads and Blocks

Host code: initialization

- A and B are now arrays: just change allocation size

```
int main()
{
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);
    Initialize(h_A, h_B);

    // Allocate device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    ...
}
```

Threads and Blocks

Host code: kernel and kernel launch

```
__global__ void vectorAdd2(float *A, float *B, float *C)
{
    int i = blockIdx.x;
    C[i] = A[i] + B[i];
}
```

- Launch n blocks of 1 thread each (for now)

Threads and Blocks

Device code

```
__global__ void vectorAdd2(float *A, float *B, float *C)
{
    int i = blockIdx.x;                                
    C[i] = A[i] + B[i];
}
```

Built-in CUDA variable:
in device code only

- Block number i processes element i
- Grid of blocks may have up to 3 dimensions
(blockIdx.x, blockIdx.y, blockIdx.z)
 - ◆ For programmer convenience: no effect on scheduling

Threads and Blocks

Multiple blocks, multiple threads/block

Fixed number of threads / block: here 64

- Host code

```
int threads = 64;           Not necessarily multiple of block size!
int blocks = (numElements + threads - 1) / threads; // Round up

vectorAdd3<<<blocks, threads>>>(d_A, d_B, d_C, numElements);
```

- Device code

```
__global__ void vectorAdd3(const float *A, const float *B, float *C,
    int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;           Global index

    if(i < n) {
        C[i] = A[i] + B[i];
    }                                         Last block may have less work to do
}
```

Thread block may also have up to 3 dimensions: `threadIdx.{x,y,z}`

51

Threads and Blocks

Barriers

- Threads can synchronize inside one block
 - ◆ Wait until all threads in the block have reached the barrier
- In C for CUDA:

```
__syncthreads();
```

- Needs to be called at the same place
for all threads of the block

```
if(tid < 5) {
    ...
} else {
    ...
}
__syncthreads();
```

Correct

```
if(a[0] == 17) {
    __syncthreads();
} else {
    __syncthreads();
}
```

Same condition
for all threads in the block

Correct

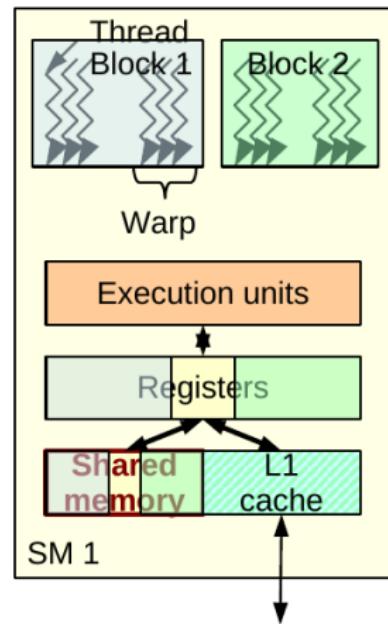
```
if(tid < 5) {
    __syncthreads();
} else {
    __syncthreads();
}
```

Wrong

Threads and Blocks

Shared memory

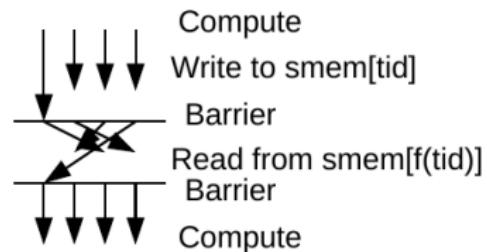
- Fast, software-managed memory
 - ◆ Faster than global memory
- Valid only inside one block
 - ◆ Each block sees its own copy
- Used to exchange data between threads
- Concurrent writes:
one thread wins, but we do not know which one



Threads and Blocks

Thread communication: common pattern

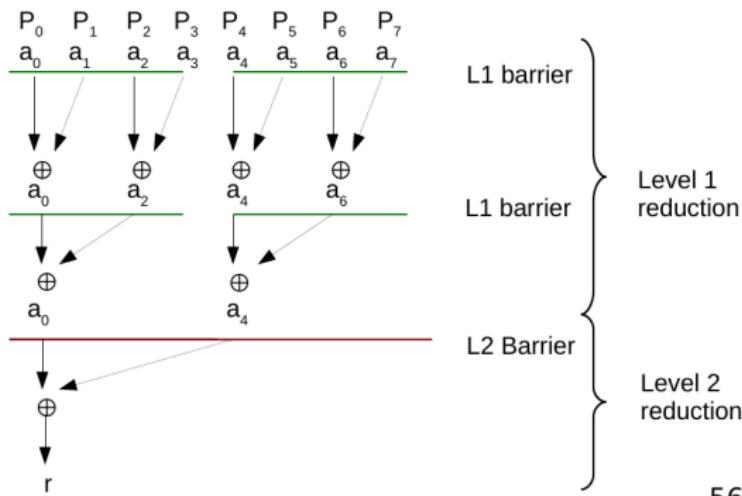
- Each thread writes to its own location
 - ◆ No write conflict
- Barrier
 - ◆ Wait until all threads have written
- Read data from other threads



Threads and Blocks

Example: parallel reduction

- Algorithm for 2-level multi-BSP model



Threads and Blocks

Reduction in CUDA: level 1

```

__global__ void reduce1(float *g_idata, float *g_odata, unsigned int n)
{
    extern __shared__ float sdata[];
    Dynamic shared memory allocation:
    will specify size later

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load from global to shared mem
    sdata[tid] = (i < n) ? g_idata[i] : 0;
    __syncthreads();

    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        int index = 2 * s * tid;

        if(index < blockDim.x) {
            sdata[index] += sdata[index + s];
        }
        __syncthreads();
    }

    // Write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}

```

Threads and Blocks

Quick sanity check to remember

- Each thread block has its own shared memory space
 - ♦ If **blockIdx** appears in the calculation of a **shared** memory index, you are probably doing something wrong!

```
__global__ void reduce1(float *g_idata, float *g_odata, unsigned int n)
{
    extern __shared__ float sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load from global to shared mem
    sdata[tid] = (i < n) ? g_idata[i] : 0;
    __syncthreads();

    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        int index = 2 * s * tid;

        if(index < blockDim.x) {
            sdata[index] += sdata[index + s];
        }
        __syncthreads();
    }

    // Write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

The code snippet shows a reduction operation. Annotations explain memory access patterns:

- A callout points to the line `sdata[tid] = (i < n) ? g_idata[i] : 0;` with the text: "Global memory index may depend on **blockIdx** and **threadIdx**".
- A callout points to the line `if(index < blockDim.x) { sdata[index] += sdata[index + s]; }` with the text: "Shared memory index may depend on **threadIdx**, but never on **blockIdx**".

Threads and Blocks

```
int smemSize = threads * sizeof(float);
reduce1<<<blocks, threads, smemSize>>>(d_idata, d_odata, size);
```



Optional parameter:
Size of dynamic shared memory per block

- Level 2: run reduction kernel again, until we have 1 block left
- By the way, is our reduction operator associative?

Threads and Blocks

A word on floating-point

- Parallel reduction requires the operator to be **associative**
- Is addition associative?
 - On reals: yes, $(a + b) + c = a + (b + c)$
 - On floating-point numbers: **no**
Example with 4 decimal digits:
 $(1.234 + 123.4) - 123.4 = 124.6 - 123.4 = 1.200$
 $1.234 + (123.4 - 123.4) = 1.234 + 0 = 1.234$

$ \begin{array}{r} 1.234 \\ + 123.4 \\ \hline 124.6 \end{array} $	$ \begin{array}{r} 123.4 \\ - 123.4 \\ \hline 000.0 \end{array} $
$ \begin{array}{r} 124.6 \\ - 123.4 \\ \hline 001.200 \end{array} $	$ \begin{array}{r} 000.0 \\ + 1.234 \\ \hline 001.234 \end{array} $

- Consequence: different result depending on thread count

Threads and Blocks

Device functions

- Kernel can call functions
- Need to be marked for GPU compilation

```
_device_ int foo(int i) {  
}
```

- A function can be compiled for both host and device

```
_host_ _device_ int bar(int i) {  
}
```

- Device functions can call device functions
 - Older GPUs do not support recursion

Async and Streams

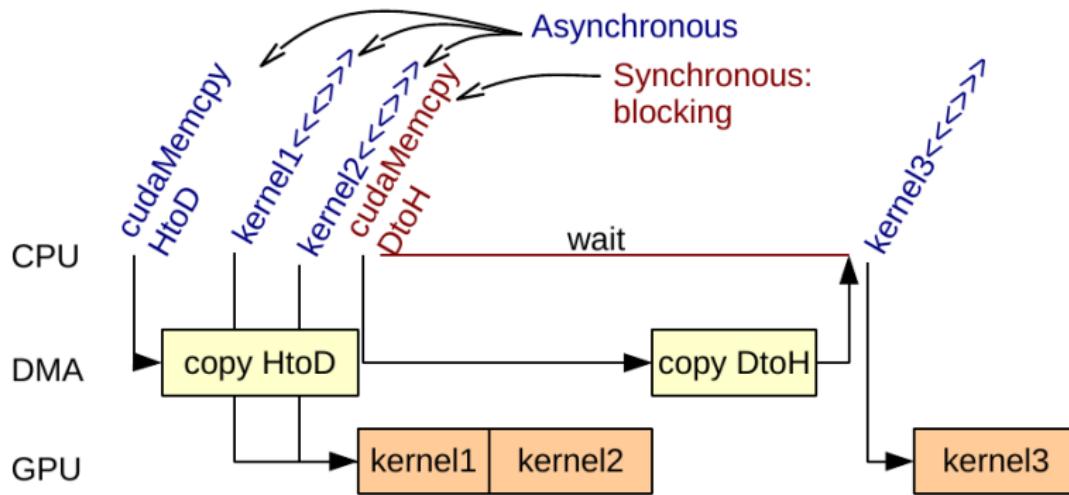
Asynchronous execution

- By default, most CUDA function calls are **asynchronous**
 - ◆ Returns immediately to CPU code
 - ◆ GPU commands are queued and executed in-order
- Some commands are **synchronous** by default
 - ◆ `cudaMemcpy(..., cudaMemcpyDeviceToHost)`
 - ◆ Asynchronous version: `cudaMemcpyAsync`
- Keep it in mind when checking for errors and measuring timing!
 - ◆ Error returned by a command may be caused by an earlier command
 - ◆ Time taken by kernel<<<>> launch is meaningless
- To force synchronization: `cudaDeviceSynchronize()`

Async and Streams

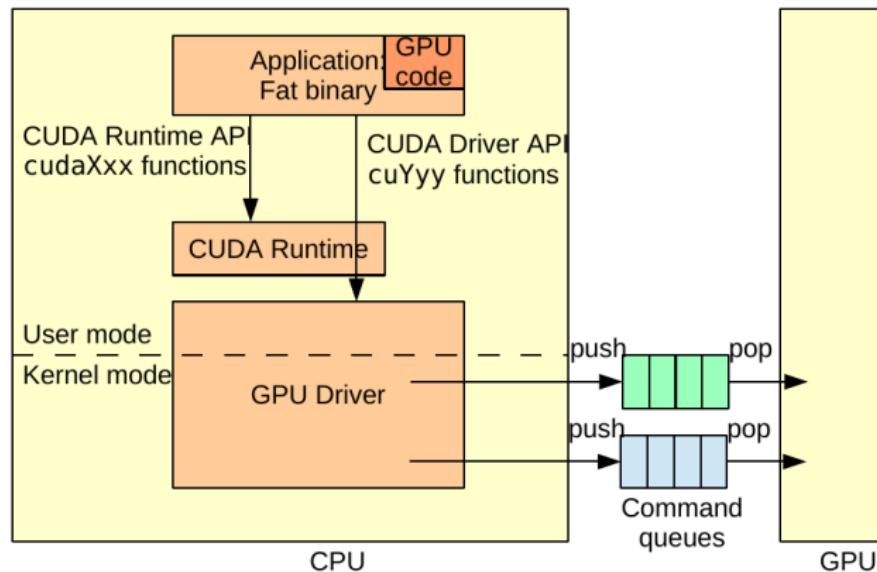
Asynchronous transfers

- Overlap CPU work with GPU work



Async and Streams

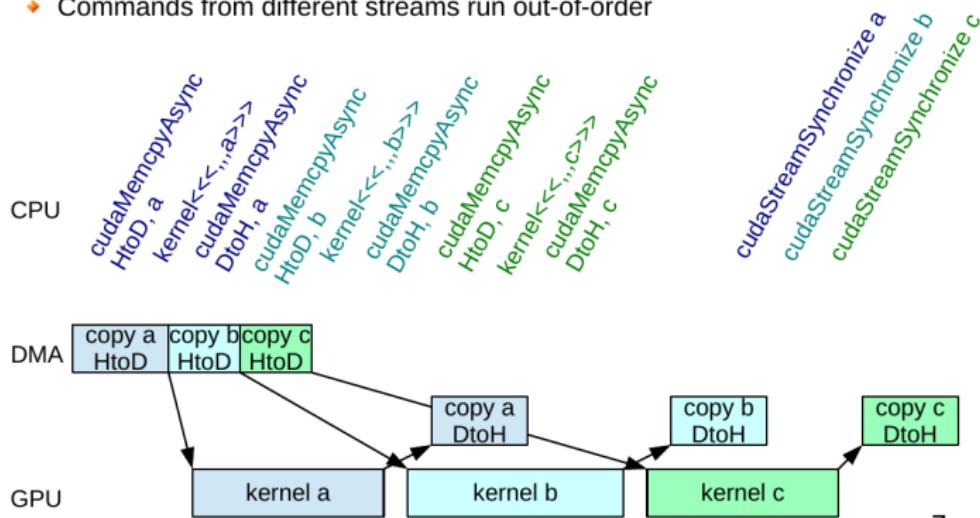
Multiple command queues / streams



Async and Streams

Streams: pipelining commands

- Command queues in OpenCL
 - ◊ Commands from the same stream run in-order
 - ◊ Commands from different streams run out-of-order



Async and Streams

Streams: benefits

- Overlap CPU-GPU communication and computation:
Direct Memory Access (DMA) copy engine
runs CPU-GPU memory transfers in background
 - ◆ Requires page-locked memory
 - ◆ Some Tesla GPUs have 2 DMA engines:
simultaneous send and receive
- Concurrent kernel execution
 - ◆ Start next kernel before previous kernel finishes
 - ◆ Mitigates impact of load imbalance / tail effect

Example

```
Kernel<<<5,,,a>>>
Kernel<<<4,,,b>>>
```

Serial kernel execution

a	block 0	a 3	b 0	b 3
a 1	a 4		b 1	
a 2		b 2		

Concurrent kernel execution

a	block 0	a 3	b 2
a 1	a 4	b 1	
a 2		b 0	b 3

Async and Streams

Page-locked memory

- By default, allocated memory is *pageable*
 - ◆ Can be swapped out to disk, moved by the OS...
- DMA transfers are only safe on *page-locked* memory
 - ◆ Fixed virtual → physical mapping
 - ◆ `cudaMemcpy` needs an intermediate copy:
slower, **synchronous only**
- `cudaMallocHost` allocates page-locked memory
 - ◆ Mandatory when using streams
- Warning: page-locked memory is a limited resource!

Async and Streams

Streams: example

- Send data, execute, receive data

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);

float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);

for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                   size, cudaMemcpyHostToDevice, stream[i]);

    MyKernel <<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);

    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                   size, cudaMemcpyDeviceToHost, stream[i]);
}

for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

Graphs and Events

Events: synchronizing streams

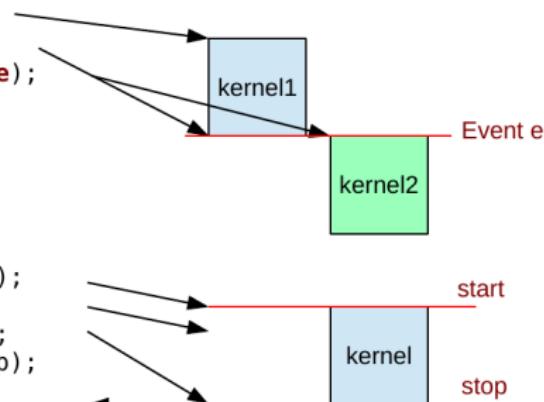
- Schedule synchronization of one stream with another
 - Specify dependencies between tasks

```
cudaEvent_t e;
cudaEventCreate(&e);
kernel1<<<, , a>>>();
cudaEventRecord(e, a);
cudaStreamWaitEvent(b, e);
kernel2<<<, , b>>>();
```

- Measure timing

```
cudaEventRecord(start, 0);
kernel<<<>>>();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

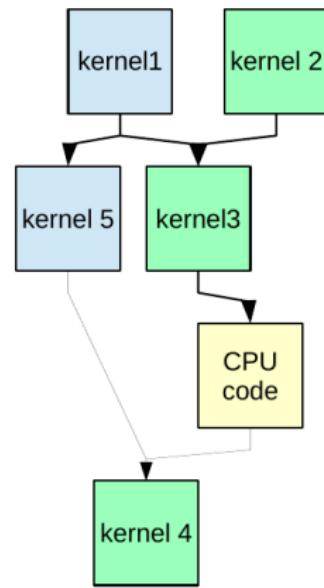
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```



Graphs and Events

Scheduling data dependency graphs

- With streams and events, we can express task dependency graphs
 - Equivalent to threads and events (e.g. semaphores) on CPU
- Example:
 - 2 GPU streams: [a] [b]
and 1 CPU thread: [yellow]
 - Where should we place events?



Graphs and Events

Scheduling data dependency graphs

```

kernel1<<<,,a>>>();
cudaEventRecord(e1, a);

kernel2<<<,,b>>>();

cudaStreamWaitEvent(b, e1);
kernel3<<<,,b>>>();
cudaEventRecord(e2, b);

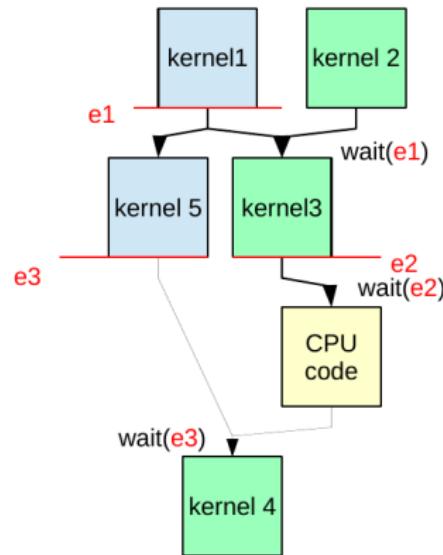
kernel5<<<,,a>>>();
cudaEventRecord(e3, a);

cudaEventSynchronize(e2);

CPU code

cudaStreamWaitEvent(b, e3);
kernel4<<<,,b>>>();

```



New alternative since CUDA 10.0: cudaGraph API

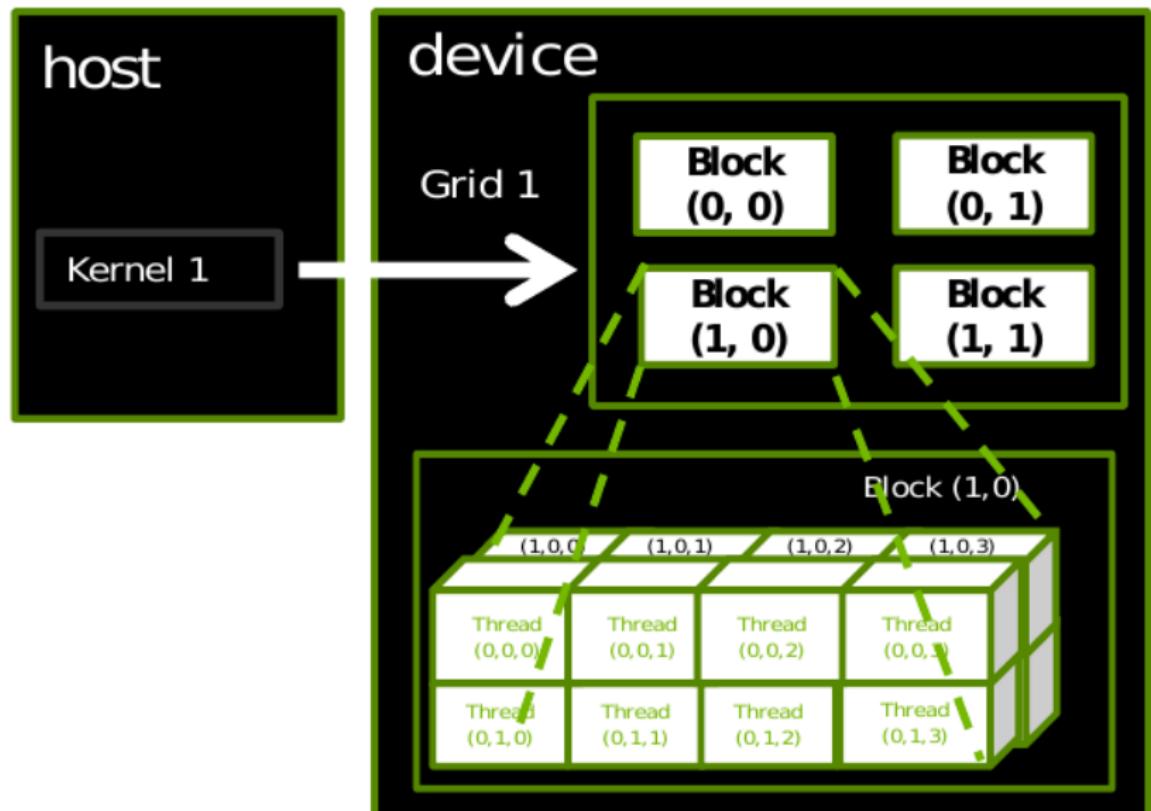
14

Déclarations de fonctions CUDA

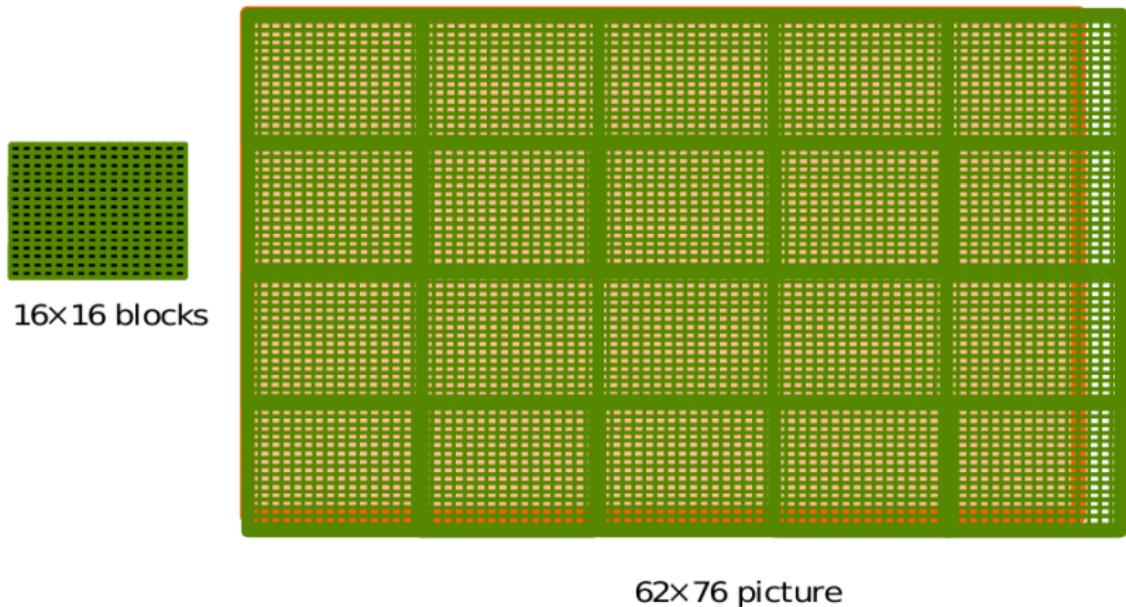
	runs on	is called from
<code>--device__ float DeviceFunc()</code>	device	device
<code>--global__ void KernelFunc()</code>	device	host
<code>--host__ float HostFunc()</code>	host	host

- `--global__` defines a kernel, must return void
- `-- device__` and `-- host__` can be used together
- `-- host__` is optional if used alone

Example of a multi-dimensional grid



Browsing an image with a 2D grid



From an image to an index

M
↓

$$\text{Row} * \text{Width} + \text{Col} = 2 * 4 + 1 = 9$$

M ₀	M ₁	M ₂	M ₃	M ₄	M ₅	M ₆	M ₇	M ₈	M ₉	M ₁₀	M ₁₁	M ₁₂	M ₁₃	M ₁₄	M ₁₅
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------

M
↓



M _{0,0}	M _{0,1}	M _{0,2}	M _{0,3}	M _{1,0}	M _{1,1}	M _{1,2}	M _{1,3}	M _{2,0}	M _{2,1}	M _{2,2}	M _{2,3}	M _{3,0}	M _{3,1}	M _{3,2}	M _{3,3}
------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------



M _{0,0}	M _{0,1}	M _{0,2}	M _{0,3}
M _{1,0}	M _{1,1}	M _{1,2}	M _{1,3}
M _{2,0}	M _{2,1}	M _{2,2}	M _{2,3}
M _{3,0}	M _{3,1}	M _{3,2}	M _{3,3}

Example of a multi-dimensional kernel

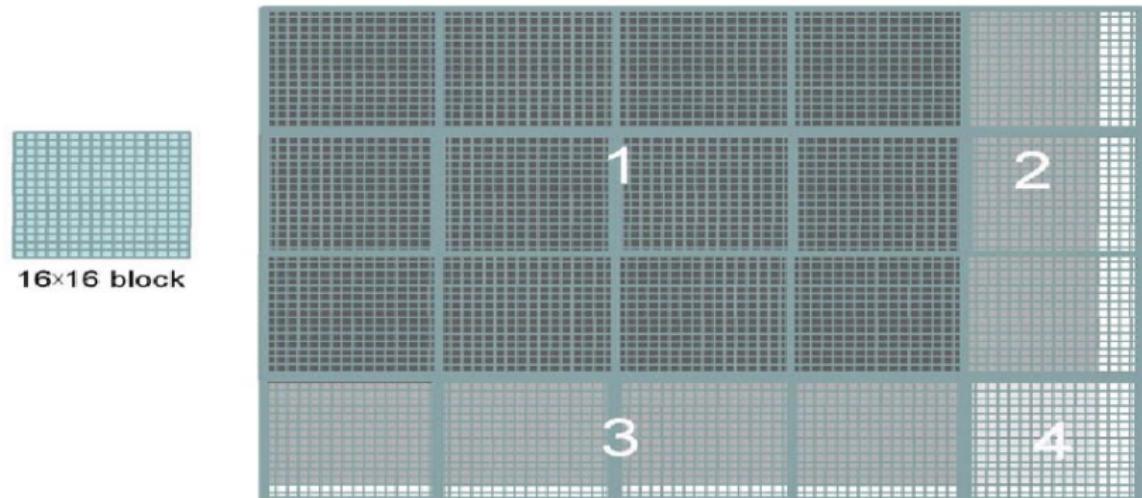
```
--global-- void PictureKernel(float* d_Pin, float* d_Pout,
                           int height, int width)
{
    // Calculate the row # of the d_Pin and d_Pout element
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    // Calculate the column # of the d_Pin and d_Pout element
    int Col = blockIdx.x*blockDim.x + threadIdx.x;
    // each thread computes one element of d_Pout if in range
    if ((Row < height) && (Col < width)) {
        d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
    }
}
```

Multi-dimensional kernel host code

```
// assume that the picture is m n,
// m pixels in y dimension and n pixels in x dimension
// input d_Pin has been allocated on and copied to device
// output d_Pout has been allocated on device

...
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);
dim3 DimBlock(16, 16, 1);
PictureKernel<<<DimGrid , DimBlock>>>(d_Pin , d_Pout , m, n);
...
```

Browse a 62x76 image with 16x16 blocks



Not all threads in a block will follow the same path in the control flow

Converting an RGB image to greyscale

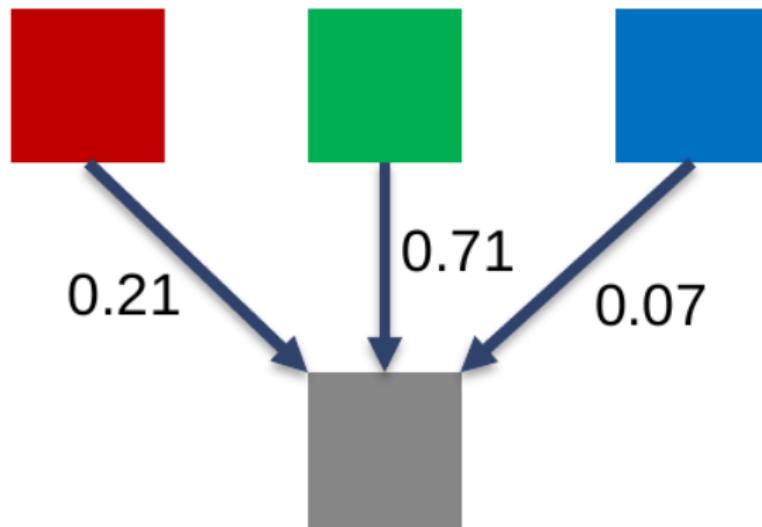


A grayscale image is an image that contains only the intensity of a value for each pixel

Conversion formula

For each pixel (r, g, b) at the index (i, j) :

$$\text{grayPixel}[i, j] = 0.21 * r + 0.71 * g + 0.07 * b$$



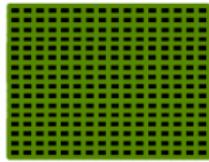
Conversion code

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
    unsigned char * rgblImage, int width, int height) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if (x < width && y < height) {
        // get 1D coordinate for the grayscale image
        int grayOffset = y*width + x;
        // one can think of the RGB image having
        // CHANNEL times columns than the gray scale image
        int rgbOffset = grayOffset*CHANNELS;
        unsigned char r = rgblImage[rgbOffset]; // red value for pixel
        unsigned char g = rgblImage[rgbOffset + 2]; // green value for pixel
        unsigned char b = rgblImage[rgbOffset + 3]; // blue value for pixel
        // perform the rescaling and store it
        // We multiply by floating point constants
        grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

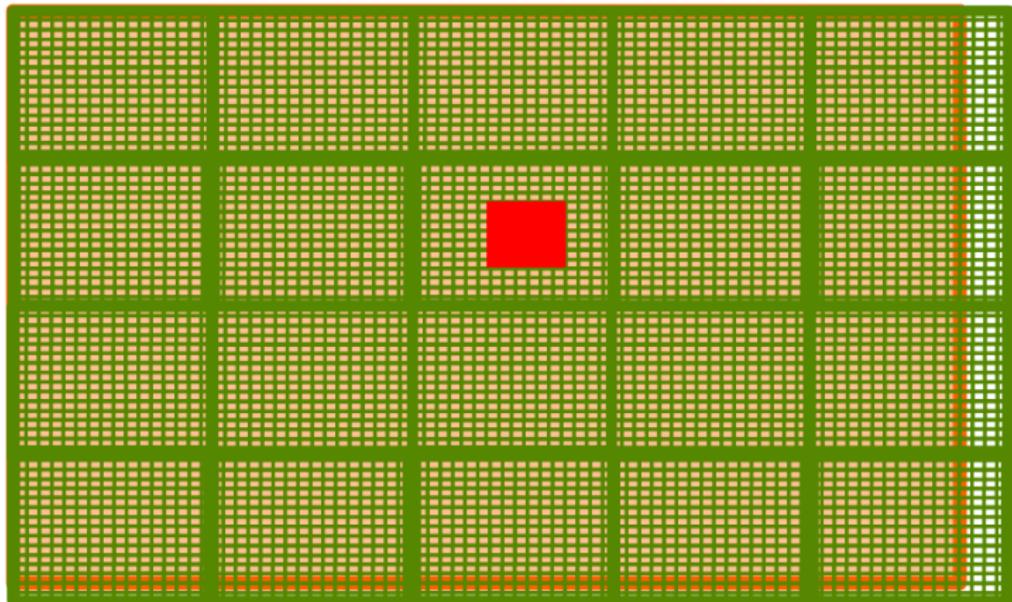
Image blur



Blurry box



Pixels
processed
by a thread
block



Blurring code

```
--global--
void blurKernel(unsigned char * in, unsigned char * out, int w,
{
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    if (Col < w && Row < h) {
        ... // Rest of our kernel
    }
}
```

Bluring code

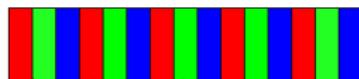
```
--global--
void blurKernel(unsigned char * in, unsigned char * out,
                 int w, int h) {
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    if (Col < w && Row < h) {
        int pixVal = 0;
        int pixels = 0;
        for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
            for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {
                int curRow = Row + blurRow;
                int curCol = Col + blurCol;
                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                    pixVal += in[curRow * w + curCol];
                    pixels++;
                }
            }
        }
        out[Row * w + Col] = (unsigned char)(pixVal / pixels);
    }
}
```

Data structure

Array of structures (AoS)

- Programmer-friendly memory layout
 - ◆ Group data logically
- Memory accesses not coalesced
 - ◆ Bad performance on GPU

```
struct Pixel {  
    float r, g, b;  
};  
Pixel image_AoS[480][640];
```



```
kernel void luminance(Pixel img[][],  
                      float luma[][],  
                      int x=tid.x; int y=tid.y;  
                      luma[y][x]=.59*img[y][x].r  
                        + .11*img[y][x].g  
                        + .30*img[y][x].b;  
}
```

Data structure

Structure of Arrays (SoA)

- Transpose the data structure
 - Group together similar data for different threads
- Benefits from memory coalescing
 - Best performance on GPU

```
struct Image {  
    float R[480][640];  
    float G[480][640];  
    float B[480][640];  
};  
Image image_SoA;
```

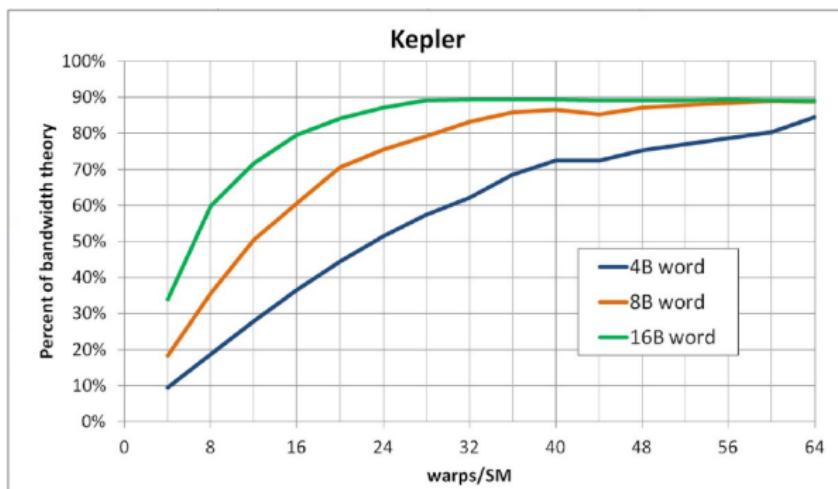


```
kernel void luminance(Image img,  
                      float luma[][]){  
    int x=tid.x; int y=tid.y;  
    luma[y][x]=.59*img.R[y][x]  
    + .11*img.G[y][x]  
    + .30*img.B[y][x];
```

Data structure

Vector loads

- We can load more data at once with vector types
 - ◆ float2, float4, int2, int4...
 - ◆ More memory parallelism
 - ◆ Allows to reach peak throughput with fewer threads



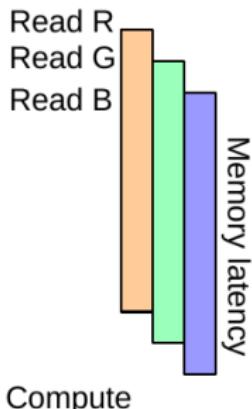
Source: Paulius Micikevicius, GPU Performance Analysis and Optimization. GTC 2012.

Data structure

Multiple outstanding loads

- Multiple independent loads from the same thread can be pipelined
 - ◆ More memory parallelism
 - ◆ Peak throughput with yet fewer threads

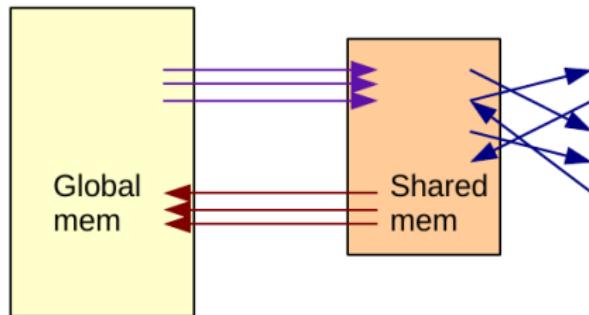
```
__global__ void luminance(Image img,
    float luma[][]) {
    int x=threadIdx.x, y=threadIdx.y;
    luma[y][x]=.59*img.R[y][x]
        + .11*img.G[y][x]
        + .30*img.B[y][x];
}
```



Data structure

Buffer accesses through shared memory

- Global memory accesses are the most expensive
 - ◆ Focus on optimizing global memory accesses
- Strategy: use shared memory as a temporary buffer



1. Load with regular accesses
2. Read and write shared memory with original pattern
3. Store back to global memory with regular accesses

Data structure

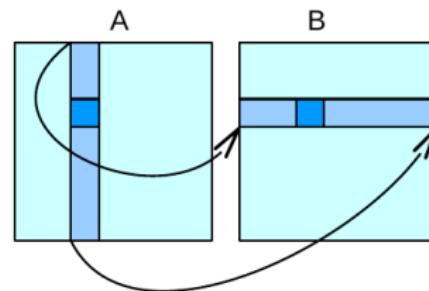
Example: matrix transpose

- $B = A^T$
- Naive algorithms
 - ◆ Option 1

Thread i, j :
 $B[j, i] = A[i, j]$

◆ Option 2

Thread i, j :
 $B[i, j] = A[j, i]$



- Which one is better?
 - ◆ What is the problem?

Data structure

Example: matrix transpose

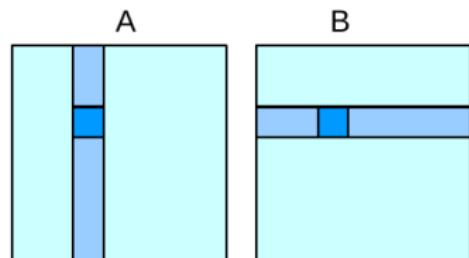
- $B = A^T$
- Naive algorithms
 - ◆ Option 1

Thread i, j :
 $B[j, i] = A[i, j]$

Coalesced Non-coalesced

◆ Option 2

Thread i, j :
 $B[i, j] = A[j, i]$



- Both are equally bad
 - ◆ Access to one array is non-coalesced

Data structure

Matrix transpose using shared memory

- Split matrices in blocks
- Load the block in shared memory
- Transpose in shared memory
- Write the block back

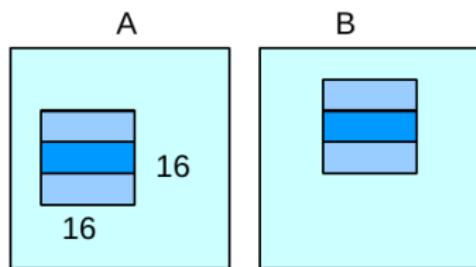
Example with 16×16 blocks

Block bx,by, Thread tx,ty:

$a[ty,tx]=A[by*16+ty,bx*16+tx]$
Syncthreads

$b[ty,tx]=a[tx,ty]$
Syncthreads

$B[by*16+ty,bx*16+tx]=b[ty,tx]$



Coalesced

Data structure

Matrix transpose using shared memory

- Split matrices in blocks
- Load the block in shared memory
- Transpose in shared memory
- Write the block back

Example with 16×16 blocks

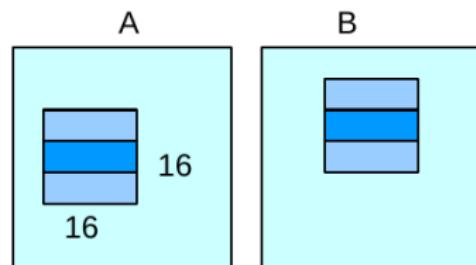
Block bx,by, Thread tx,ty:

$a[ty,tx]=A[by*16+ty,bx*16+tx]$

Syncthreads

$b_local=a[tx,ty]$

$B[by*16+ty,bx*16+tx]=b_local$



Same location read and written:
→ local variable, per thread

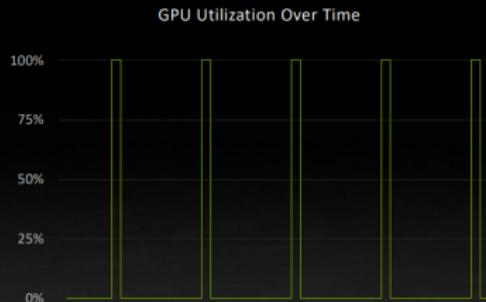
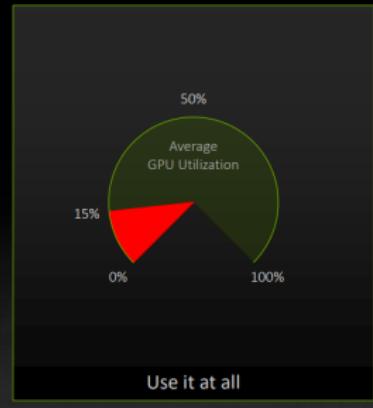
Data structure

Objection

- Isn't it just moving the problem to shared memory?
- Yes: shared memory has access restrictions too
- But
 - ◆ Shared memory is much faster, even for irregular accesses
 - ◆ We can optimize shared memory access patterns too

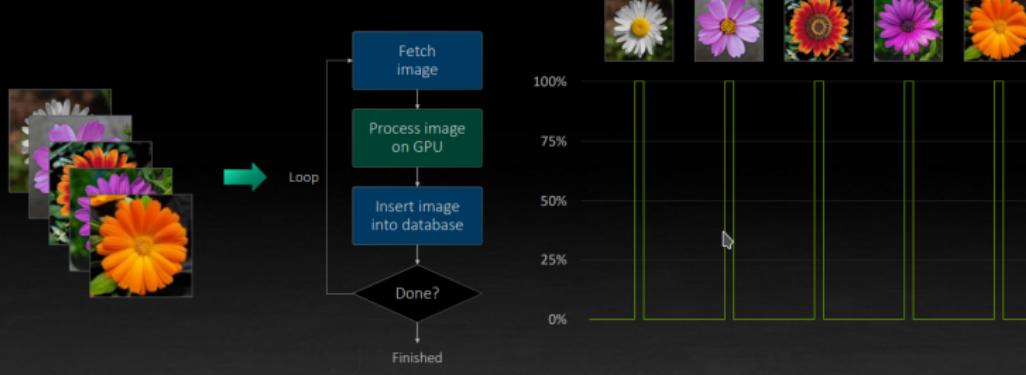
Using (effectively) CUDA core

KEEPING BUSY



Using (effectively) CUDA core

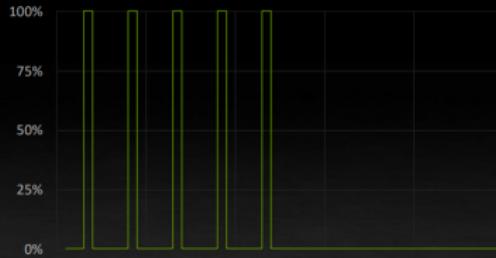
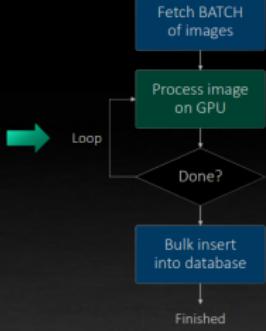
KEEPING BUSY



Using (effectively) CUDA core

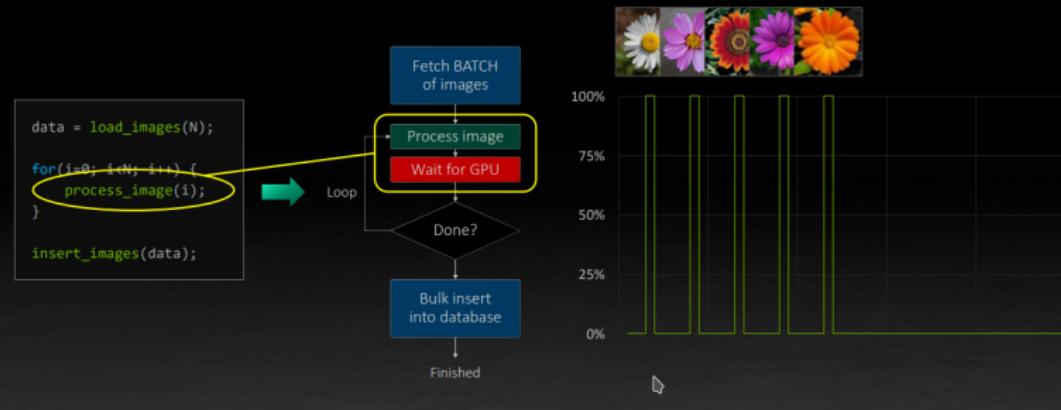
STILL NOT ALL THAT BUSY

```
data = load_images(N);
for(i=0; i<N; i++) {
    process_image(i);
}
insert_images(data);
```



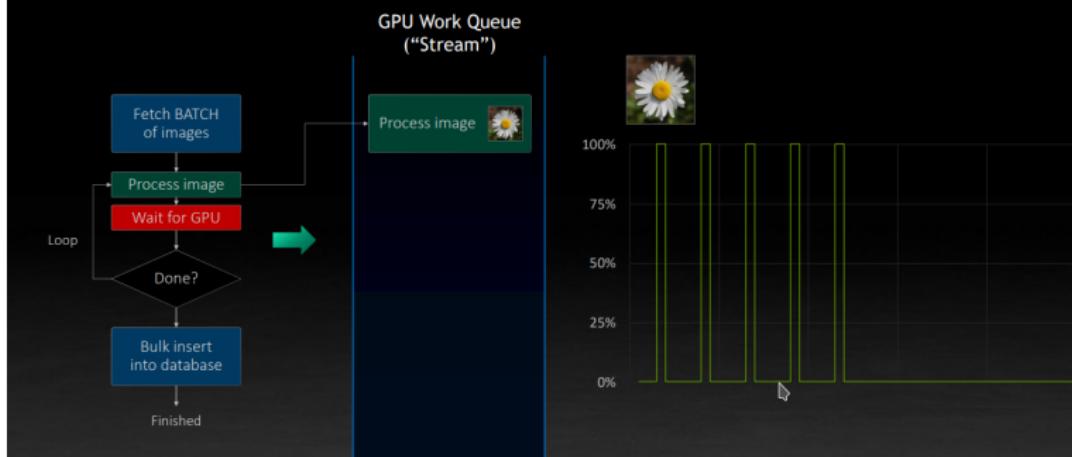
Using (effectively) CUDA core

SYNCHRONOUS EXECUTION



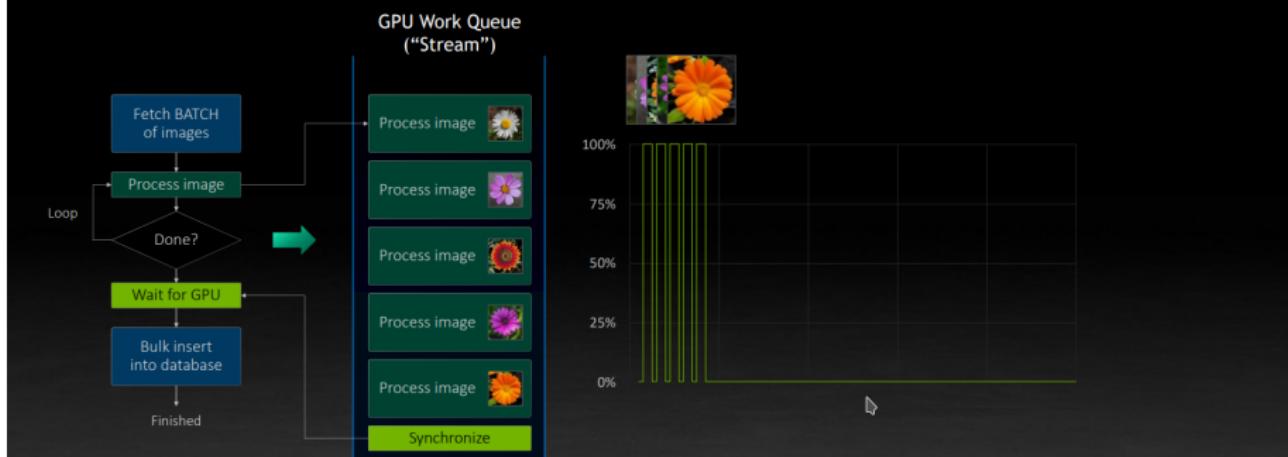
Using (effectively) CUDA core

SYNCHRONOUS EXECUTION



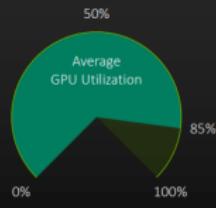
Using (effectively) CUDA core

ASYNCHRONOUS EXECUTION

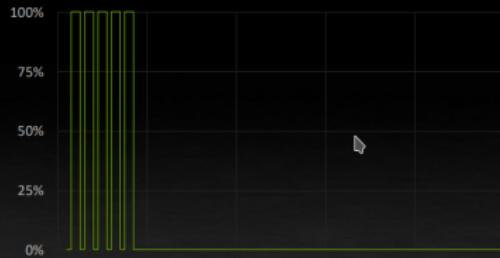


Using (effectively) CUDA core

ASYNCHRONOUS EXECUTION



Keeping the GPU busy



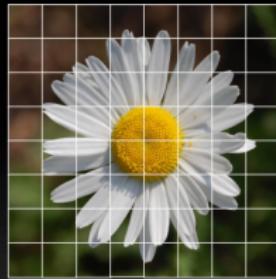
Using (effectively) CUDA core

START WITH SOME WORK TO PROCESS



Using (effectively) CUDA core

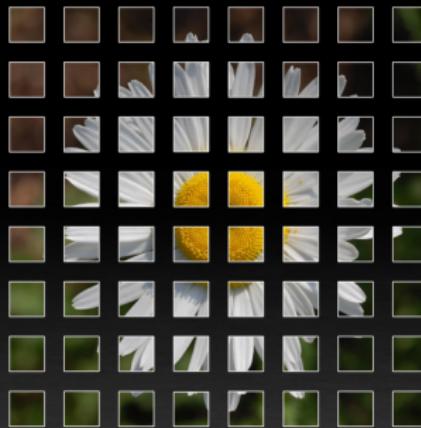
DIVIDE INTO A SET OF EQUAL-SIZED BLOCKS: THIS IS THE “GRID” OF WORK



Using (effectively) CUDA core

EACH BLOCK WILL NOW BE PROCESSED INDEPENDENTLY

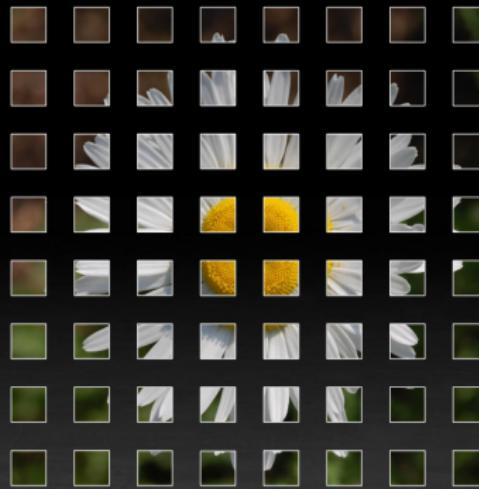
CUDA does not guarantee the order of execution and you cannot exchange data between blocks



Using (effectively) CUDA core

EACH BLOCK WILL NOW BE PROCESSED INDEPENDENTLY

CUDA does not guarantee the order of execution and you cannot exchange data between blocks



Using (effectively) CUDA core

EVERY BLOCK GET PLACED ONTO AN SM

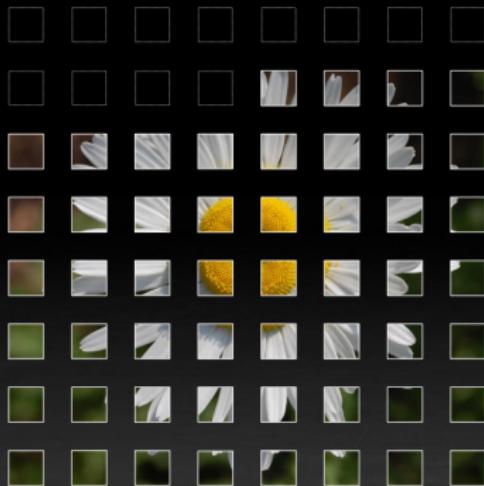
CUDA does not guarantee the order of execution and you cannot exchange data between blocks



Using (effectively) CUDA core

EVERY BLOCK GET PLACED ONTO AN SM

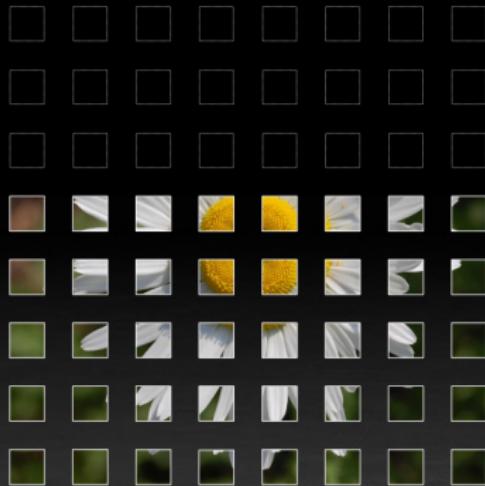
CUDA does not guarantee the order of execution and you cannot exchange data between blocks



Using (effectively) CUDA core

BLOCKS CONTINUE TO GET PLACED UNTIL EACH SM IS “FULL”

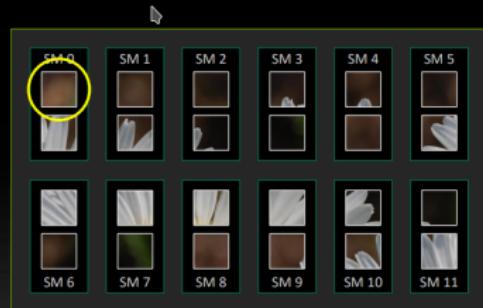
When a block completes its work and exits, a new block is placed in its spot until the whole grid is done



Using (effectively) CUDA core

BLOCKS CONTINUE TO GET PLACED UNTIL EACH SM IS “FULL”

When a block completes its work and exits, a new block is placed in its spot until the whole grid is done



Using (effectively) CUDA core

BLOCKS CONTINUE TO GET PLACED UNTIL EACH SM IS “FULL”

When a block completes its work and exits, a new block is placed in its spot until the whole grid is done



Using (effectively) CUDA core

BLOCKS CONTINUE TO GET PLACED UNTIL EACH SM IS “FULL”

When a block completes its work and exits, a new block is placed in its spot until the whole grid is done



Using (effectively) CUDA core

BLOCKS CONTINUE TO GET PLACED UNTIL EACH SM IS “FULL”

When a block completes its work and exits, a new block is placed in its spot until the whole grid is done



Using (effectively) CUDA core

WHAT DOES IT MEAN FOR AN SM TO BE “FULL”?



Using (effectively) CUDA core

LOOKING INSIDE A STREAMING MULTIPROCESSOR

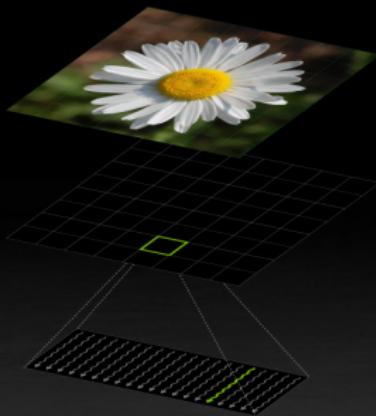


A100 SM Resources

2048	Max threads per SM
32	Max blocks per SM
65,536	Total registers per SM
160 kB	Total shared memory in SM
32	Threads per warp
4	Concurrent warps active
64	FP32 cores per SM
32	FP64 cores per SM
192 kB	Max L1 cache size
90 GB/sec	Load bandwidth per SM
1410 MHz	GPU Boost Clock

Using (effectively) CUDA core

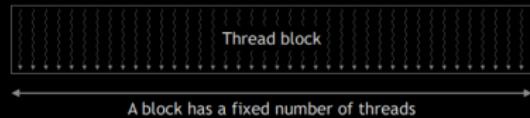
THE CUDA PROGRAMMING MODEL



Grid
of work

Divide into
many blocks

Many threads
in each block



```
_shared__ float mean = 0.0f;  
__device__ float mean_euclidian_distance(float2 *p1, float2 *p2) {  
    // Compute the Euclidian distance between two points  
    float2 dp = p2[threadIdx.x] - p1[threadIdx.x];  
    float dist = sqrtf(dp.x * dp.x + dp.y * dp.y);  
  
    // Accumulate the mean distance atomically and return distance  
    atomicAdd(&mean, dist / blockDim.x);  
    return dist;  
}
```

Every thread runs exactly the same program
(this is the "SIMT" model)

Using (effectively) CUDA core

ANATOMY OF A THREAD BLOCK

All blocks in a grid run the same program using the same number of threads, leading to 3 resource requirements

1. Block size - the number of threads which must be concurrent



```
__shared__ float mean = 0.0f;  
__device__ float mean_euclidian_distance(float2 *p1, float2 *p2) {  
    // Compute the Euclidian distance between two points  
    float2 dp = p2[threadIdx.x] - p1[threadIdx.x];  
    float dist = sqrtf(dp.x * dp.x + dp.y * dp.y);  
  
    // Accumulate the mean distance atomically and return distance  
    atomicAdd(&mean, dist / blockDim.x);  
    return dist;  
}
```

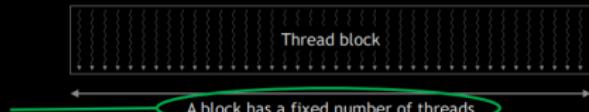
Every thread runs exactly the same program
(this is the "SIMT" model)

Using (effectively) CUDA core

ANATOMY OF A THREAD BLOCK

All blocks in a grid run the same program using the same number of threads, leading to 3 resource requirements

1. Block size - the number of threads which must be concurrent



2. Shared memory - common to all threads in a block

```
__shared__ float mean = 0.0f;
__device__ float mean_euclidian_distance(float2 *p1, float2 *p2) {
    // Compute the Euclidian distance between two points
    float2 dp = p2[threadIdx.x] - p1[threadIdx.x];
    float dist = sqrtf(dp.x * dp.x + dp.y * dp.y);

    // Accumulate the mean distance atomically and return distance
    atomicAdd(&mean, dist / blockDim.x);
    return dist;
}
```



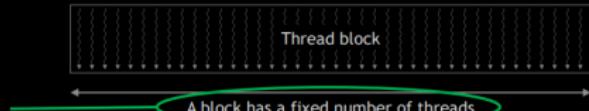
Every thread runs exactly the same program
(this is the "SIMT" model)

Using (effectively) CUDA core

ANATOMY OF A THREAD BLOCK

All blocks in a grid run the same program using the same number of threads, leading to 3 resource requirements

1. Block size - the number of threads which must be concurrent



2. Shared memory - common to all threads in a block

```
__shared__ float mean = 0.0f;
__device__ float mean_euclidian_distance(float2 *p1, float2 *p2) {
    // Compute the Euclidian distance between two points
    float2 dp = p2[threadIdx.x] - p1[threadIdx.x];
    float dist = sqrtf(dp.x * dp.x + dp.y * dp.y);

    // Accumulate the mean distance atomically and return distance
    atomicAdd(&mean, dist / blockDim.x);
    return dist;
}
```

3. Registers - depends on program complexity

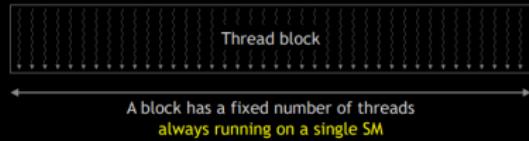
Registers are a per-thread resource, so total budget is:
(threads-per-block x registers-per-thread)

Every thread runs exactly the same program
(this is the "SIMT" model)

Using (effectively) CUDA core

HOW THE GPU PLACES BLOCKS ON AN SM

A100 SM Key Resources	
2048 Threads	
65536 Registers	
160kB Shared Memory	



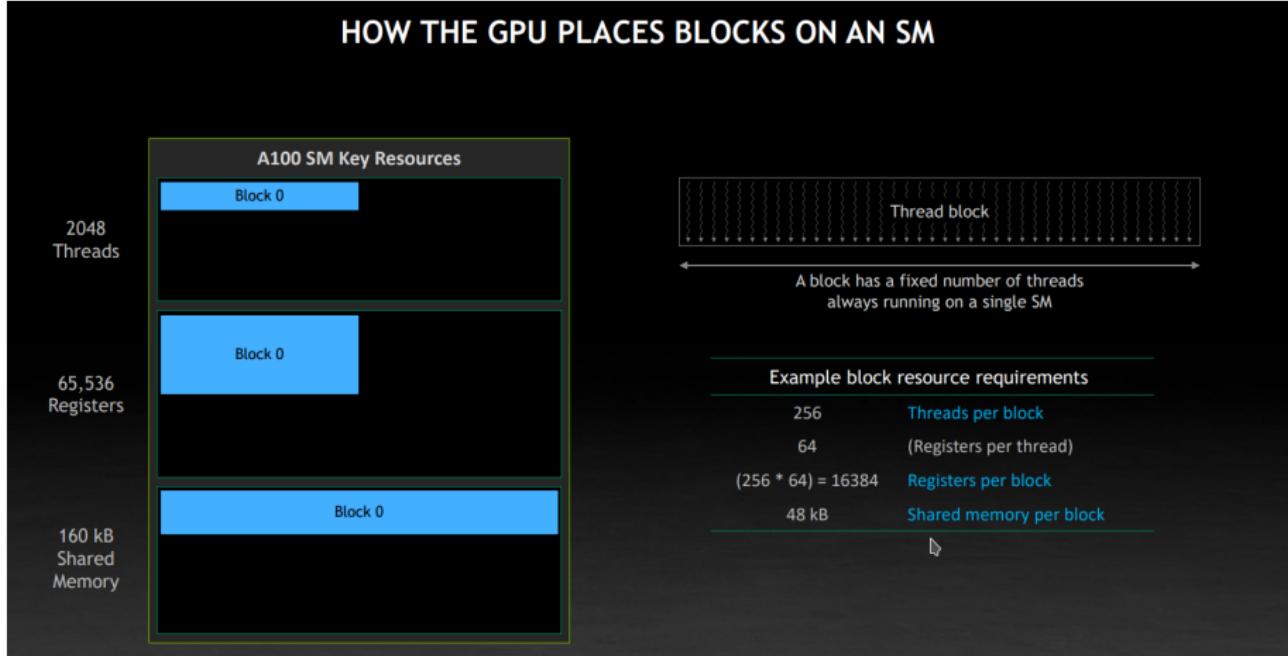
Example block resource requirements

256	Threads per block
64	(Registers per thread)
$(256 * 64) = 16384$	Registers per block
48 kB	Shared memory per block



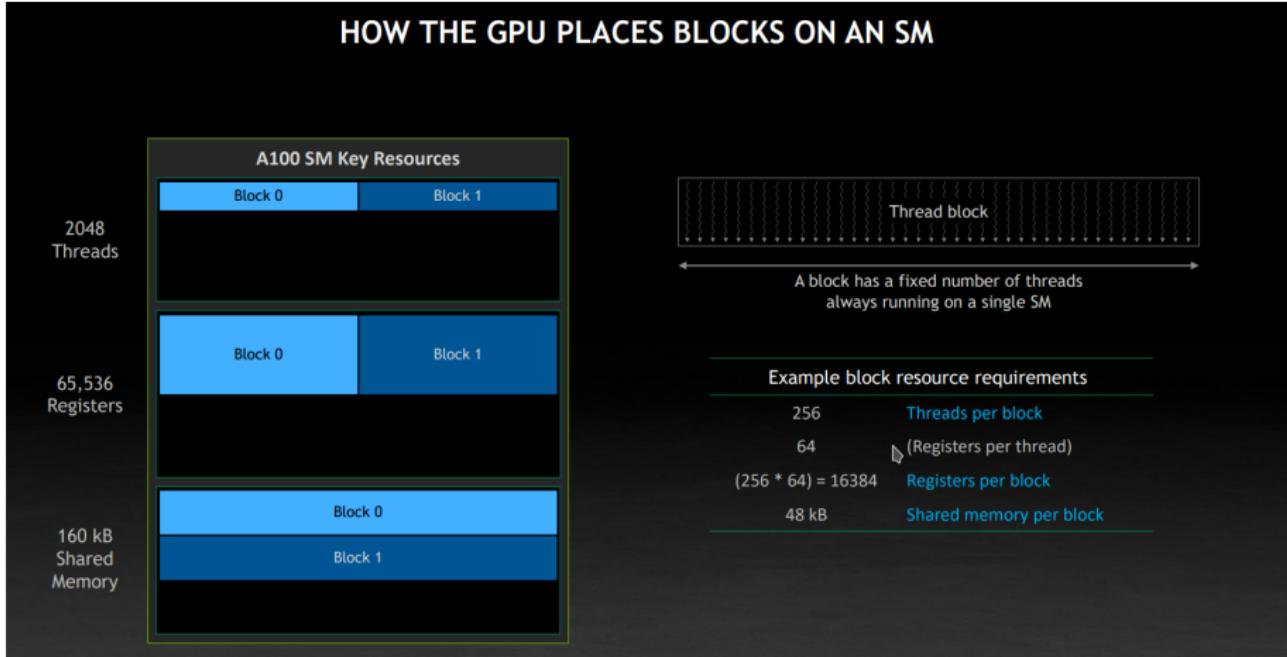
Using (effectively) CUDA core

HOW THE GPU PLACES BLOCKS ON AN SM



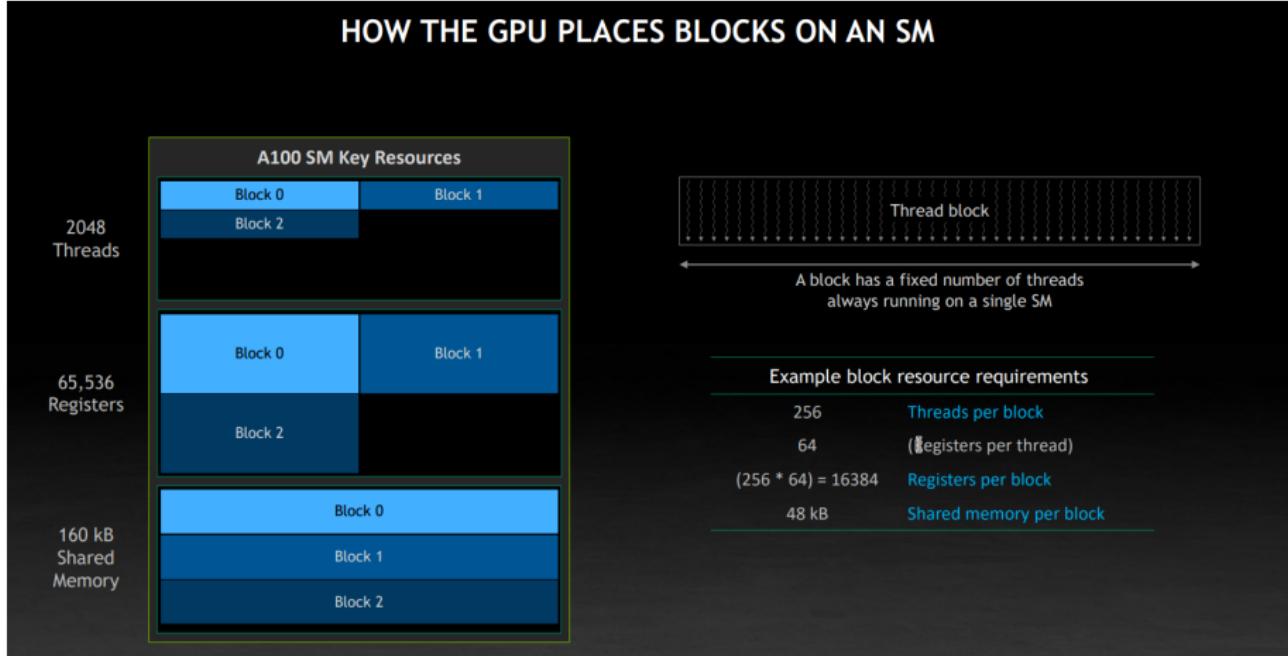
Using (effectively) CUDA core

HOW THE GPU PLACES BLOCKS ON AN SM



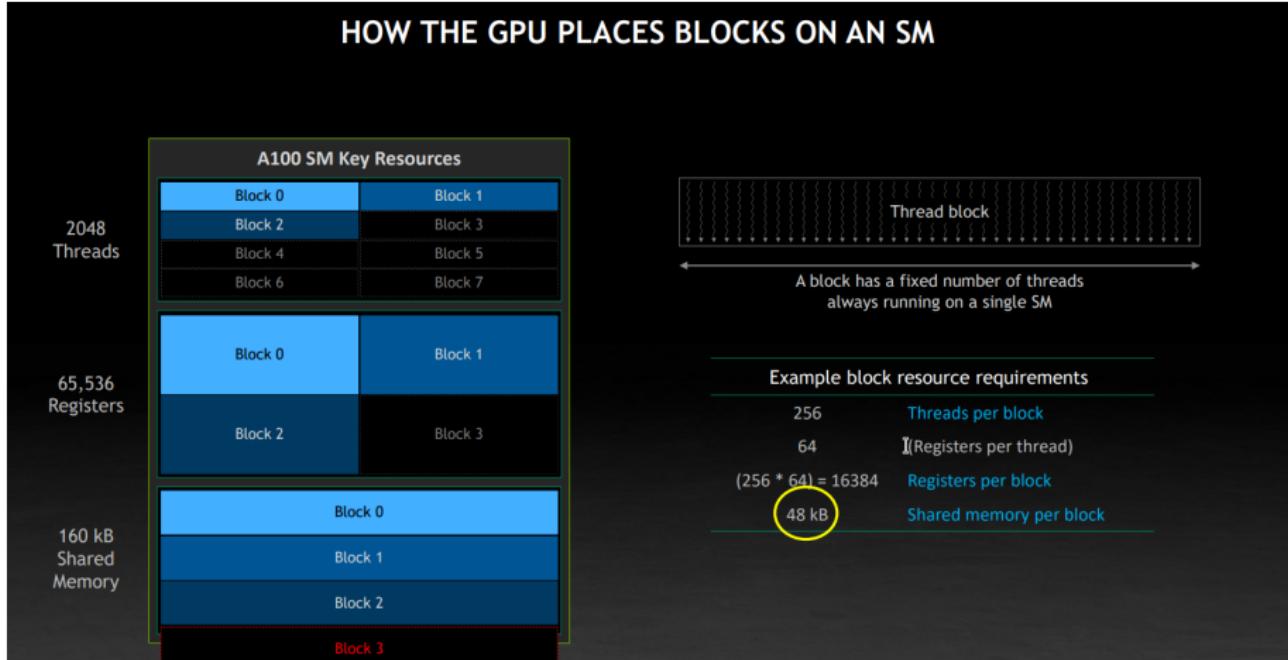
Using (effectively) CUDA core

HOW THE GPU PLACES BLOCKS ON AN SM



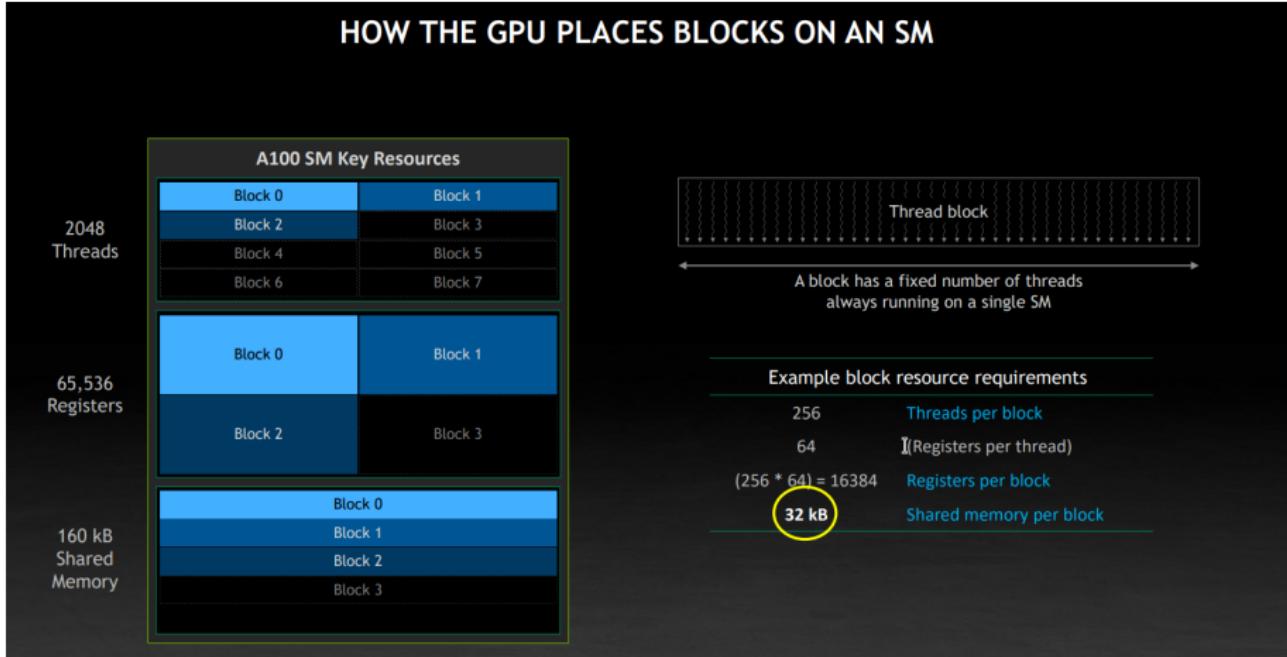
Using (effectively) CUDA core

HOW THE GPU PLACES BLOCKS ON AN SM



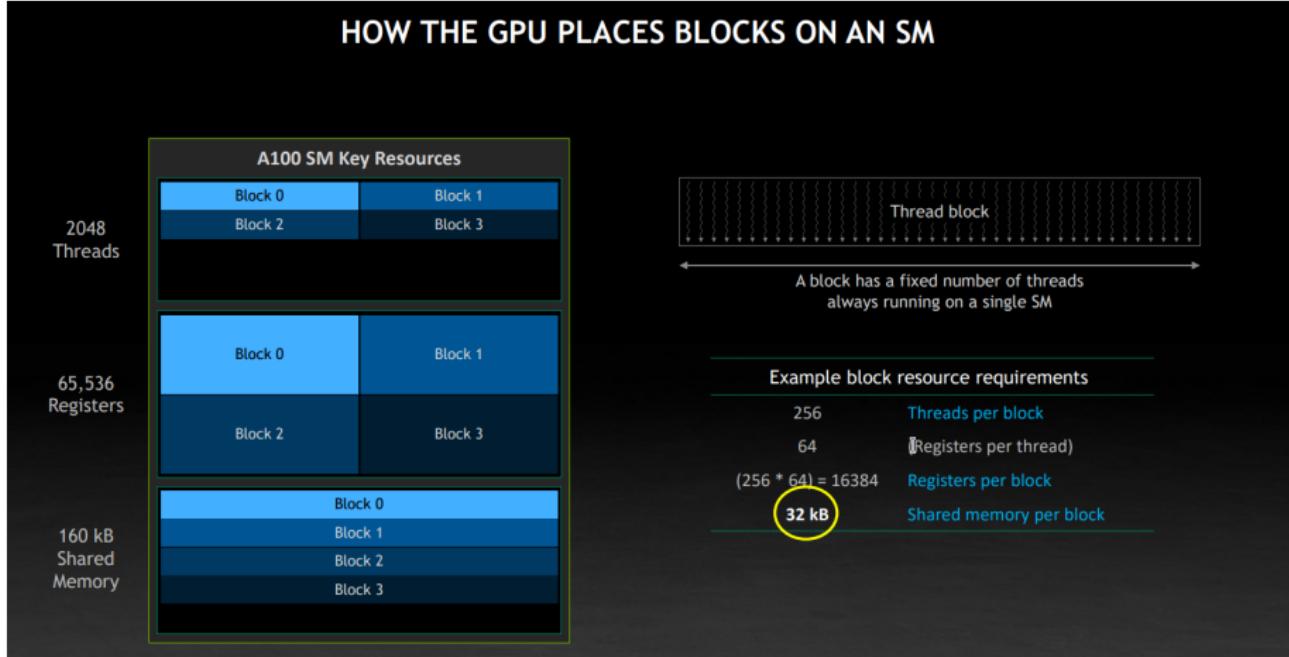
Using (effectively) CUDA core

HOW THE GPU PLACES BLOCKS ON AN SM



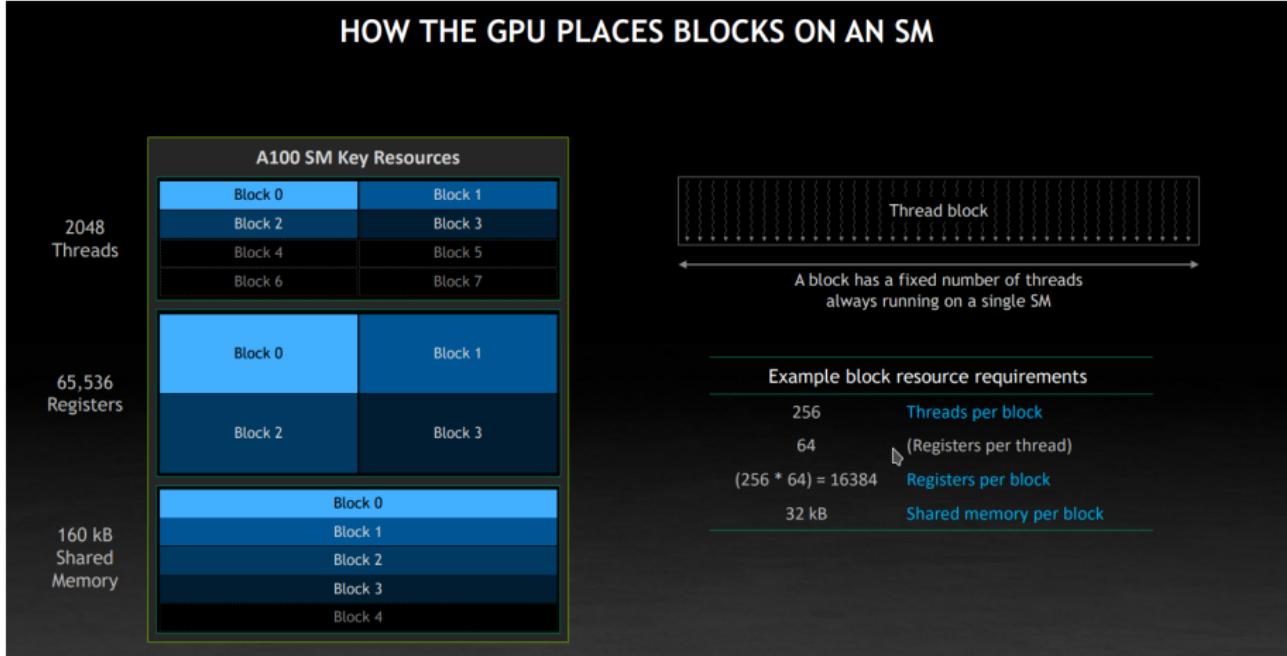
Using (effectively) CUDA core

HOW THE GPU PLACES BLOCKS ON AN SM



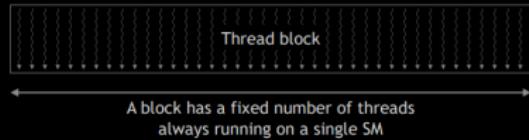
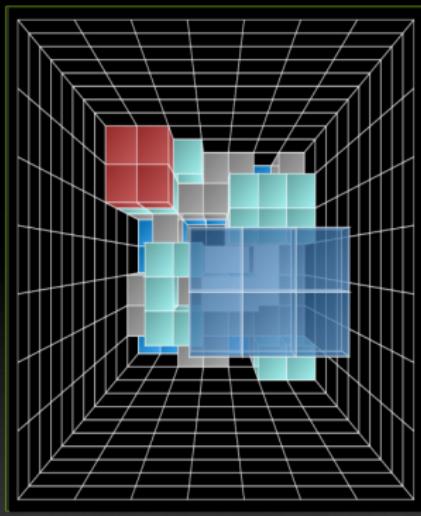
Using (effectively) CUDA core

HOW THE GPU PLACES BLOCKS ON AN SM



Using (effectively) CUDA core

HOW THE GPU PLACES BLOCKS ON AN SM

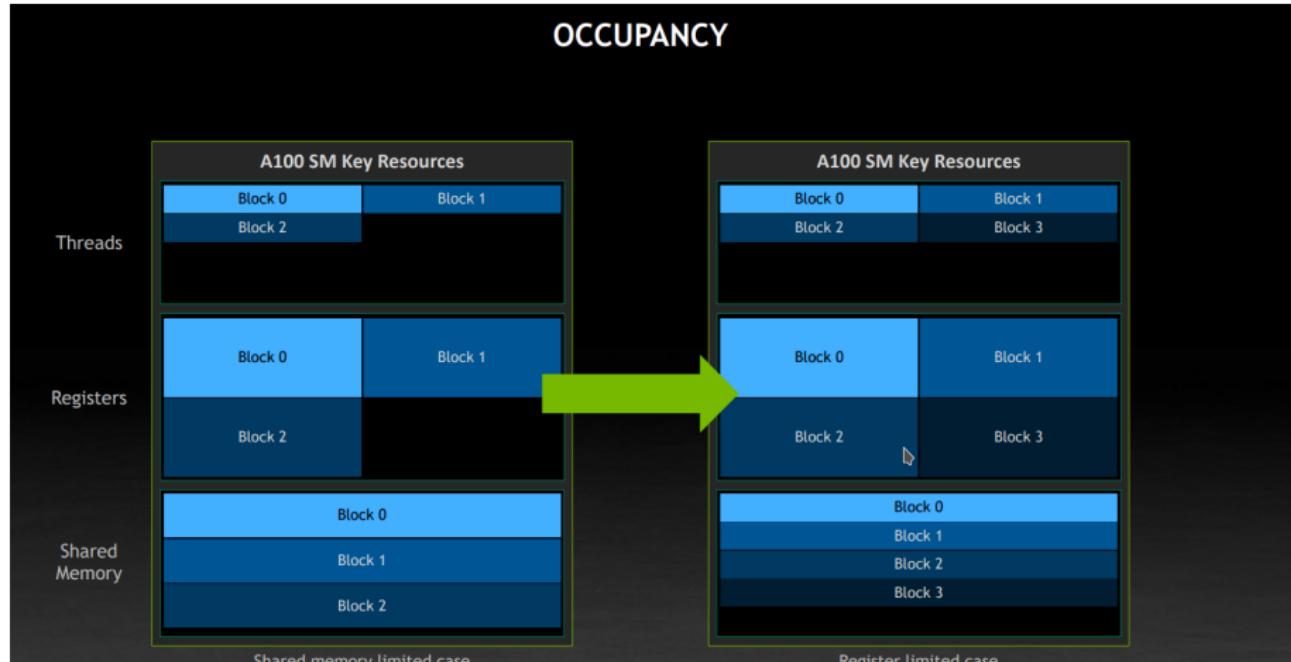


Example block resource requirements

256	↳ Threads per block
64	(Registers per thread)
$(256 * 64) = 16384$	Registers per block
32 kB	Shared memory per block

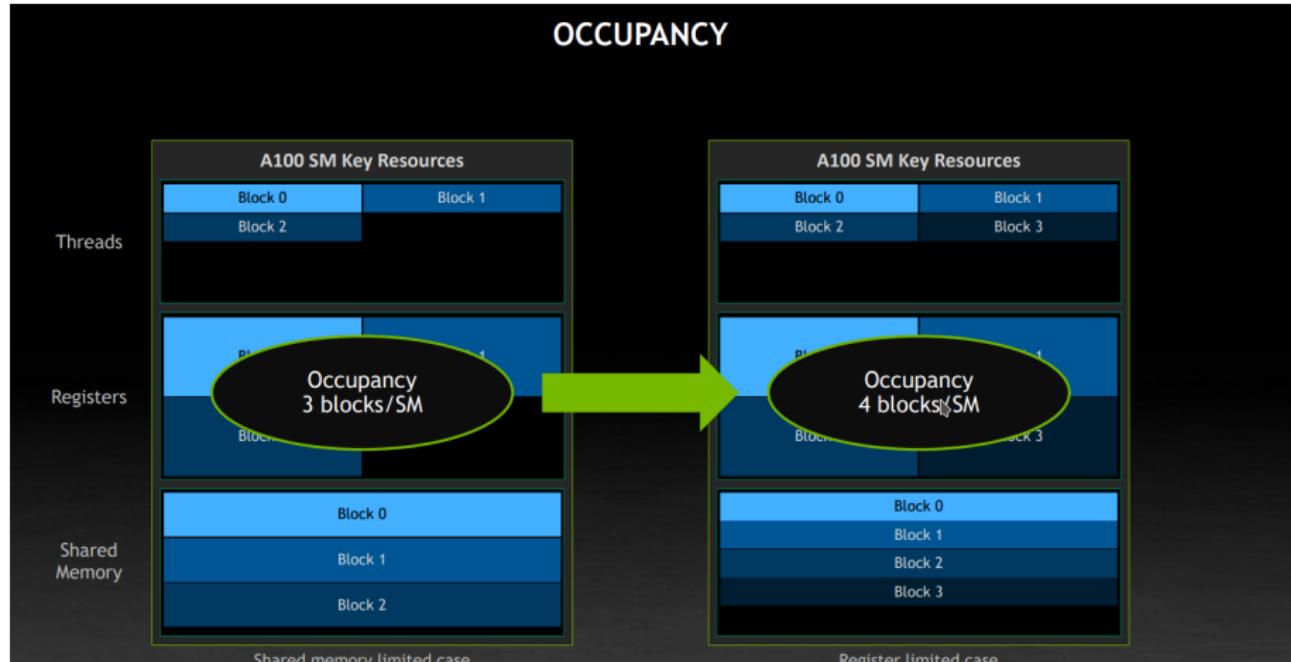
Using (effectively) CUDA core

OCCUPANCY



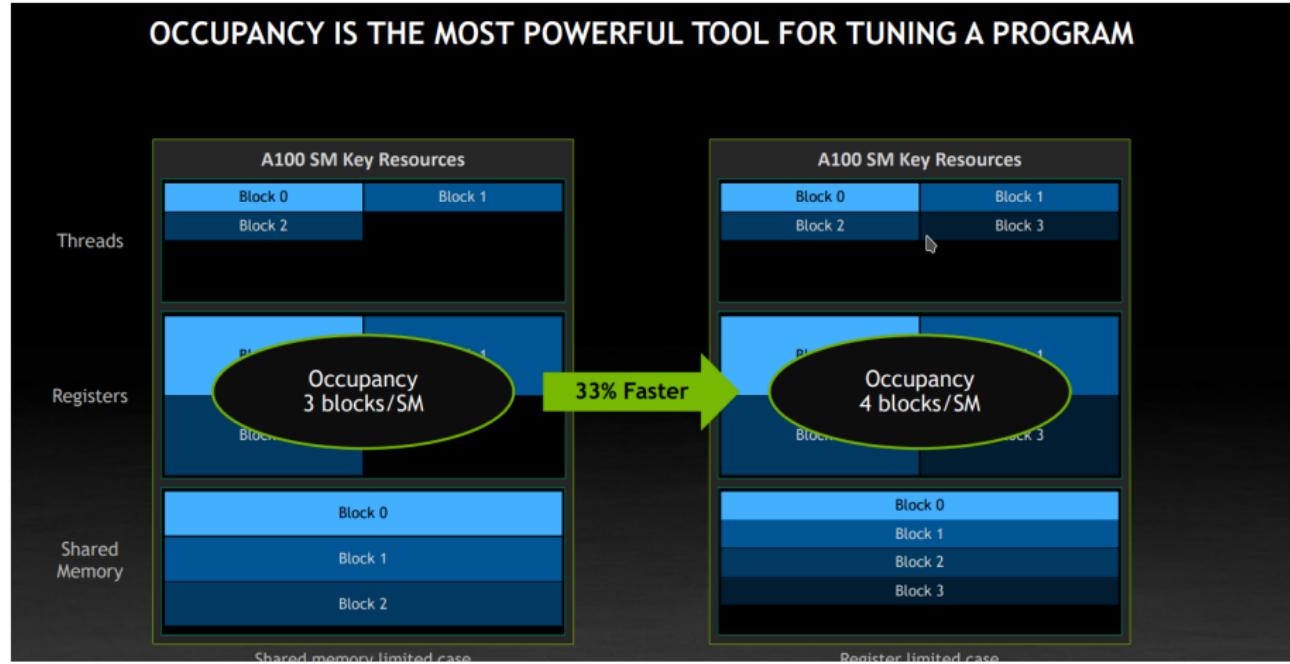
Using (effectively) CUDA core

OCCUPANCY



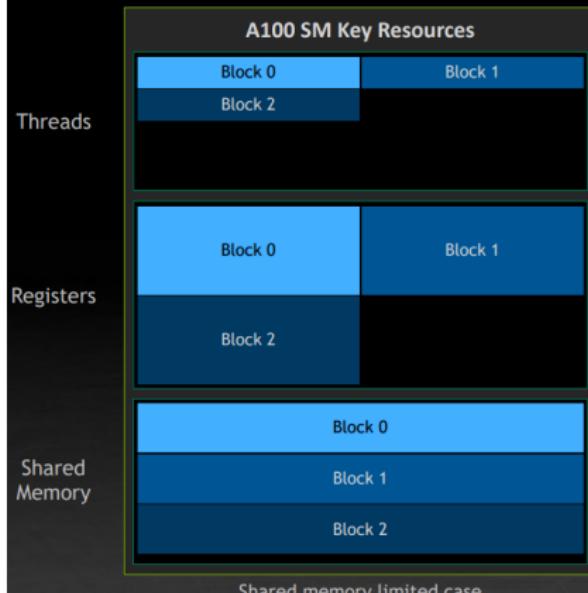
Using (effectively) CUDA core

OCCUPANCY IS THE MOST POWERFUL TOOL FOR TUNING A PROGRAM



Using (effectively) CUDA core

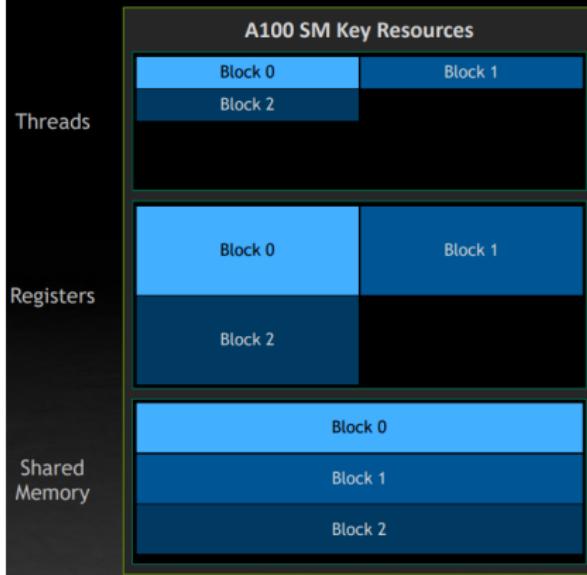
FILLING IN THE GAPS



Resource requirements (blue grid)	
256	Threads per block
64	(Registers per thread)
$(256 * 64) = 16384$	Registers per block
48 kB	Shared memory per block

Using (effectively) CUDA core

FILLING IN THE GAPS

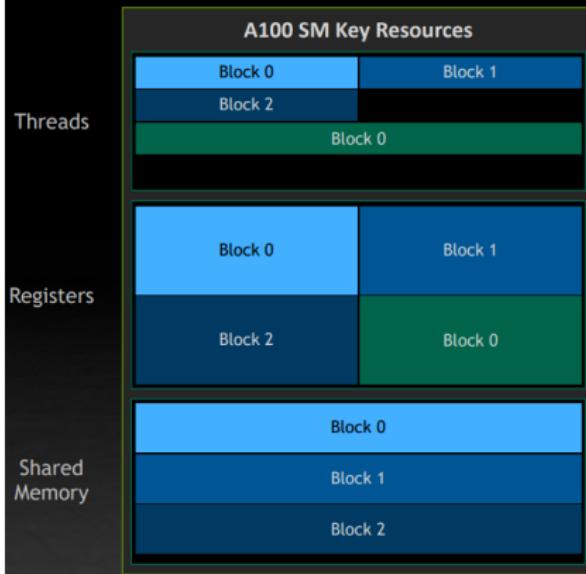


Resource requirements (blue grid)	
256	Threads per block
64	(Registers per thread)
$(256 * 64) = 16384$	Registers per block
48 kB	Shared memory per block

Resource requirements (green grid)	
512	Threads per block
32	(Registers per thread)
$(512 * 32) = 16384$	Registers per block
0 kB	Shared memory per block

Using (effectively) CUDA core

FILLING IN THE GAPS



Resource requirements (blue grid)

256 Threads per block

64 (Registers per thread)

$(256 * 64) = 16384$ Registers per block

48 kB Shared memory per block

Resource requirements (green grid)

512 Threads per block

32 (Registers per thread)

$(512 * 32) = 16384$ Registers per block

0 kB Shared memory per block

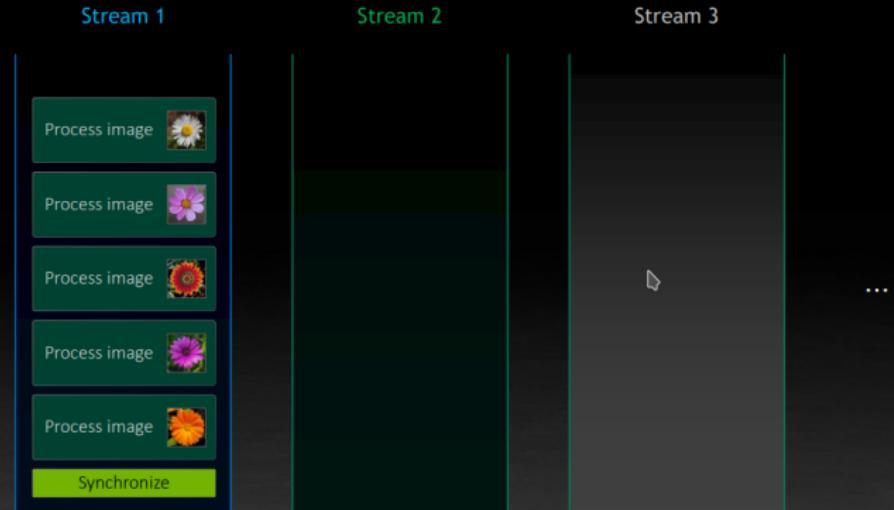
Using (effectively) CUDA core

ASYNCHRONOUS EXECUTION



Using (effectively) CUDA core

ASYNCHRONOUS EXECUTION



Using (effectively) CUDA core

EVEN-MORE-ASYNCHRONOUS EXECUTION

Stream 1

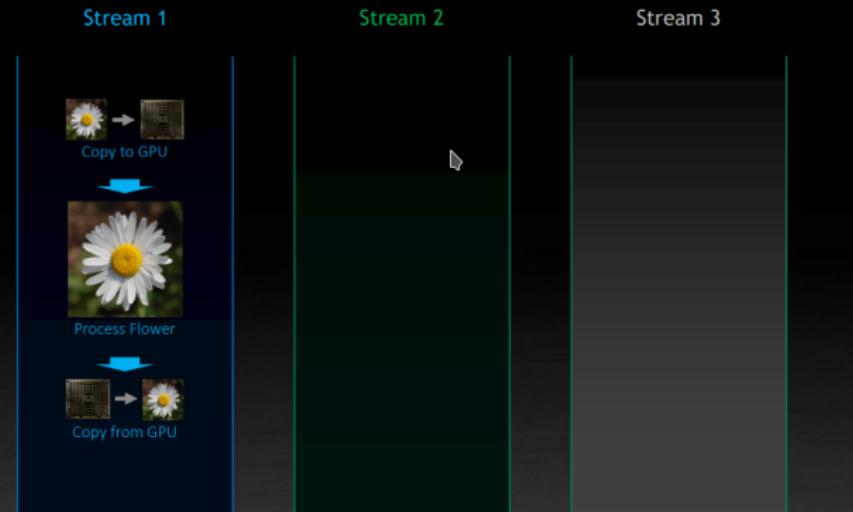
Stream 2

Stream 3



Using (effectively) CUDA core

EVEN-MORE-ASYNCHRONOUS EXECUTION



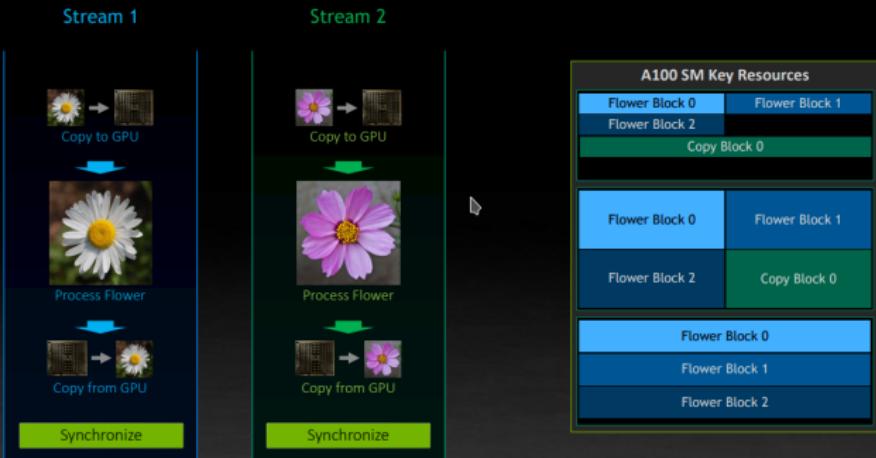
Using (effectively) CUDA core

EVEN-MORE-ASYNCHRONOUS EXECUTION



Using (effectively) CUDA core

KEEPING THE GPU FULL



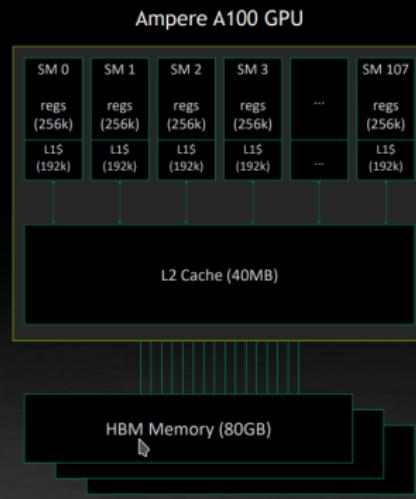
Using (effectively) CUDA core

HOW MUCH DATA CAN THE A100 GPU PULL IN?

108 SMs in the GPU, running @ 1410 MHz (boost clock)

Each SM can load 64 Bytes per clock cycle

Peak memory request rate = $64B \times 108 \text{ SMs} \times 1410 \text{ MHz} = \text{9750 Gigabytes/sec}$



Using (effectively) CUDA core

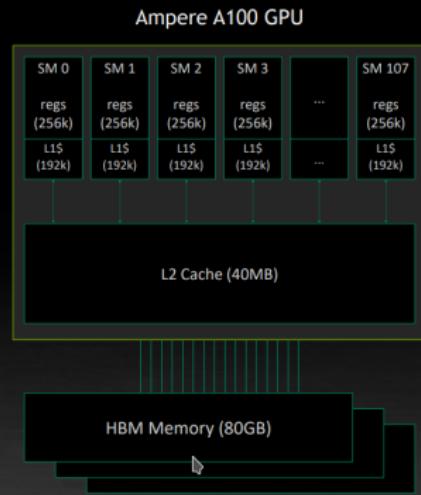
HOW MUCH DATA CAN THE A100 GPU PULL IN?

108 SMs in the GPU, running @ 1410 MHz (boost clock)

Each SM can load 64 Bytes per clock cycle

Peak memory request rate = $64B \times 108 \text{ SMs} \times 1410 \text{ MHz} = 9750 \text{ Gigabytes/sec}$

HBM2 Memory Bandwidth = **1555 GBytes / sec**



Using (effectively) CUDA core

HOW MUCH DATA CAN THE A100 GPU PULL IN?

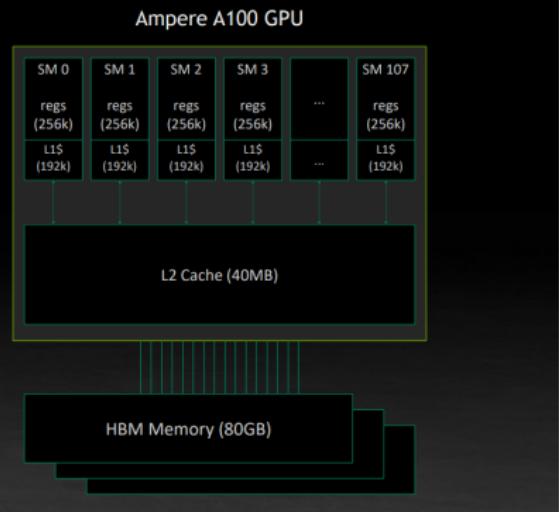
108 SMs in the GPU, running @ 1410 MHz (boost clock)

Each SM can load 64 Bytes per clock cycle

Peak memory request rate = $64B \times 108 \text{ SMs} \times 1410 \text{ MHz} = 9750 \text{ Gigabytes/sec}$

$$\text{Ratio of bandwidth requested to bandwidth provided} = \frac{9750}{1555} = 6.3x$$

HBM2 Memory Bandwidth = 1555 GBytes / sec



Using (effectively) CUDA core

BUT FLOPS AREN'T THE ISSUE - BANDWIDTH IS

SMs	108
Total threads	221,184
Peak FP64 TFLOP/s (tensor)	19.5
Tensor Core Precision	FP64, TF32, BF16, FP16, I8, I4, B1
Shared Memory per SM	160 kB
L2 Cache Size	40960 kB
Memory Bandwidth	1555 GB/sec
GPU Boost Clock	1410 MHz
NVLink Interconnect	600 GB/sec

FP64 FLOP/S BASED ON MEMORY SPEED = 1555 GB/SEC / 8 BYTES = 194 GFLOP/S

19.5 TFLOPs (tensor)



Hitachi SR2201 supercomputer

Using (effectively) CUDA core

BUT FLOPS AREN'T THE ISSUE - BANDWIDTH IS

SMs	108
Total threads	221,184
Peak FP64 TFLOP/s (tensor)	19.5
Tensor Core Precision	FP64, TF32, BF16, FP16, I8, I4, B1
Shared Memory per SM	160 kB
L2 Cache Size	40960 kB
Memory Bandwidth	1555 GB/sec
GPU Boost Clock	1410 MHz
NVLink Interconnect	600 GB/sec

FP64 FLOP/S BASED ON MEMORY SPEED = 1555 GB/SEC / 8 BYTES = 194 GFLOP/S

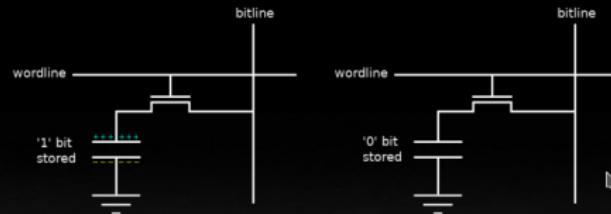
1555 GB/sec to 194 GFLOP/S



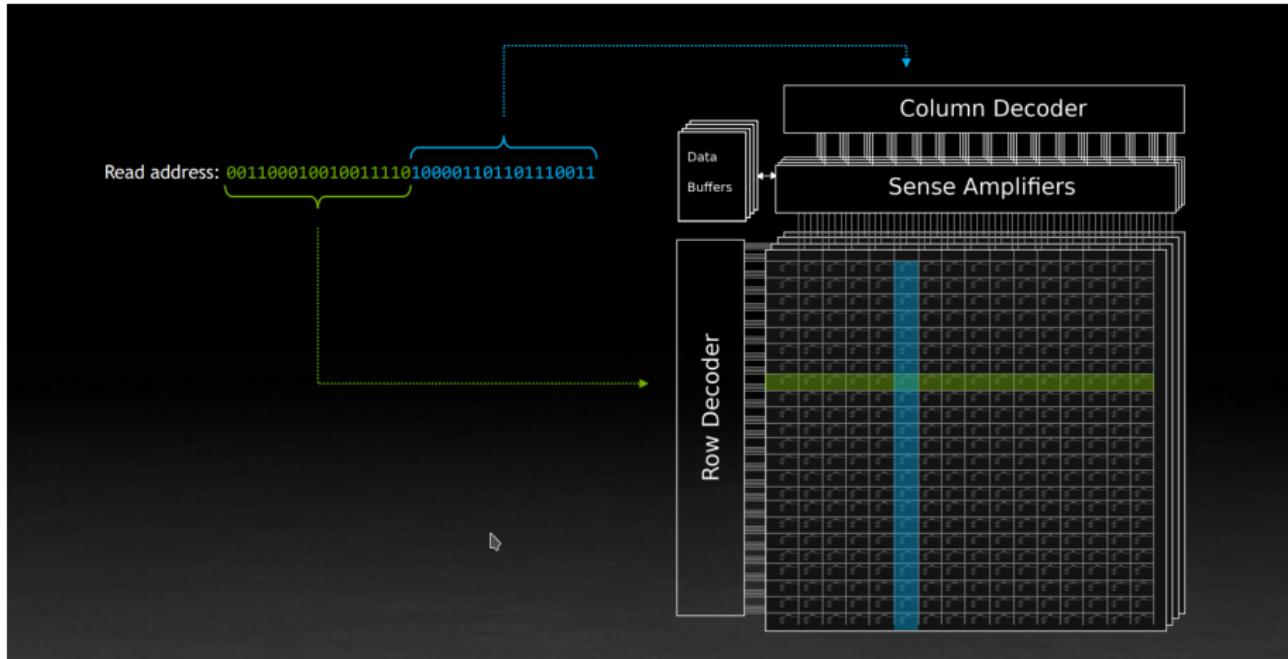
1x iPhone 11 Pro

Using (effectively) CUDA core

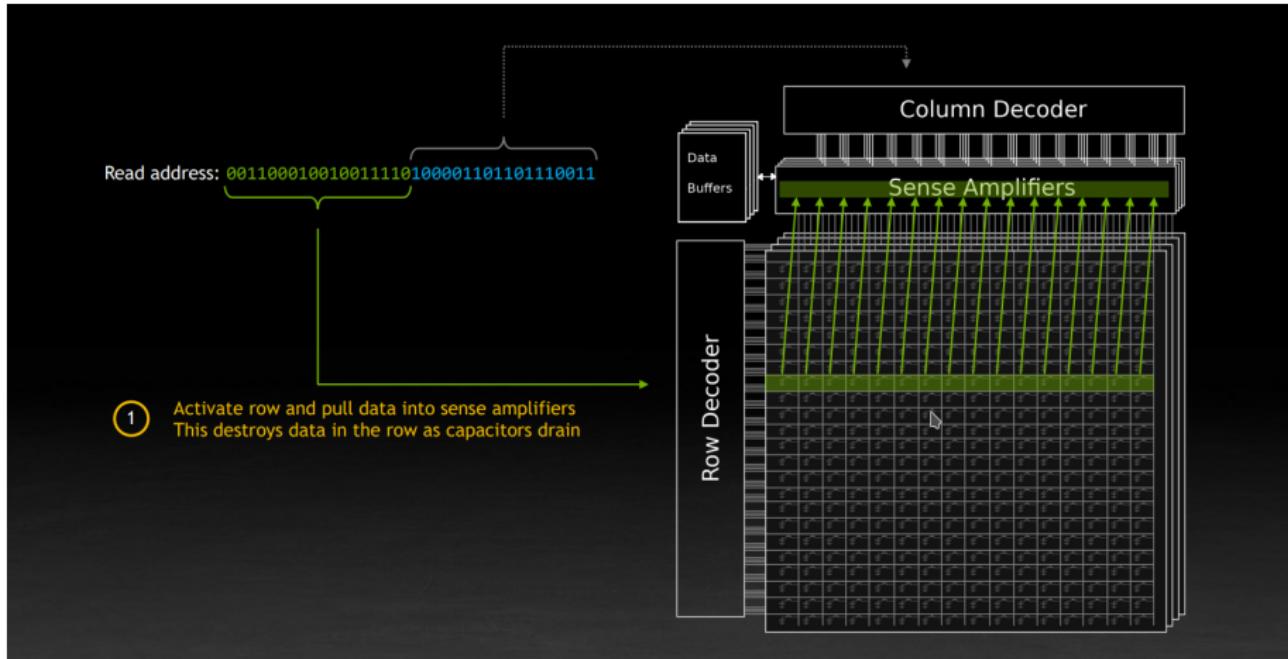
A CLOSER LOOK AT RANDOM ACCESS MEMORY



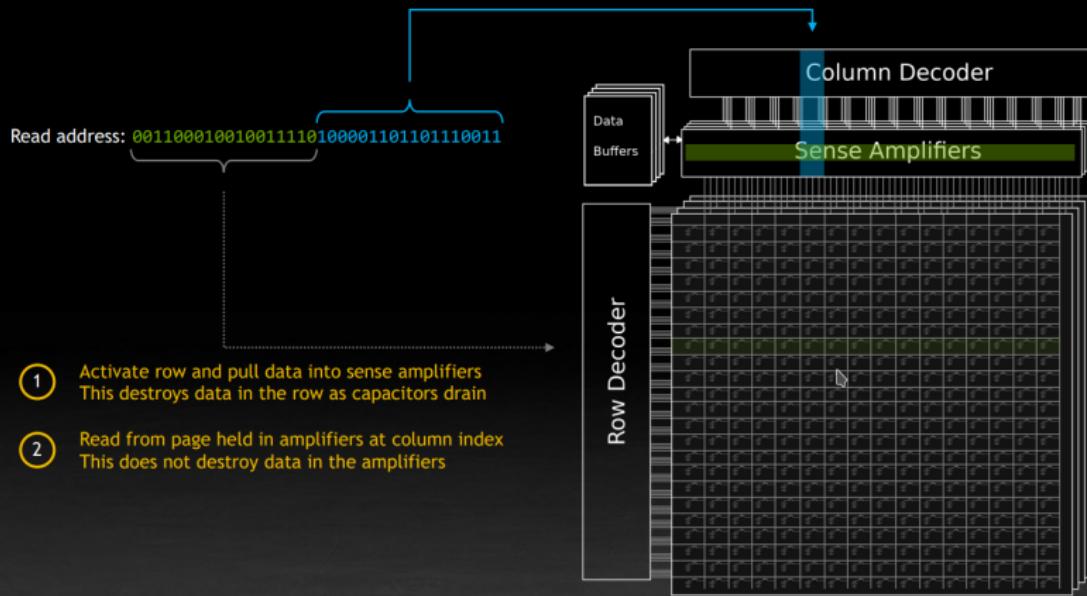
Using (effectively) CUDA core



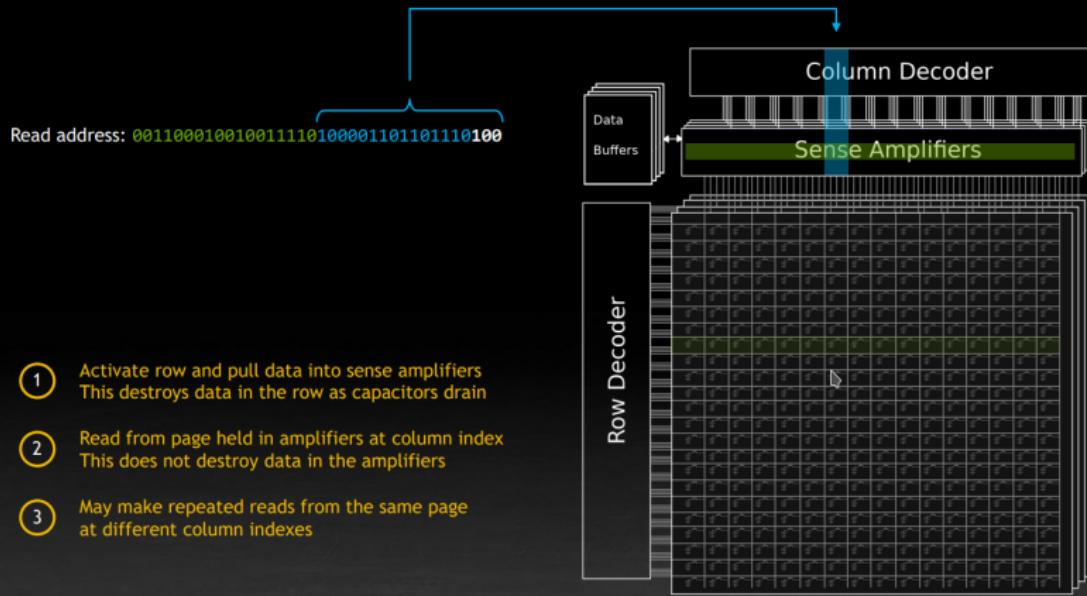
Using (effectively) CUDA core



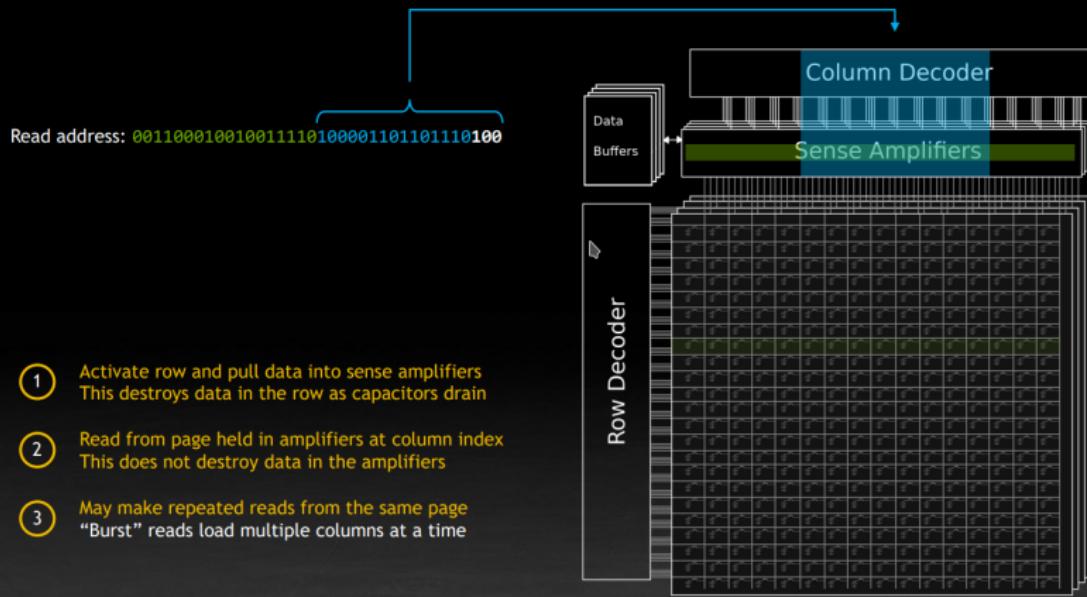
Using (effectively) CUDA core



Using (effectively) CUDA core



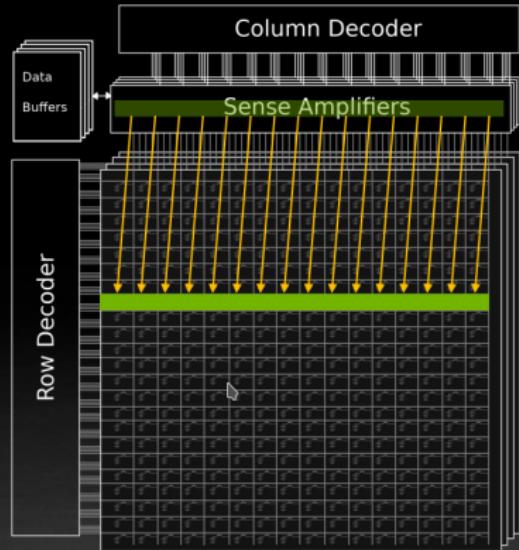
Using (effectively) CUDA core



Using (effectively) CUDA core

Read address: 001100010010011101100001101101110011

- ① Activate row and pull data into sense amplifiers
This destroys data in the row as capacitors drain
- ② Read from page held in amplifiers at column index
This does not destroy data in the amplifiers
- ③ May make repeated reads from the same page
“Burst” reads load multiple columns at a time
- ④ Before a new page is fetched, old row must be written back because data was destroyed



Using (effectively) CUDA core

Example HBM values

Time to read new column: $C_L = 16$ cycles

Time to load new page: $T_{RCD} = 16$ cycles

Time to write back data: $T_{RP} = 16$ cycles

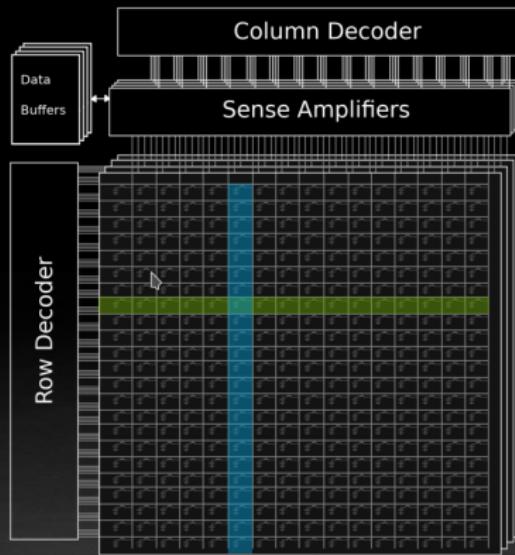
Page (row) size = 1kB

Each read has a cost ($C_L = 16$ cycles)

Switching page has 3x larger cost ($T_{RP} + T_{RCD} + C_L = 48$ cycles)

This is because switching page requires charging/discharging capacitors with a physical RC time constant:

$$V_C = V_S(1 - e^{(-t/RC)})$$



Using (effectively) CUDA core

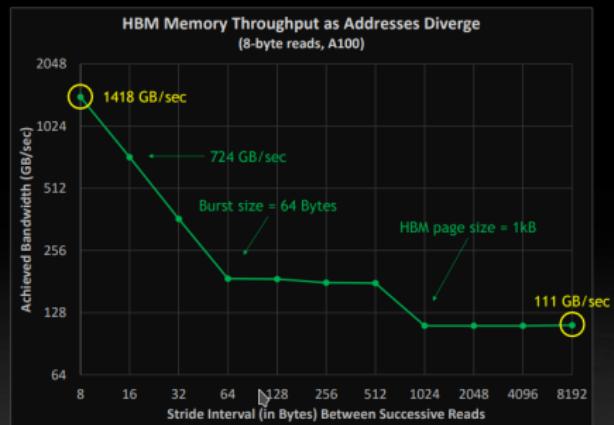
SO WHAT DOES THIS ALL MEAN?

We'd expect a significant performance difference for coalesced vs. scattered reads

On A100, memory bandwidth for widely-spaced reads is

$$\frac{111}{1418} = 8\% \text{ of peak bandwidth}$$

That's $1/13^{\text{th}}$ of peak bandwidth!



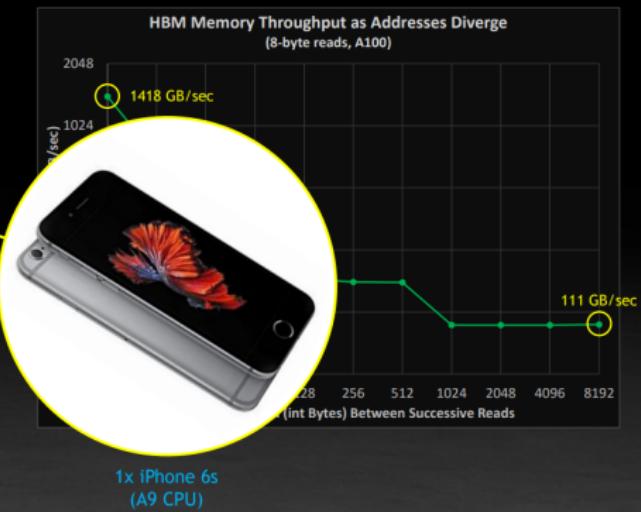
Using (effectively) CUDA core

SO WHAT DOES THIS ALL MEAN?

We'd expect a significant performance difference for coalesced vs. scattered reads

On A100, memory bandwidth for widely-spaced reads is

$$\frac{111}{1418} = 8\% \text{ of peak bandwidth}$$



Using (effectively) CUDA core

DATA ACCESS PATTERNS REALLY MATTER

```
for(y=0; y<M; y++) {  
    for(x=0; x<N; x++) {  
        load(array[y][x]);  
    }  
}
```

Row-major array traversal

Column read latency C_L



Row-major array layout

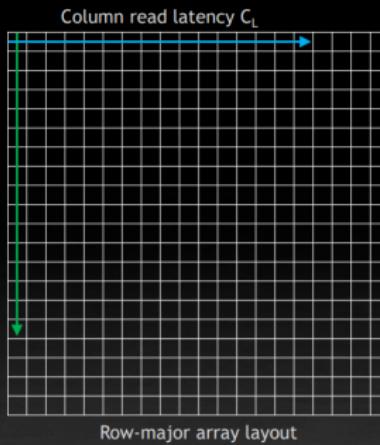


Using (effectively) CUDA core

DATA ACCESS PATTERNS REALLY MATTER

```
for(y=0; y<M; y++) {  
    for(x=0; x<N; x++) {  
        load(array[y][x]);  
    }  
}
```

Row-major array traversal



Row read latency
 $T_{RAS} = T_{RP} + T_{RDC} + C_L$
13x slower than
column access

```
for(x=0; x<N; x++) {  
    for(y=0; y<M; y++) {  
        load(array[y][x]);  
    }  
}
```

Column-major array traversal



Using (effectively) CUDA core

The reason you're using a GPU is for performance

This means using all the GPU resources that you can,
which means managing memory access patterns



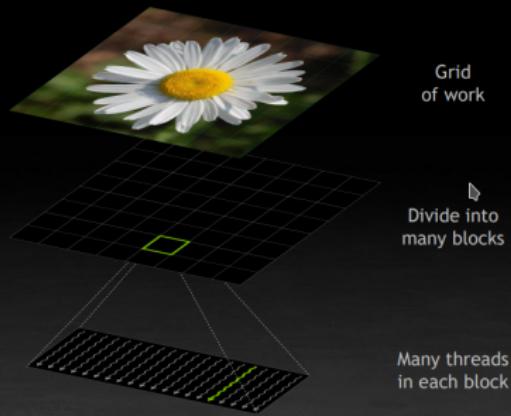
Using (effectively) CUDA core

But what's this got to do with CUDA?



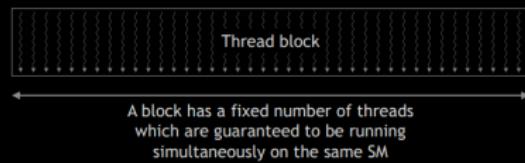
Using (effectively) CUDA core

CUDA'S GPU EXECUTION HIERARCHY



Using (effectively) CUDA core

THE CUDA THREAD BLOCK



I

Using (effectively) CUDA core

EVERY THREAD RUNS EXACTLY THE SAME PROGRAM

This is the “SIMT” model



```
__global__ void euclidian_distance(float2 *p1, float2 *p2, float *distance, int count) {
    // Calculate the index of the point my thread is working on
    int index = threadIdx.x + (blockIdx.x * blockDim.x);

    // Check if thread is in-range before reading data
    if (index < count) {
        // Compute the Euclidian distance between two points
        float2 dp = p2[index] - p1[index];
        float dist = sqrtf(dp.x * dp.x + dp.y * dp.y);

        // Write out the computed distance
        distance[index] = dist;
    }
}
```

Using (effectively) CUDA core

EVERY THREAD RUNS EXACTLY THE SAME PROGRAM

This is the “SIMT” model



It's all about this
one line of code

```
__global__ void euclidian_distance(Float3 *p1, Float3 *p2, float *distance, int count) {  
    // Calculate the index of the point my thread is working on  
    int index = threadIdx.x + (blockIdx.x * blockDim.x);  
  
    // Check if thread is in-range before reading data  
    if (index < count) {  
        // Compute the Euclidian distance between two points  
        float2 dp = p2[index] - p1[index];  
        float dist = sqrtf(dp.x * dp.x + dp.y * dp.y);  
  
        // Write out the computed distance  
        distance[index] = dist;  
    }  
}
```



Using (effectively) CUDA core

EVERY THREAD RUNS EXACTLY THE SAME PROGRAM

This is the “SIMT” model



This line is what
SIMT is all about

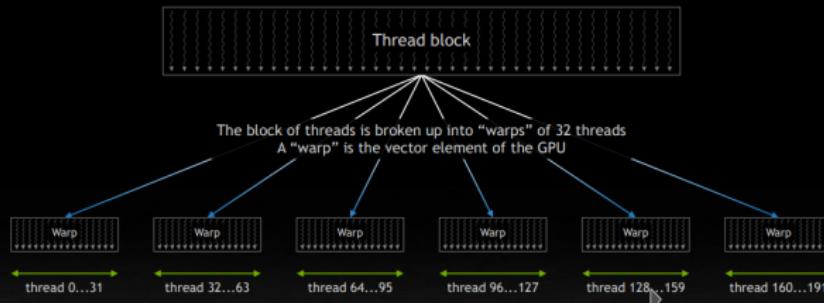
```
__global__ void euclidian_distance(float2 *p1, float2 *p2, float *distance, int count) {
    // Calculate the index of the point my thread is working on
    int index = threadIdx.x + (blockIdx.x * blockDim.x);

    // Check if thread is in-range before reading data
    if (index < count) {
        // Compute the Euclidian distance between two points
        float2 dp = p2[index] - p1[index];
        float dist = sqrtf(dp.x * dp.x + dp.y * dp.y);

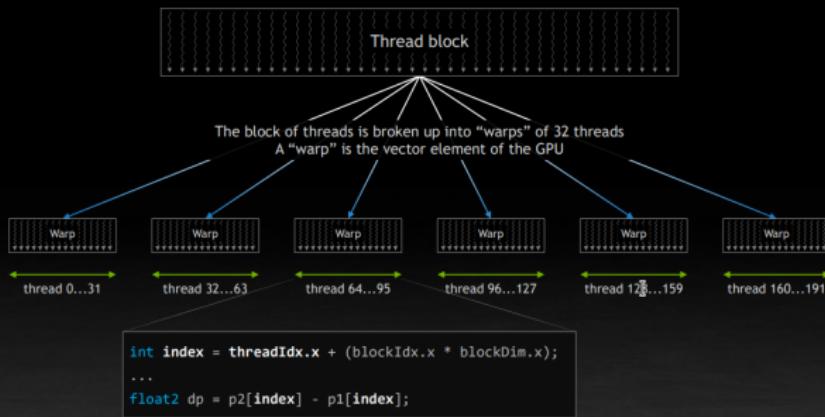
        // Write out the computed distance
        distance[index] = dist;
    }
}
```



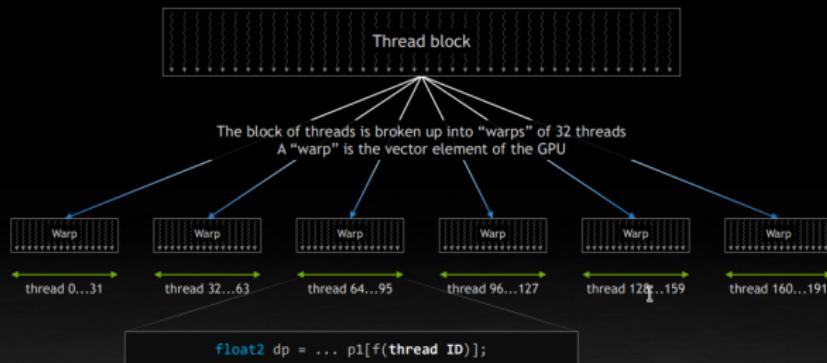
Using (effectively) CUDA core



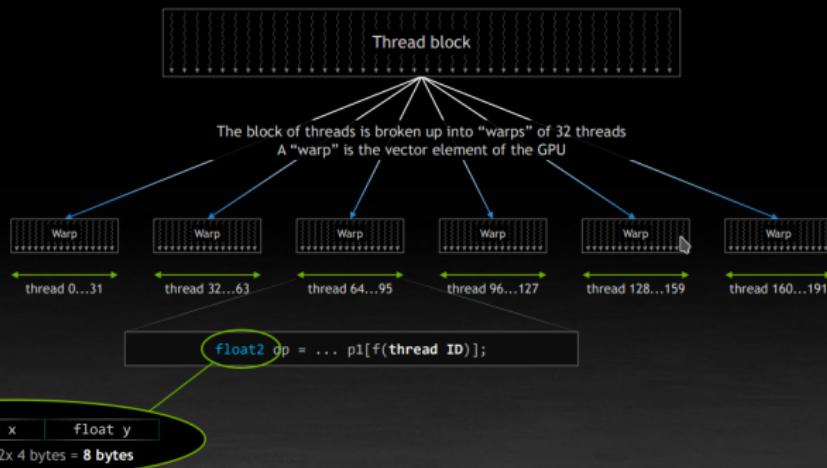
Using (effectively) CUDA core



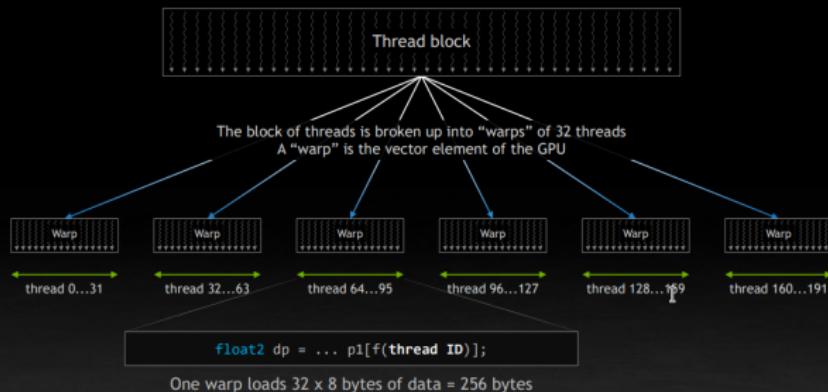
Using (effectively) CUDA core



Using (effectively) CUDA core



Using (effectively) CUDA core



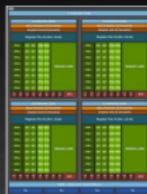
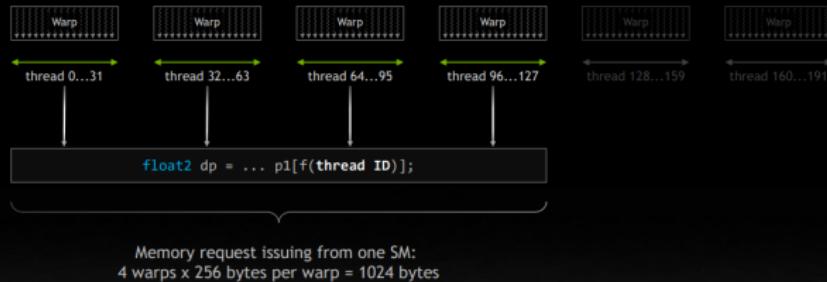
Using (effectively) CUDA core

WARP EXECUTION ON THE GPU



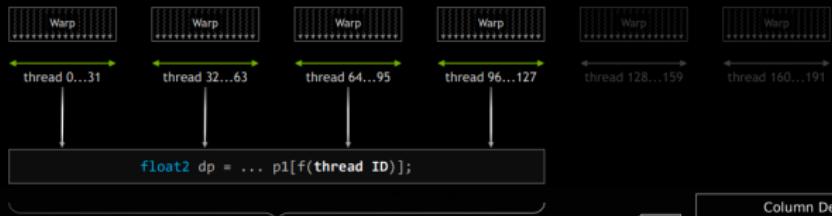
Using (effectively) CUDA core

WARP EXECUTION ON THE GPU

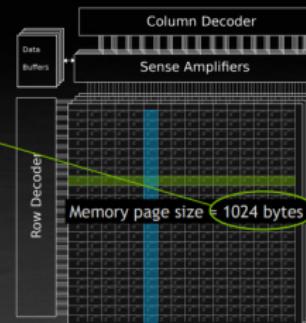


Using (effectively) CUDA core

WARP EXECUTION ON THE GPU



Memory request issuing from one SM:
4 warps x 256 bytes per warp = 1024 bytes



Using (effectively) CUDA core



Using (effectively) CUDA core



For a single thread, this would look like random-address memory reads

But because this is executed in parallel by 128 threads, it's actually adjacent reads of whole pages of memory

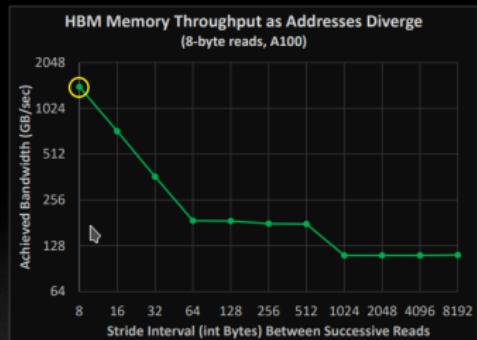


Using (effectively) CUDA core



For a single thread, this would look like random-address memory reads

But because this is executed in parallel by 128 threads, it's actually adjacent reads of whole pages of memory



BLAS and CUDA

Example: matrix transpose

- Where are bank conflicts?

Block bx,by, Thread tx,ty:

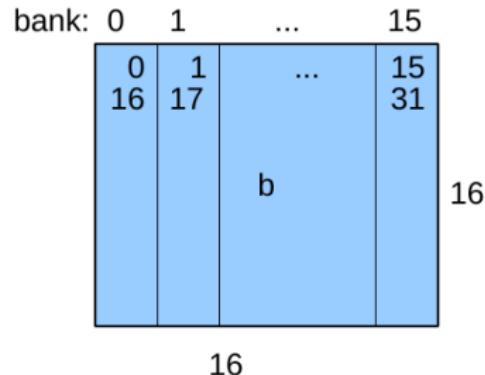
a[ty*16+tx]=A[by*16+ty,bx*16+tx]

Syncthreads

b[ty*16+tx]=a[tx*16+ty]

Syncthreads

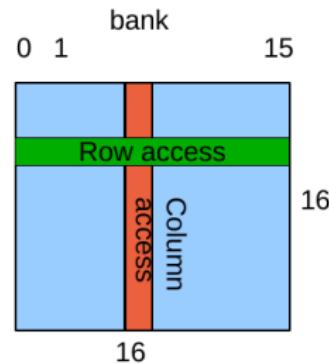
B[by*16+ty,bx*16+tx]=b[ty*16+tx]



BLAS and CUDA

Example: matrix transpose

- Where are bank conflicts?



Block bx,by, Thread tx,ty:

$a[ty*16+tx]=A[by*16+ty,bx*16+tx]$

Synchthreads

$b[ty*16+tx]=a[tx*16+ty]$

Synchthreads

$B[by*16+ty,bx*16+tx]=b[ty*16+tx]$

Column access: systematic conflicts

- How to avoid them?

BLAS and CUDA

Remapping data

- Solution 1: pad with empty cells

```
Block bx,by, Thread tx,ty:  

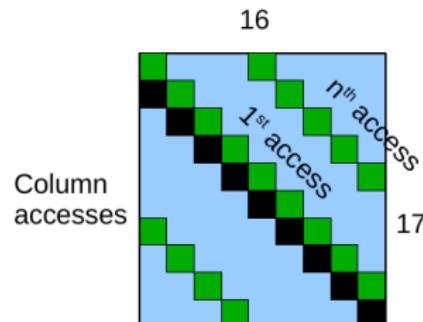
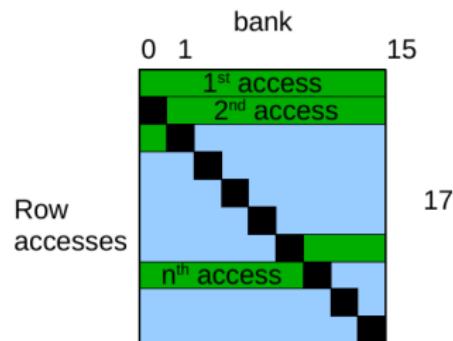
a[ty*17+tx]=A[by*16+ty,bx*16+tx]  

Syncthreads  

b[ty*16+tx]=a[tx*17+ty]  

Syncthreads  

B[by*16+ty,bx*16+tx]=b[ty*17+tx]
```



- No bank conflicts
- Memory overhead

BLAS and CUDA

Remapping data

- Solution 2: different mapping function
 - Example: map $[y,x]$ to $y*16+(x+y \bmod 16)$
 - Or $y*16+(x \wedge y)$

Block bx, by , Thread tx, ty :

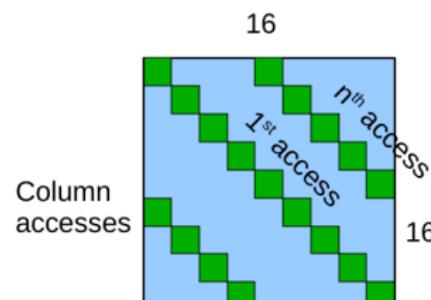
$a[ty*16+(tx+ty)\%16]=A[by*16+ty,bx*16+tx]$

Syncthreads

$b[ty*16+tx]=a[tx*16+(ty+tx)\%16]$

Syncthreads

$B[by*16+ty,bx*16+tx]=b[ty*17+tx]$



- No bank conflicts
- No memory overhead

BLAS and CUDA

Multiple grid/block dimensions

- Grid and block size are of type `dim3`
 - ◆ Support up to 3 dimensions

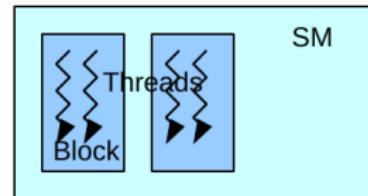
```
dim3 dimBlock(tx, ty, tz);
dim3 dimGrid(bx, by, bz);
my_kernel<<<dimGrid, dimBlock>>>(arguments);
    ◆ Implicit cast from int to dim3
    y and z sizes are 1 by default
```

- On device side, `threadIdx`, `blockDim`, `blockIdx`, `gridDim` are also of type `dim3`
 - ◆ Access members with `.x`, `.y`, `.z`

BLAS and CUDA

Occupancy metric

- Threads per SM / max threads per SM
- Resource usage may cause non-ideal occupancy
 - ◆ Register usage
 - ◆ Shared memory usage
 - ◆ Non-dividable block size



Available registers: 32768



Usage: 64 registers/thread,
blocks of 256 threads

→ Only 2 blocks / SM

Available shared memory: 16KB



Usage: 12KB/block

→ Only 1 block / SM

Max threads/SM: 768 threads



Block size: 512 threads

→ Only 1 block / SM

Could run 3 blocks of 256 threads

BLAS and CUDA

How many threads?

- As many as possible (maximize occupancy)?
 - + Maximal data-parallelism
 - Latency hiding
 - Locality
 - Store private data of each thread
 - Thread management overhead
 - Initialization, redundant operations
- Trade-off between parallelism and memory locality

BLAS and CUDA

Multiple elements per thread

- Block size (16, 16) → (8, 16)
- 2 elements per thread: (x, y) and (x+8, y)

```

c[0] = 0
c[1] = 0
for k = 0 to n-1 step 16
    a[y,x] = A[i+y,k+x]
    b[y,x] = B[k+y,j+x]
    a[y+8,x] = A[i+y+8,k+x]
    b[y+8,x] = B[k+y+8,j+x]
    Barrier
    for k2 = 0 to 15
        c[0] += a[y,k2]*b[k2,x]
        c[1] += a[y+8,k2]*b[k2,x]
    Barrier
    C[i+y,j+x] = c[0]
    C[i+y+8,j+x] = c[1]

```



More outstanding loads

- What about shared memory?

BLAS and CUDA

Data reuse

- Share reads to submatrix b
 - Fewer shared memory accesses
 - Exchange data through registers

```

c[0] = 0
c[1] = 0
for k = 0 to n-1 step 16
    a[y,x] = A[i+y,k+x]
    b[y,x] = B[k+y,j+x]
    a[y+8,x] = A[i+y+8,k+x]
    b[y+8,x] = B[k+y+8,j+x]
    Barrier
    for k2 = 0 to 15
        bl = b[k2,x]
        c[0] += a[y,k2]*bl
        c[1] += a[y+8,k2]*bl
    Barrier
    C[i+y,j+x] = c[0]
    C[i+y+8,j+x] = c[1]

```

- Improves register usage too. Why?

BLAS and CUDA

Expressing vector loads

- Multiple **consecutive** elements per thread
 - Here: $4*x, 4*x+1, 4*x+2, 4*x+3$
- Load, store, and compute on short vectors

```

c[0..3] = 0
for k = 0 to n-1 step 4*16
  a[y,4*x..4*x+3]) = A[i+y,k+4*x..k+4*x+3]
  b[y,4*x..4*x+3] = B[k+y,j+4*x..j+4*x+3]
  Barrier
  for k2 = 0 to 15
    bl[0..3] = b[k2,4*x..4*x+3]
    c[0..3] += a[y,k2]*bl[0..3]
  Barrier
  C[i+y,4*(j+x)..4*(j+x)+3] = c[0..3]

```

Vector loads from global memory

Vector loads/store-in shared memory

Scalar-vector product

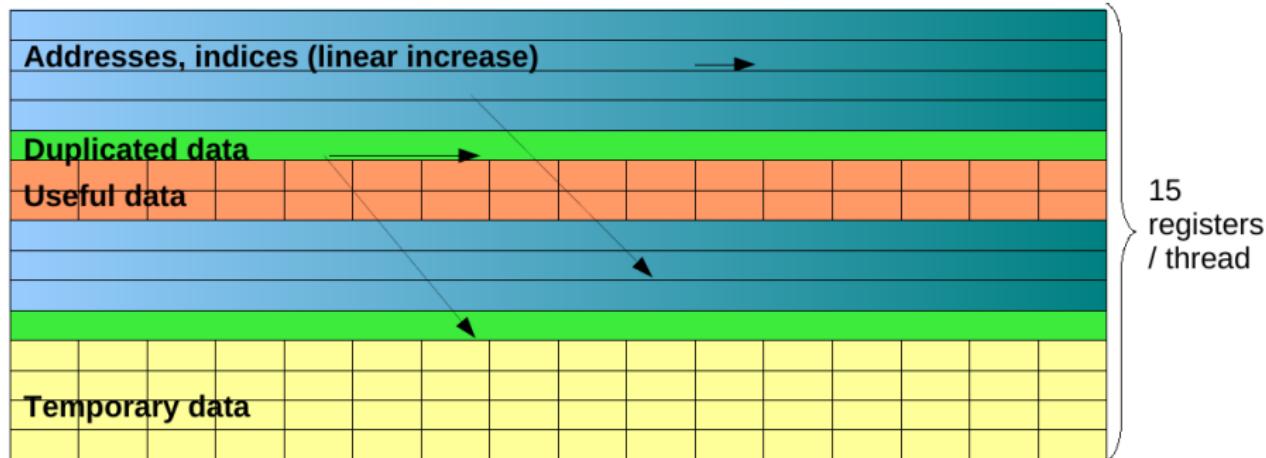
Vector store to global memory

BLAS and CUDA

Example: SGEMM from CUBLAS 1.1

NVIDIA's reference matrix multiplication code in 2008

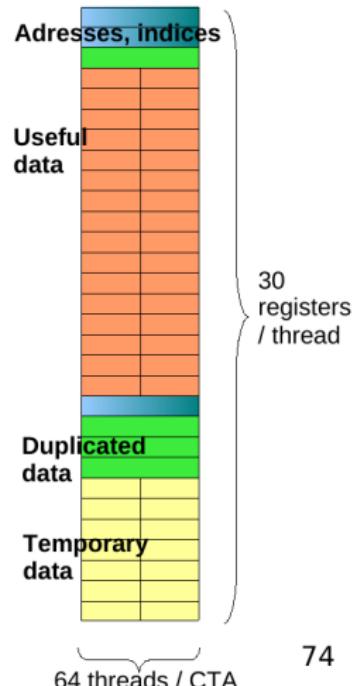
- 512 threads / CTA, 15 registers / thread
- 9 registers / 15 contain redundant data
- Only 2 registers really needed



BLAS and CUDA

Fewer threads, more computations

- Optimized version:
SGEMM in CUBLAS 2.0
 - 8 elements computed / thread
 - Unrolled loops
 - Less traffic through shared memory, more through registers
- Overhead amortized
 - 1920 registers vs. 7680 for the same amount of work
 - Works for redundant computations too
- Instruction-level parallelism is still relevant



74

BLAS and CUDA

Re-expressing parallelism

I

- Converting types of parallelism



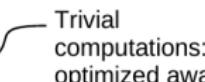
- General strategy

- ◆ Design phase: focus on thread-level parallelism
- ◆ Optimization phase:
convert TLP to Instruction-level or Data-level parallelism

BLAS and CUDA

Loop unrolling

- Can improve performance
 - ◆ Amortizes loop overhead over several iterations
 - ◆ May allow constant propagation, common sub-expression elimination...
- Unrolling is **necessary** to keep arrays in registers

<p>Not unrolled</p> <pre>int a[4]; for(int i = 0; i < 4; i++) { a[i] = 3 * i; }</pre> <p> Indirect addressing: a in local memory</p>	<p>Unrolled</p> <pre>int a[4]; a[0] = 3 * 0; a[1] = 3 * 1; a[2] = 3 * 2; a[3] = 3 * 3;</pre> <p> Static addressing: a in registers</p> <p> Trivial computations: optimized away</p>
--	--

- The compiler can unroll for you

```
#pragma unroll
for(int i = 0; i < 4; i++) {
    a[i] = 3 * i;
}
```

Branch divergence

Warp-based execution

- Threads in a warp run in lockstep
 - I
- On NVIDIA architectures, warp is 32 threads
- A block is made of warps
(warps do not cross block boundaries)
 - ◆ Block size multiple of 32 for best performance

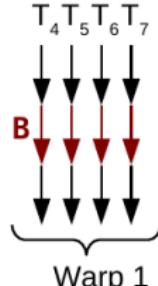
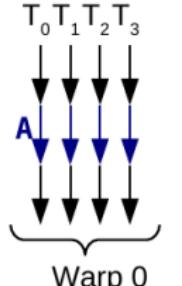
Branch divergence

Branch divergence

- Conditional block

```
if(c) {  
    // A  
}  
else {  
    // B  
}
```

- All threads of a warp take the same path



With imaginary 4-thread warps

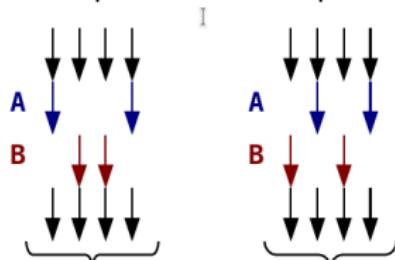
Branch divergence

Branch divergence

- Conditional block

```
if(c) {  
    // A  
}  
else {  
    // B  
}
```

- Threads in a warp take different paths



- Warps have to go through both A and B: lower performance

Data structure

Avoiding branch divergence

- Hoist identical computations and memory accesses outside conditional blocks

```
if(tid % 2) {                                float t = 1.0f/tid;
    s += 1.0f/tid;                            if(tid % 2) {
}                                              s += t;
else {                                         }
    s -= 1.0f/tid;                           else {
}                                              s -= t;
}
```

- When possible, re-schedule work to make non-divergent warps

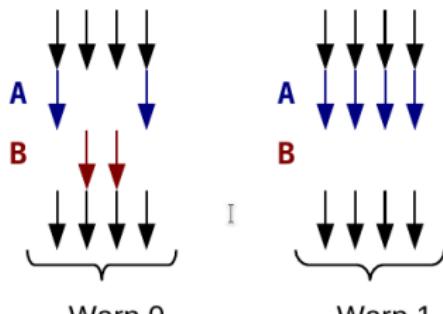
```
// Compute 2 values per thread
int i = 2 * tid;
s += 1.0f/i - 1.0f/(i+1);
```

- What if I use C's ternary operator (?:) instead of if?
(or tricks like ANDing with a mask, multiplying by a boolean...)

Branch divergence

Ternary operator ? good : bad

- Run both branches and select: $R = c ? A : B;$
 - No more divergence?
- All threads have to take both paths
No matter whether the condition is divergent or not



- Does **not** solve divergence: we lose in all cases!
- Only benefit: fewer instructions
 - May be faster for short, often-divergent branches

NVCC-specific compiler

- NVIDIA provides a specific CUDA-C / C ++ compiler
- NVCC compiles the device code (GPU) and transmits the host code (CPU) to the host compiler (gcc / clang / ...)
- NVCC is based on LLVM

Code example CUDA 1 : Hello World

```
--global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("HelloWorld!\n");  
    return 0;  
}
```

```
$ nvcc main.cu  
$ ./a.out  
Hello World!
```

Debugging tools

NSIGHT



CUDA-GDB



CUDA MEMCHECK



NVIDIA Provided

allinea
DDT

TotalView®

3rd Party

[http://on-demand.gputechconf.com/gtc/2014/presentations/
S4578-cuda-debugging-command-line-tools.pdf](http://on-demand.gputechconf.com/gtc/2014/presentations/S4578-cuda-debugging-command-line-tools.pdf)

printf

- Same format as printf C
- Buffered output: Flush only at explicit synchronization points
- Unordered output: As for a multi-threaded program
- Possible to change the buffer size:

```
cudaDeviceSetLimit(cudaLimitPrintFifoSize, size_t size);
```

Compilation flags

- Be careful, there are 2 compilers to use
 - NVCC: device code
 - Host compiler: C / C ++ code
- NVCC supports some of the host's compilation flags
 - But if the flag is not supported, it is possible to transfer it to the host's compiler via `-Xcompiler`
 - Example: `-Xcompiler -fopenmp`
- Debugging flags
 - `-g`: Include debugging symbols for host code
 - `-G`: Include debugging symbols for device code (disables optimizations)
item `-lineinfo`: Include line information (in code) in symbols (no impact on optimization)

cuda-memcheck : valgrind memory tools for CUDA

- Memory debugging tools
 - Does not require recompilation: `Istinline cuda-memcheck ./exe`
- Can detect the following errors
 - Memory leaks
 - Memory errors (Out Of Bounds, misaligned access, illegal instruction, etc.)
 - Competition situation
 - Illegal barriers
 - Memory not initialized
- To activate the display of the line where the error is located:
 - `-Xcompiler -rdynamic -lineinfo`

cuda-memcheck: Example 1

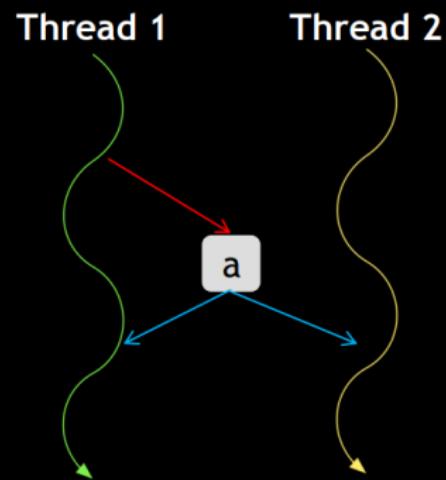
By default, detection of misaligned or out of bounds memory accesses

```
Invalid __global__ read of size 4
at 0x000000b8 in basic.cu:27:kernel2
by thread (5,0,0) in block (3,0,0)
Address 0x05500015 is misaligned
```

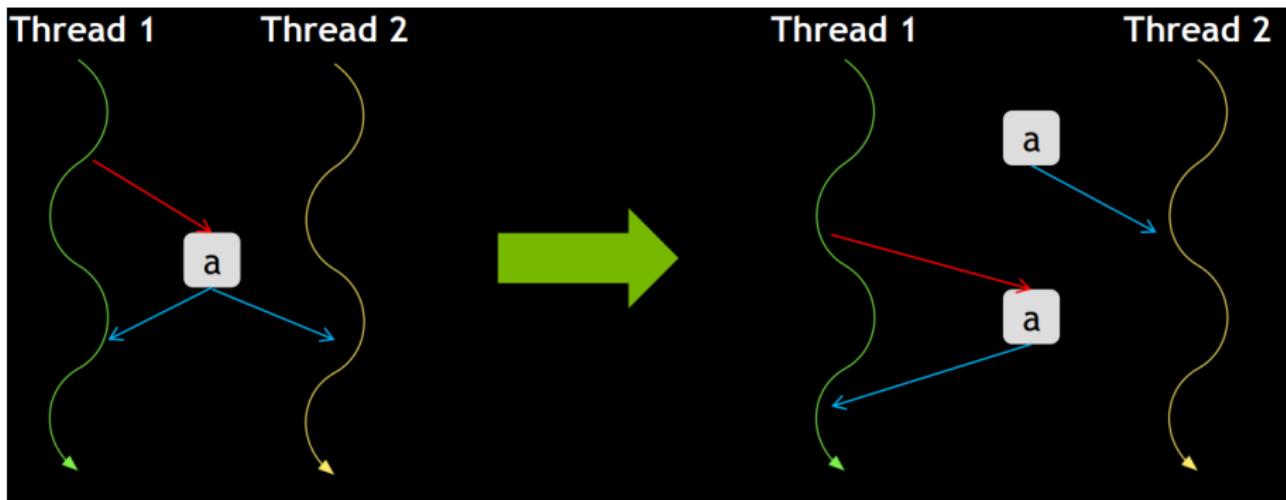
```
Malloc/Free error encountered : Double free
at 0x0002de18
by thread (1,0,0) in block (0,0,0)
Address 0x50c8b99a0
```

cuda-memcheck: Race condition example

```
__global__ int bcast(void) {
    int x;
    __shared__ int a;
    if (threadIdx.x == WRITER)
        a = threadIdx.x;
    x = a;
    // do some work
}
```



cuda-memcheck: Data sharing between 2 threads



- Thread 2 can read the variable *a* before or after modification
- Requires an order
- `cuda-memcheck --tool racecheck --racecheck-report analysis`

cuda gdb : gdb for CUDA

- cuda-gdb is an extension for CUDA to GDB
 - Allows you to debug the CUDA and CPU code at the same time
- Works on Linux and Mac OSX
- For Windows, use NSIGHT Visual Studio Edition

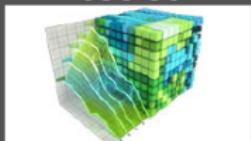
```
(cuda-gdb) b main                      // set break point at main
(cuda-gdb) r                           // run application
(cuda-gdb) l                           // print line context
(cuda-gdb) b foo                        // break at kernel foo
(cuda-gdb) c                           // continue
(cuda-gdb) cuda thread                 // print current thread
(cuda-gdb) cuda thread 10              // switch to thread 10
(cuda-gdb) cuda block                  // print current block
(cuda-gdb) cuda block 1                // switch to block 1
(cuda-gdb) d                           // delete all break points
(cuda-gdb) set cuda memcheck on      // turn on cuda memcheck
(cuda-gdb) r                           // run from the beginning
```

Profiling tools

NSIGHT



NVVP

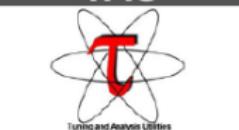


NVPROF

```
#=2561= Profiling result:
Time: 1.2561ms Avg: 1.2561ms Max: 1.2561ms Min: 1.2561ms
49.80% 166.49ms 160708 1.7278us 1.549ms 2.466us 1.2561ms
dev_0: Memset :detail:Ldevice_generate_FuncSet->detail()::FSet
23.23% 1.2561ms 120000 1.2561ms 1.2561ms 1.2561ms 1.2561ms
1.11% 1.2561ms 10000 1.2561ms 1.2561ms 1.2561ms 1.2561ms
1.00% 1.2561ms 1000 1.2561ms 1.2561ms 1.2561ms 1.2561ms
1.00% 1.2561ms 100 1.2561ms 1.2561ms 1.2561ms 1.2561ms
17.87% 296.48ms 296 1.4539ms 1.249ms 1.715ms kernelCopy
2.38% 1.2561ms 200 1.2561ms 1.2561ms 1.2561ms 1.2561ms
1.56% 38.17ms 381 48.26ms 52.9ms 17.47ms CUDA-1
8.83% 14.18ms 266 84.99ms 71.88ms 64.74ms kernelCall
8.79% 1.2561ms 100 1.2561ms 1.2561ms 1.2561ms 1.2561ms
8.49% 12.87ms 295 66.37ms 35.08ms 41.38ms kernels
0.03% 10.39ms 296 54.50ms 31.08ms 56.10ms kernels
0.26% 1.2561ms 296 27.36ms 22.35ms 31.53ms CUDA-1
0.12% 1.2561ms 1 2.1542ms 2.1542ms 2.1542ms 1.2561ms
```

NVIDIA Provided

TAU



VampirTrace



3rd Party

nvprof

A command line profiler

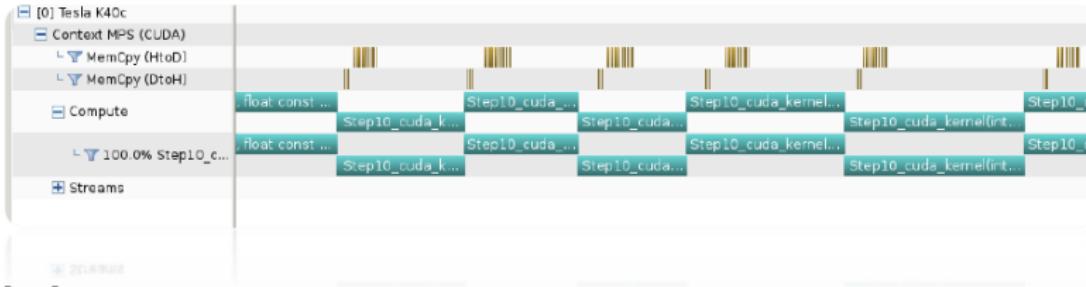
- Calculate the execution time of each kernel
- Calculate transfer time from / to device memory
- Collection of metrics and events
- Supports complex process hierarchies
- Collect the necessary profiles for NVIDIA Visual Profiler
- Does not require recompilation

Example of using nvprof

- Basic information: `nvprof`
- List the available metrics: `nvprof --query-metrics`
- Efficiency of data access:
`nvprof --metrics gld_efficiency, gst_efficiency`
- Create a timeline that can be loaded by NVVP:
`nvprof -o profile.timeline`
- Create a trace of the analysis metrics to read in NVVP:
`nvprof -o profile.metrics --analysis-metrics`

NVIDIA Visual Profiler (NVP)

Timeline



Guided System

1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or memory latency. The numbers at right indicate that the performance of kernel "Step10_cuda_kernel" is most likely limited by compute.

Perform Compute Analysis

The next likely bottleneck to performance for this kernel is compute so you can first perform compute analysis to determine how it is limiting performance.

Perform Latency Analysis

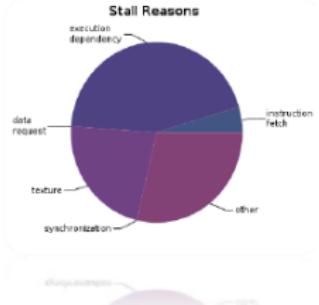
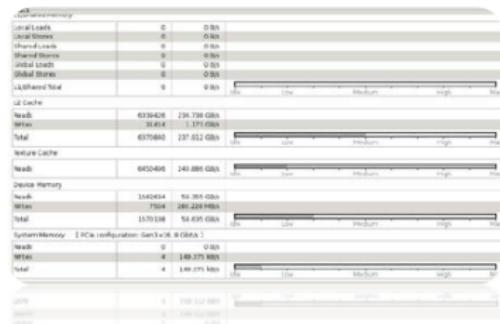
Perform Memory Bandwidth Analysis

Instruction and memory latency and memory bandwidth are likely to be the primary bottlenecks for this kernel, but you may still want to perform those analyses.

Run Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Analysis

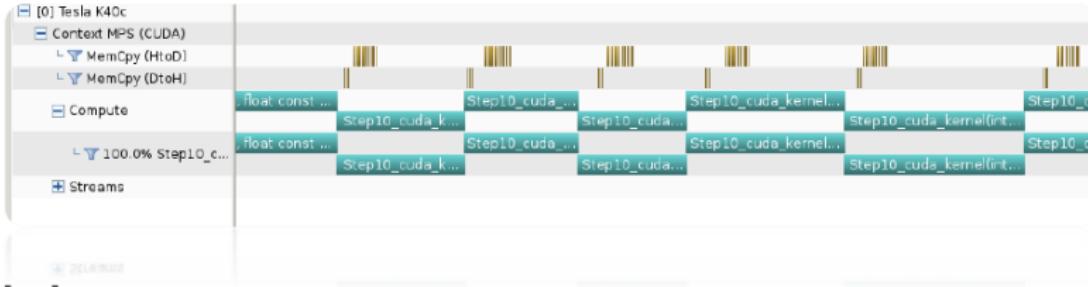


NVTX

- Increase the traces of execution
- The NVTX library allows you to add annotations
 - Add: `#include<nvToolsExt.h>`
 - And add the link to the compilation: `-lnvToolsExt`
- To mark the start of an annotation:
`nvtxRangePushA("description");`
- To mark the end of an annotation: `nvtxRangePop();`
- Ability to overlap annotations

NVTX Profile

Timeline



Guided System

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or memory latency. The bars at right indicate that the performance of kernel "Step10_cuda_kernel" is most likely limited by compute.

[Perform Compute Analysis](#)

The next step is to analyze the performance for this kernel as compute is very fast. First perform compute analysis to determine how it is limiting performance.

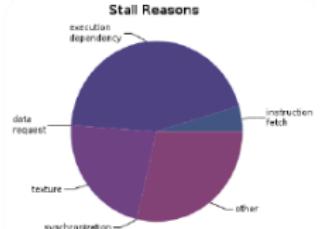
[Perform Latency Analysis](#)

Instruction and memory latency and memory bandwidth are likely to be the primary bottlenecks for this kernel, but you may still want to perform these analyses.

[Perform Memory Bandwidth Analysis](#)

If you modify the kernel you need to rerun your application to update this analysis.

Analysis

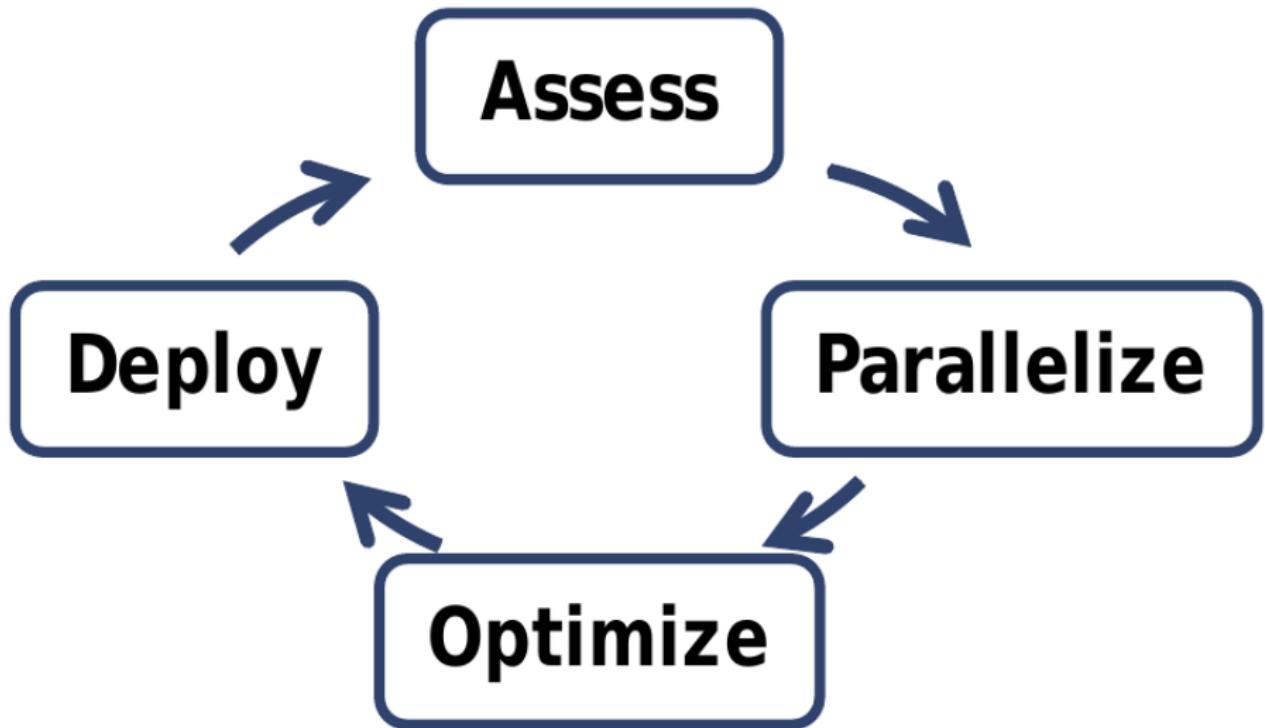


NSIGHT

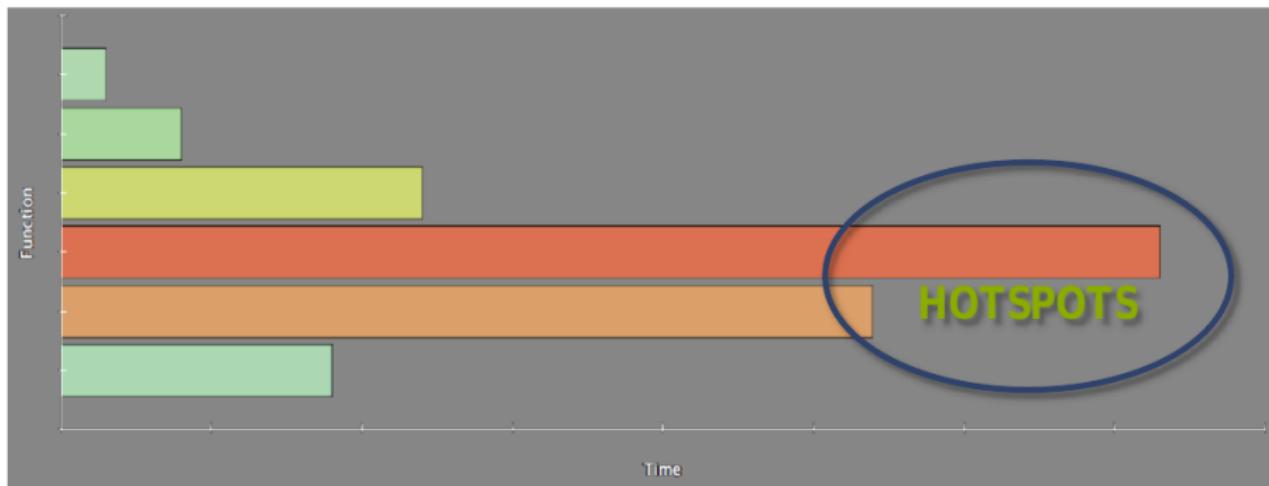
- Specific IDE for CUDA
 - Editing source code: syntax colorization, refactoring code, etc.
 - Compilation chain
 - Visual Debugger
 - Visual Profiler
- Linux / Macintosh
 - The editor is Eclipse
 - The debugger is cuda-gdb with a graphical interface
 - The profiler is NVVP
- Windows
 - Integrates directly into Visual Studio
 - The profiler is NSIGHT VSE



CUDA Optimization



CUDA Optimization: Evaluation



- Profile your code and find the hot spot (s)
- Focus on the points that will have the most benefit

CUDA Optimization: Parallelism

Applications

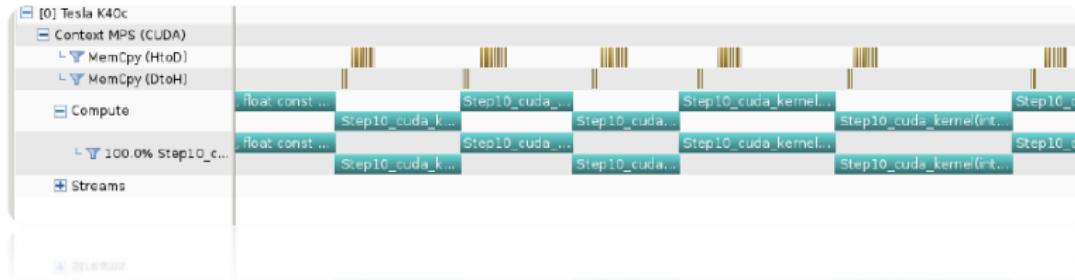
Libraries

Compiler
Directives

Programming
Languages

CUDA Optimization: Optimization

Timeline



Guided System

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound

The first step in analysing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction latency. The following metrics at right indicate that the performance of kernel "Step10_cuda_kernel" is most likely limited by compute.

Perform Compute Analysis

The most likely bottleneck performance for this kernel is compute. You should now perform compute analysis to determine how it is limiting performance.

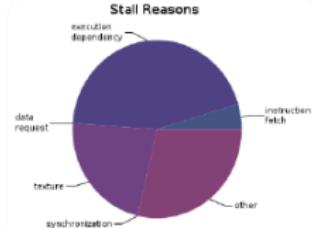
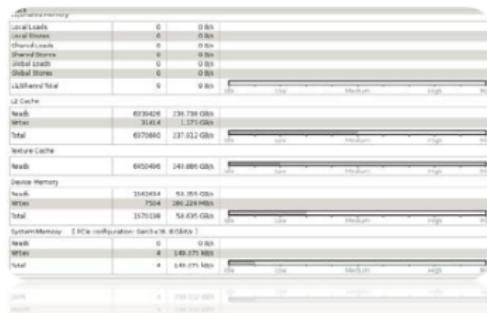
Perform Latency Analysis

Memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform these analyses.

Perform Memory Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Analysis



CUDA Optimization: Bottleneck analysis

- Don't assume an optimization was wrong
- Verify if it was wrong with the profiler

129 GB/s → 84 GB/s

L1/Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	2097152	1,351.979 GB/s
Shared Stores	131072	84.499 GB/s
Global Loads	131072	42.249 GB/s
Global Stores	131072	42.249 GB/s
Atomic	0	0 B/s
L1/Shared Total	2490388	1,520.977 GB/s

Idle Low Medium

gpuTranspose_kernel(int, int, float const *, float *)	
Start	547.303 ms (
End	547.716 ms (
Duration	413.872 µs
Grid Size	[64,64,1]
Block Size	[32,32,1]
Registers/thread	10
Shared Memory/Block	4 kB
Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	▲ 5.9%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficiency	97.1%
Occupancy	
Achieved	86.7%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 kB
Shared Memory Executed	48 kB

Shared Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

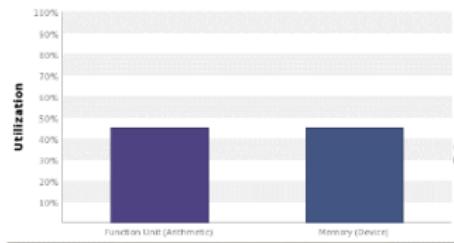
Optimization: Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each access pattern of the memory access.

▼ Line / File main.cu - /home/jluitjens/code/CudaHandsOn/Example19

49 Shared Load Transactions/Access = 16, Ideal Transactions/Access = 1 [2097152 transactions for 131072 total executions]

CUDA Optimization: Performance analysis

<code>gpuTranspose_kernel(int, int, float const *, float</code>	
Start	770.067
End	770.324
Duration	256.714
Grid Size	[64,64,1
Block Size	[32,32,1
Registers/thread	10
Shared Memory/Block	4.125 KiB
Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	⚠ 50%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficiency	97.1%
Occupancy	
Achieved	87.7%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB



84 GB/s → 137 GB/s

L1/Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	131072	138.433 GB/s
Shared Stores	131720	139.118 GB/s
Global Loads	131072	69.217 GB/s
Global Stores	131072	69.217 GB/s
Atomic	0	0 B/s
Total	524936	413.594 GB/s

Idle Low Medium

L2 Cache		
L1 Reads	524288	69.217 GB/s
L1 Writes	524288	69.217 GB/s
Texture Reads	0	0 B/s
Atomic	0	0 B/s
Noncoherent Reads	0	0 B/s
Total	1048576	138.433 GB/s

Idle Low Medium

Texture Cache		
Reads	0	0 B/s

Idle Low Medium

Device Memory		
Reads	524908	69.395 GB/s
Writes	524289	69.217 GB/s
Total	1049237	138.523 GB/s

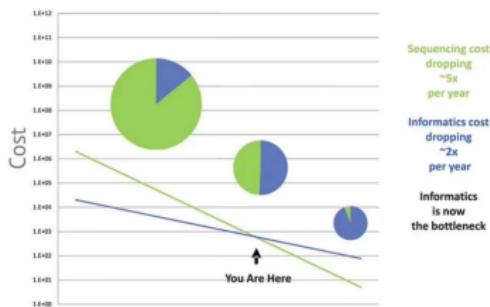
Idle Low Medium

CUDA and genomics

Dynamic Programming

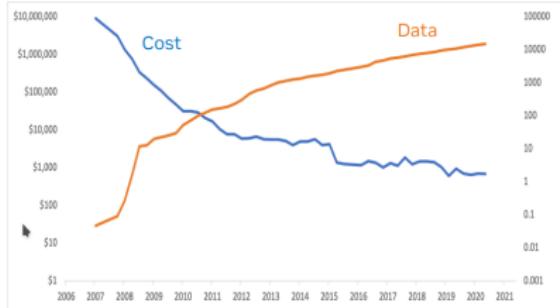
Why accelerating Sequence alignment matters?

DNA Sequencing Economics



Cost of sequencing dropping twice as fast as informatics

Source: <https://www.businessinsider.com.au/super-cheap-genome-sequencing>



1000 Genomes
Project launches
(2008)

100,000 Genomes
Project launches
(2013)

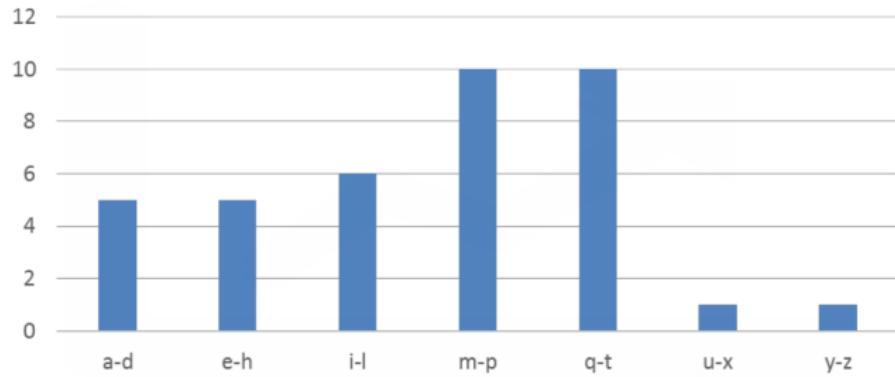
1,000,000 Genomes
Project launches
(2018)

Histogram

- A feature and model extraction method for large data sets
 - Feature extraction for object recognition in images
 - Fraud detection in credit card transactions
 - Correlation of celestial objects movements in astrophysics
 - ...
- Basic operation of histograms: For each element in the data set, use a value to identify a bag/characteristic to be incremented

Example: a text histogram

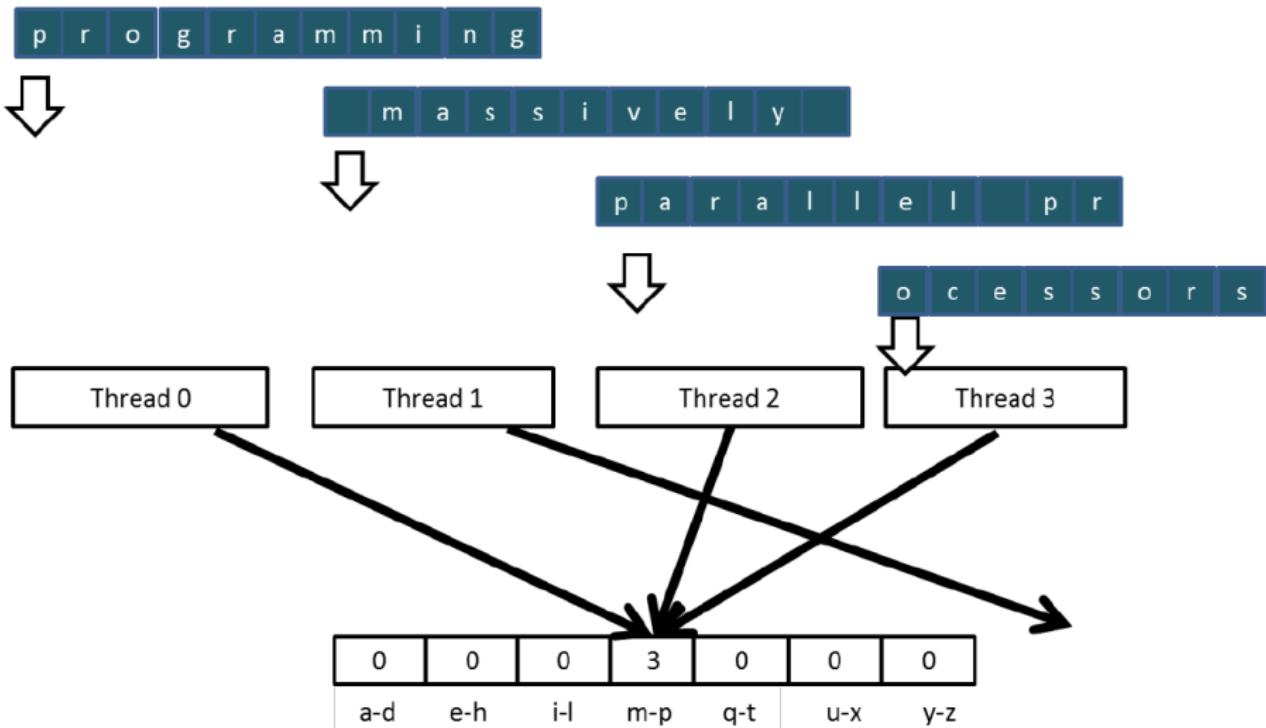
- Define bags as sets of 3 letters of the alphabet: a-d, e-h, i-l, n-p, ...
- For each character in the input string, increment the corresponding bag counter
- Example of results for the phrase "Programming Massively Parallel Processors"



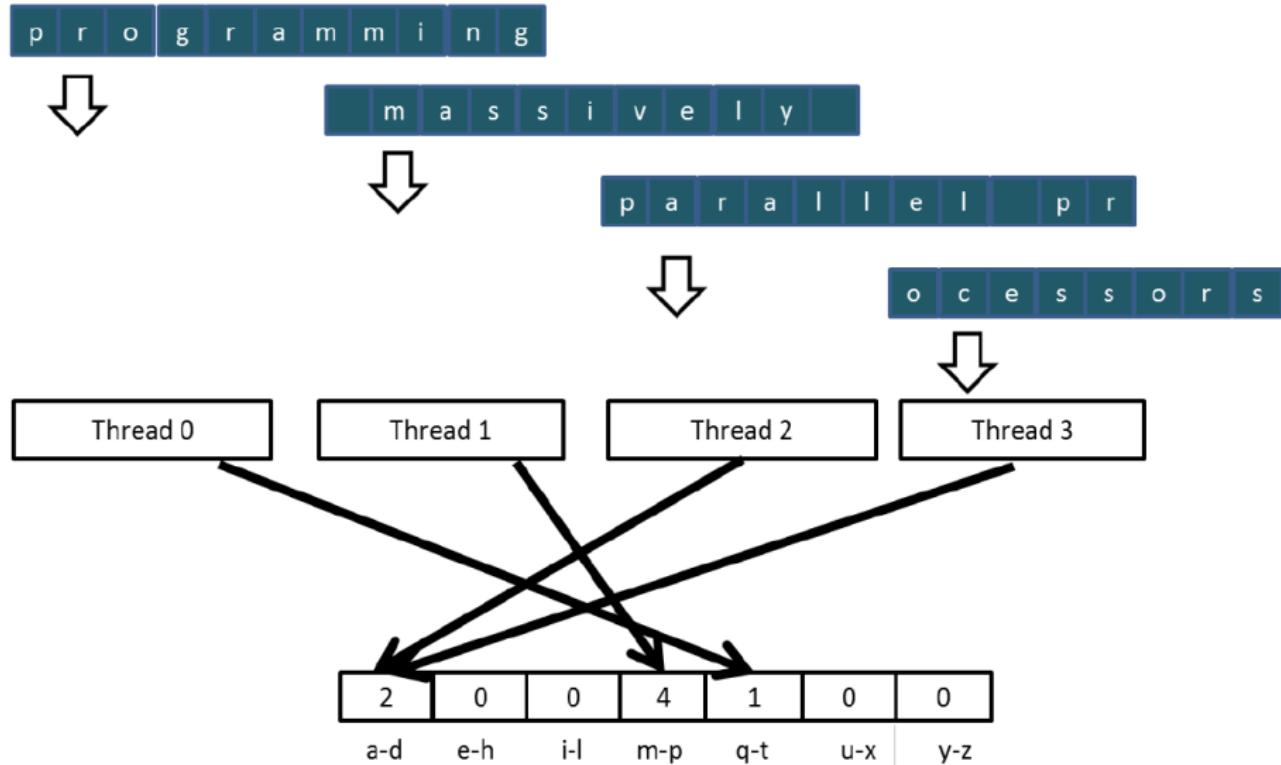
Simple algorithm for parallelizing a histogram

- Partitioning of the entrance in sections
- Each thread is in charge of processing a section
- Each thread runs through its section
- For each letter, the counter of the corresponding bag is incremented

Partitioned sections (Iteration #1)

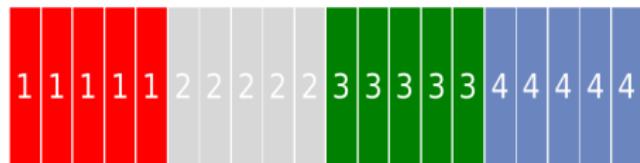


Partitioned sections (Iteration #2)



Effects of partitioning on the efficiency of memory accesses

- Partitioning into sections can have a negative impact on memory access efficiency
 - Adjacent threads do not access adjacent memory addresses
 - Accesses are not coalescing
 - Memory bandwidth is not used efficiently



Effects of partitioning on the efficiency of memory accesses

- Partitioning into sections can have a negative impact on memory access efficiency
 - Adjacent threads do not access adjacent memory addresses
 - Accesses are not coalescing
 - Memory bandwidth is not used efficiently

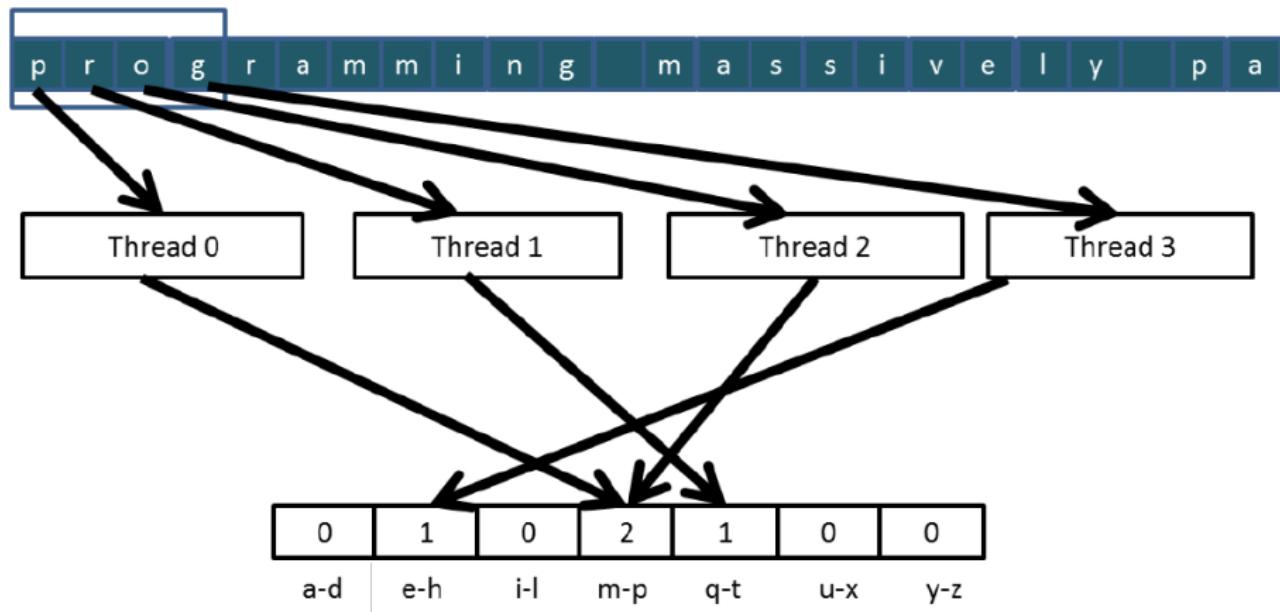


- Change to an interlaced partitioning algorithm
 - All threads compute a contiguous section of elements
 - They all move to the next section and start again
 - The memory is accessed coalescently



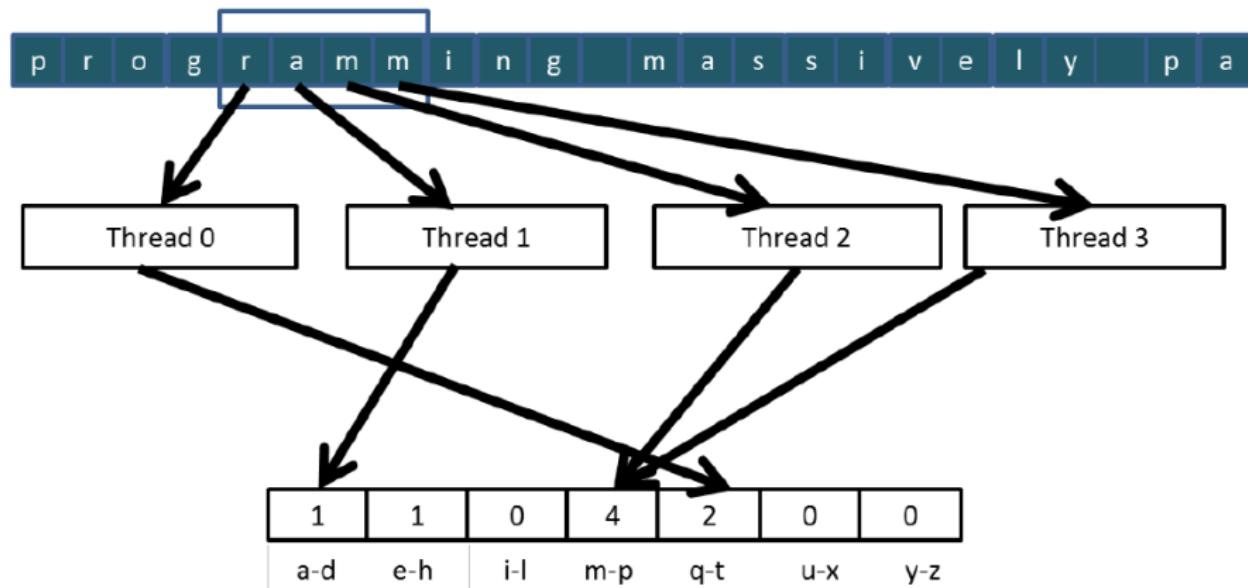
Interleaved input partitioning (Iteration #1)

For coalescing memory access and better performance



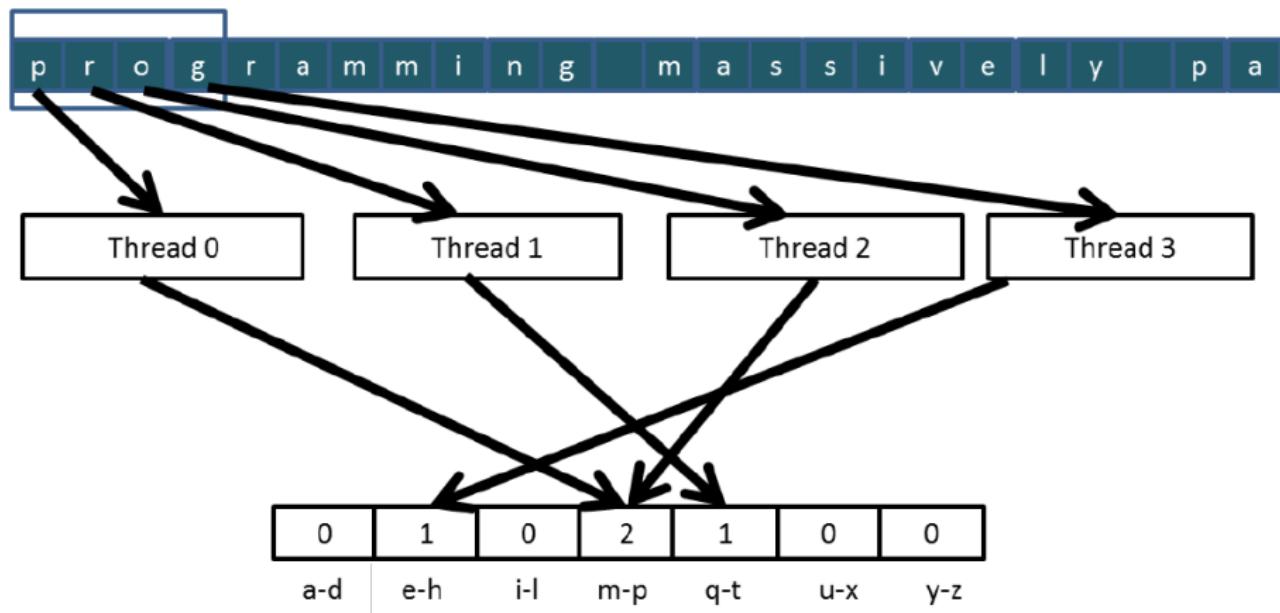
Interleaved input partitioning (Iteration #2)

For coalescing memory access and better performance



Example of histogram: Read-Modify-Write operation

For coalescing memory access and better performance



The Read-Modify-Write operation used in collaborative models

- For example, several cashiers count the total amount of cash in a safe
- Each person takes a pile and counts it
- There is a centralized display of the amount already calculated
- Each time someone finishes counting a stack, he reads the current total sum (read), adds the subtotal of his stack (modify-write)
- A recurring problem: The subtotal of some stacks is not present in the final sum

A common and parallel characterization service

- For example, several customer service personnel serve in a shared queue
- The system must maintain 2 numbers:
 - the number to give to the next person entering the queue (I)
 - the number of the next person to be served (S)
- The system gives a number to each person entering the queue (read I) and increments this number by 1 to give it to the next incoming customer (modify-write I)
- A display panel shows the number of the next customer to be served
- When a staff is available, it calls the number (read S) and increments the displayed number by 1 (modify-write S)
- Problems
 - Several customers receive the same number, only one is served
 - Multiple staff serve the same number

A common arbitration rule

- For example, several customers book airline tickets in parallel
- Everyone
 - Look at the map of the seats in the plane (reading)
 - Decide which seat they want
 - Update the seating plan and mark the selected one as taken (edit-write)
- Problems
 - Several passengers may have reserved the same seat

Race condition when running threads in parallel

thread1: $Old \leftarrow Mem[x]$
 $New \leftarrow Old + 1$
 $Mem[x] \leftarrow New$

thread2: $Old \leftarrow Mem[x]$
 $New \leftarrow Old + 1$
 $Mem[x] \leftarrow New$

- *Old* and *New* are variables allocated in registers and therefore specific to a thread
- Question : If $Mem[x]$ was initialized to 0; what would be the value of $Mem[x]$ after threads 1 and 2 have finished their execution ?
- Question : What do threads have in their *Old* variable ?

Race condition when running threads in parallel

thread1: $Old \leftarrow Mem[x]$
 $New \leftarrow Old + 1$
 $Mem[x] \leftarrow New$

thread2: $Old \leftarrow Mem[x]$
 $New \leftarrow Old + 1$
 $Mem[x] \leftarrow New$

- *Old* and *New* are variables allocated in registers and therefore specific to a thread
- Question : If *Mem[x]* was initialized to 0; what would be the value of *Mem[x]* after threads 1 and 2 have finished their execution ?
- Question : What do threads have in their *Old* variable ?
- Unfortunately, the answer can vary according to the relative timing between the execution of the 2 threads, we are in a situation of competition on the data

Scenario #1 of thread timing

Temps	Thread 1	Thread 2
1	(0) $Old \leftarrow Mem[x]$	
2	(1) $New \leftarrow Old + 1$	
3	(1) $Mem[x] \leftarrow New$	
4		(1) $Old \leftarrow Mem[x]$
5		(2) $New \leftarrow Old + 1$
6		(2) $Mem[x] \leftarrow New$

- Thread 1 $Old = 0$
- Thread 2 $Old = 1$
- $Mem[x] = 2$ after the execution sequence

Scenario #2 of thread timing

Temps	Thread 1	Thread 2
1		(0) $Old \leftarrow Mem[x]$
2		(1) $New \leftarrow Old + 1$
3		(1) $Mem[x] \leftarrow New$
4	(1) $Old \leftarrow Mem[x]$	
5	(2) $New \leftarrow Old + 1$	
6	(2) $Mem[x] \leftarrow New$	

- Thread 1 $Old = 1$
- Thread 2 $Old = 0$
- $Mem[x] = 2$ after the execution sequence

Scenario #3 of thread timing

Temps	Thread 1	Thread 2
1	(0) $Old \leftarrow Mem[x]$	
2	(1) $New \leftarrow Old + 1$	
3		(1) $Old \leftarrow Mem[x]$
4	(1) $Mem[x] \leftarrow New$	
5		(1) $New \leftarrow Old + 1$
6		(1) $Mem[x] \leftarrow New$

- Thread 1 $Old = 0$
- Thread 2 $Old = 0$
- $Mem[x] = 1$ after the execution sequence

Scenario #4 of thread timing

Temps	Thread 1	Thread 2
1		(0) $Old \leftarrow Mem[x]$
2		(1) $New \leftarrow Old + 1$
3	(0) $Old \leftarrow Mem[x]$	
4		(1) $Mem[x] \leftarrow New$
5	(1) $New \leftarrow Old + 1$	
6	(1) $Mem[x] \leftarrow New$	

- Thread 1 $Old = 0$
- Thread 2 $Old = 0$
- $Mem[x] = 1$ after the execution sequence

The purpose of atomic operations: to ensure accurate results

```
thread1: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

```
thread2: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

OU

```
thread1: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

```
thread2: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

Basic concepts of atomic operations

- The Read-Modify-Write operation is performed at a memory address by a single hardware instruction
 - Read the old value, calculate the new value, and write the new value to the designated memory address
- The hardware guarantees that no other thread can perform another Read-Modify-Write operation at the same address until the end of the atomic operation
 - All other threads that try to perform an atomic operation on the same address will be held in a queue
 - All threads that perform an atomic operation at the same address will be serialized

Atomic operations in CUDA

- Atomic operations are accessed by calling specific functions that are translated into single instructions (also called *intrinsic functions* or *intrinsics*)
 - Atomic *add*, *sub*, *inc*, *dec*, *min*, *max*, *exch* (exchange), *CAS* (compare and exchange)
 - Read the CUDA documentation for more details
- Atomic addition function : `int atomicAdd(int* address, int val);`
- Reads a 32-bit word *old* from the address pointed to by *address* in global or shared memory, computes $(old + val)$ and writes the result to memory at the same address. The function returns *old*.

Other atomic addition functions in CUDA

- Atomic addition of 32-bit unsigned integers

```
unsigned int atomicAdd(unsigned int* adress, unsigned int val);
```

- Atomic addition of 64-bit unsigned integers

```
unsigned long long int atomicAdd(unsigned long long int* adress,
```

- Atomic addition of single precision floating numbers

```
float atomicAdd(float* adress, float val);
```

A simple example of a histogram core on text (1/2)

- The kernel receives a pointer to an input buffer
- Each thread is in charge of computing a subset of the input buffer

```
--global__ void histo_kernel(unsigned char *buffer,
                           long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    // All threads handle blockDim.x * gridDim.x
    // consecutive elements
    while (i < size) {
        atomicAdd( &(histo[buffer[i]]), 1);
        i += stride;
    }
}
```

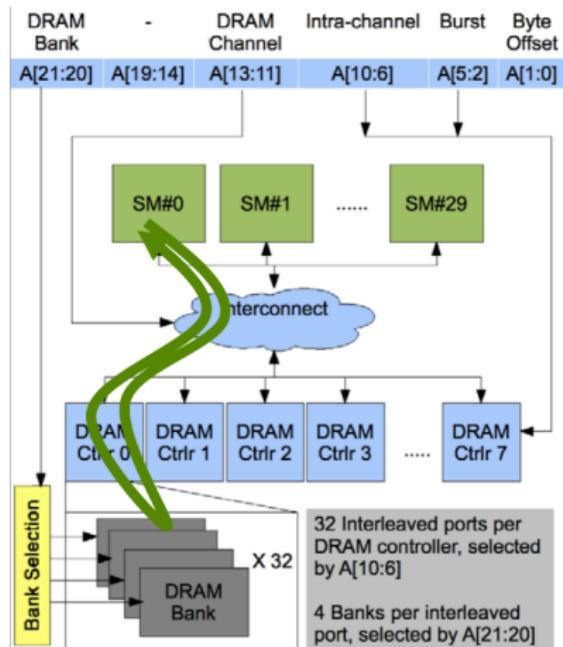
A simple example of a histogram core on text (2/2)

- The kernel receives a pointer to an input buffer
- Each thread is in charge of computing a subset of the input buffer

```
--global__ void histo_kernel(unsigned char *buffer,
                           long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    // All threads handle blockDim.x * gridDim.x
    // consecutive elements
    while (i < size) {
        int alphabet_position = buffer[i] - "a"
        if (alphabet_position >= 0 && alpha_position < 26)
            atomicAdd(&(histo[alphabet_position / 4]), 1);
        i += stride;
    }
}
```

Atomic operations in global memory (DRAM) (1/2)

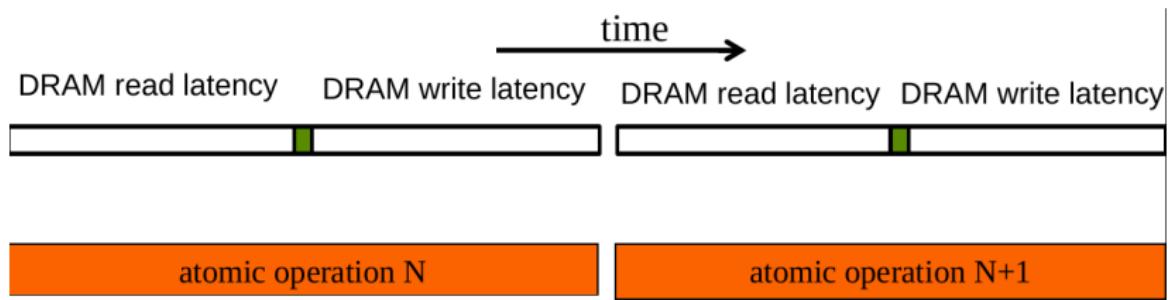
- An atomic operation on a DRAM address starts with a read that has a latency of a few hundred cycles
- An atomic operation ends with a write on the same address with also a latency of a few hundred cycles
- During this time, no one else can access this address



Atomic operations in global memory (DRAM) (2/2)

Each Read-Modify-Write operation pays 2 latencies of access to the memory

- All atomic operations on the same variable (DRAM address) are serialized



Latency determines bandwidth

- The bandwidth of atomic operations on the same memory address in DRAM can be defined as the speed at which each application can execute an atomic operation
- The speed of atomic operations on an address is limited by the total latency of the Read-Modify-Write sequence, usually more than 1000 cycles for addresses in global memory (DRAM)
- This means that if several threads try to perform atomic operations on the same address (contention), the memory bandwidth is reduced to less than $\frac{1}{1000}$ of the maximum bandwidth of a single memory channel!

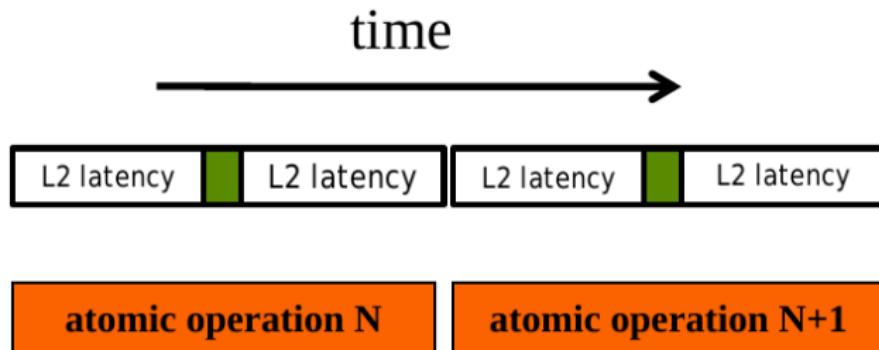
Similar example: Checkout in a supermarket

- Some customers realize they have forgotten an item after they have started to check out
- They run down the aisles and retrieve the missing item while the line is waiting
 - The speed of the checkout process is drastically reduced due to the long latency of running through the aisles of the supermarket and back to the checkout
- Imagine a store where every customer starts lining up before they start picking up items
 - The checkout speed will be 1 on the total running time of each customer

Hardware improvement: Atomic operation in L2 cache

Atomic operation in L2 cache

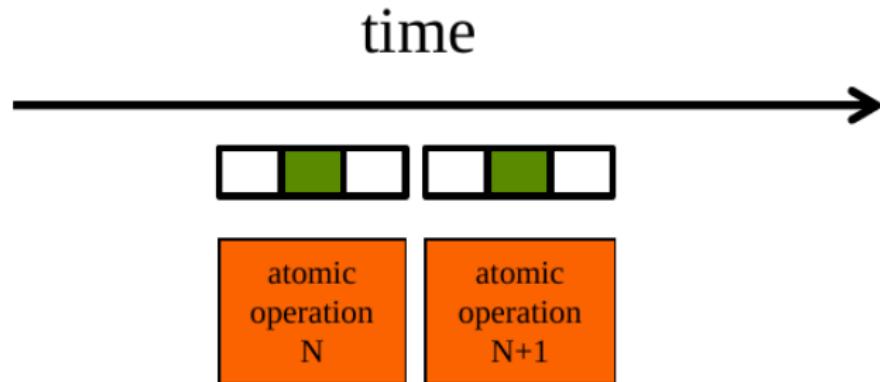
- Average latency, about $\frac{1}{10}$ of that of DRAM
- Shared between all blocks
- Almost no development/algorithmic cost compared to global memory



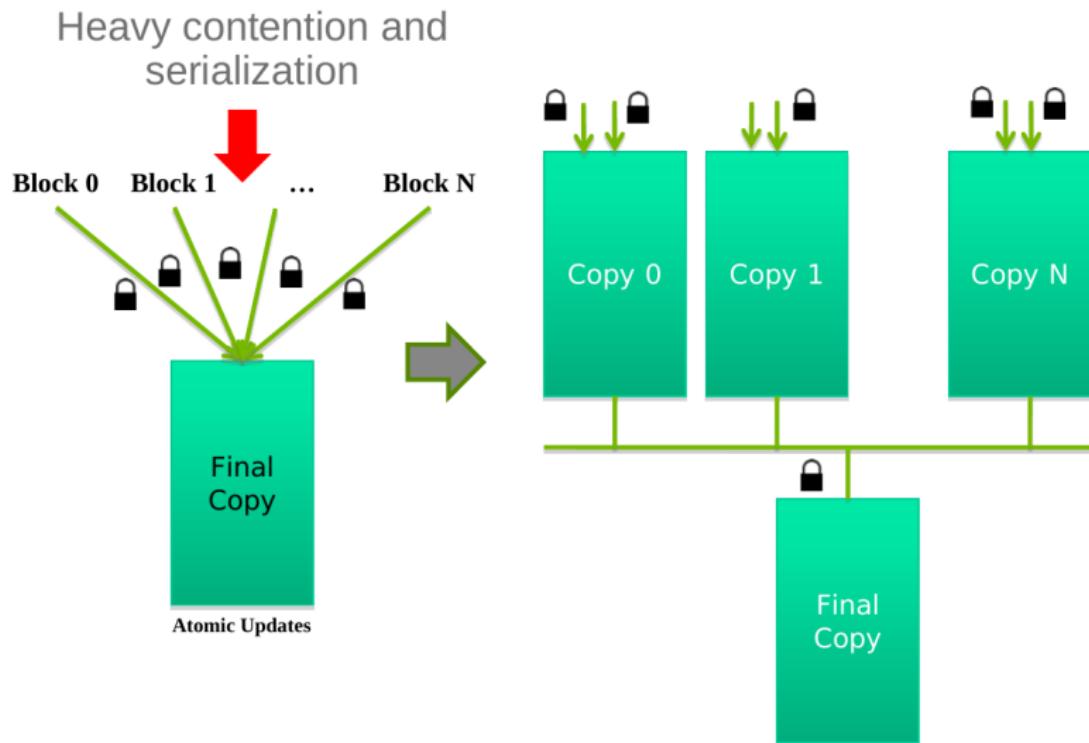
Hardware improvement: Atomic operation in shared memory

Atomic operation in shared memory

- Very low latency
- Private to each block of threads
- Little required a change of algorithm

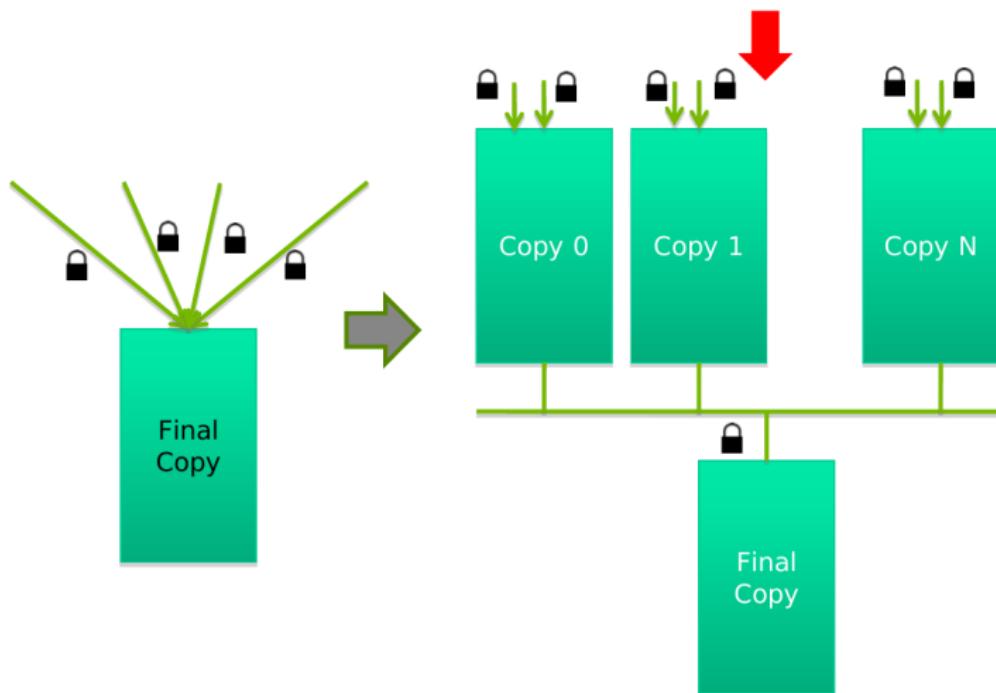


Privatization (1/3)

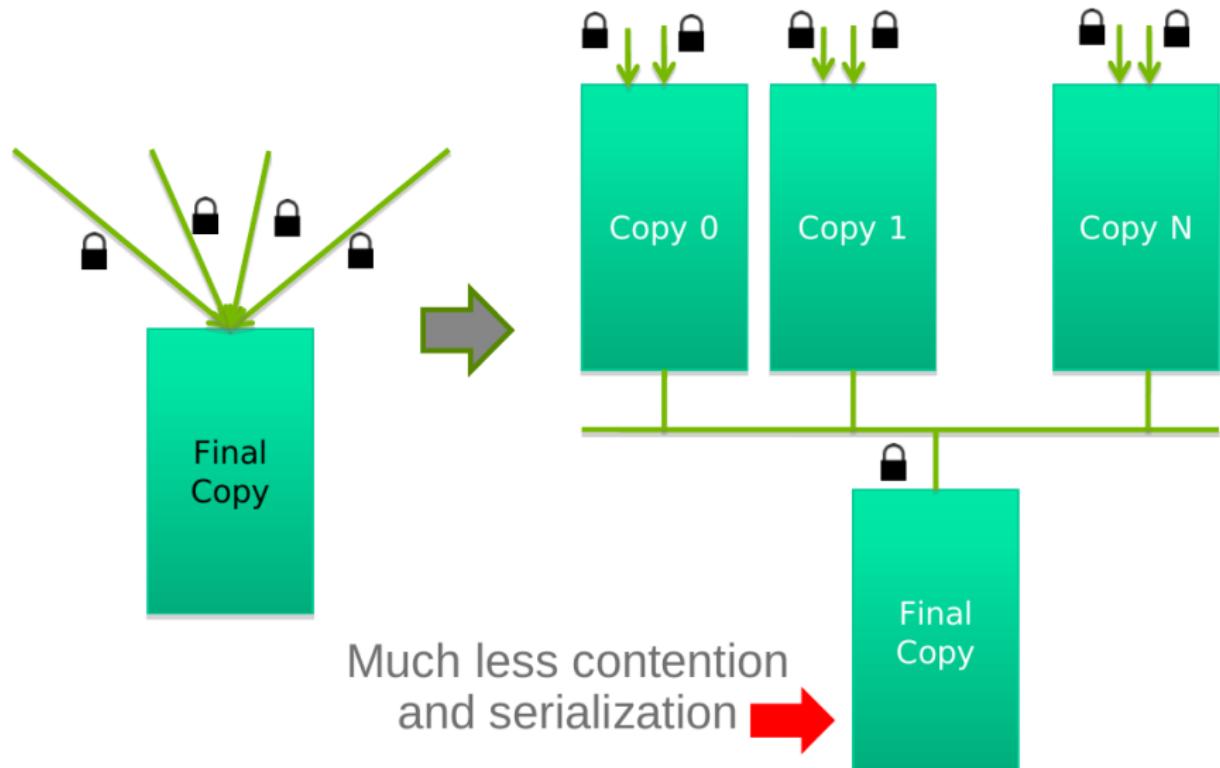


Privatization (2/3)

Much less contention and serialization



Privatization (3/3)



Cost and benefit of privatization

- Cost

- overhead for the creation and initialization of private copies
- overhead for the accumulation of the content of the different private copies in the final copy

- Profit

- Much less contention and serialization when accessing private copies and the final copy
- Total performance can often be improved by more than 10x

Atomic operations in shared memory for histograms

- Each subset of threads are in the same block
- Much more bandwidth than for atomic operations in DRAM (100x) or L2 (10x)
- Much less contention: only threads of the same block can access the variable in shared memory
- This is one of the major cases of use of shared memory

Atomic operation in shared memory requires privatization

Create private copies of the *histo[]* array for each block of threads

```
--global__ void histo_kernel(unsigned char *buffer,
                           long size, unsigned int *histo)
{
    __shared__ unsigned int histo_private[7];
```

Atomic operation in shared memory requires privatization

Initialize feature counters in the private copy of *histo[]*

```
--global__ void histo_kernel(unsigned char *buffer,
                           long size, unsigned int *histo)
{
    --shared-- unsigned int histo_private[7];

    if (threadIdx.x < 7) histo_private[threadIdx.x] = 0;
    __syncthreads();
```

Build a private histogram

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
// stride is total number of threads
int stride = blockDim.x * gridDim.x;
while (i < size) {
    atomicAdd( &(private_histo[buffer[i]/4]), 1);
    i += stride;
}
```

Build the final histogram

```
// wait for all other threads in the block to finish
__syncthreads();
if (threadIdx.x < 7) {
    atomicAdd(&(histo[threadIdx.x]), private_histo[threadIdx.x])
}
}
```

More information on privatization

- Privatization is a powerful and frequently used technique for application parallelization (beyond CUDA)
- The operations must be associative and commutative
 - The add operation (*add*) of the histogram is associative and commutative
 - No privatization if the operation does not meet these criteria
- The size of the privatized histogram must be small *i.e.* it holds in the shared memory
- What to do if the histogram is too big to be privatized
 - Sometimes it is possible to partially privatize the output histogram and use ranges to test whether to use the global or shared memory

Definition of the Inclusive Scan or Prefixed Sum

Definition : The operation inclusive scan takes an associative binary operator *oplus* (plus circle) and an array of n elements

$[x_0, x_1, \dots, x_n]$

and returns an array $[x_0, x_0 \oplus x_1, \dots, x_0 \oplus x_1 \oplus \dots \oplus x_n]$

Example : if *oplus* is the addition, then the operation scan on the array should return

3 1 7 0 4 1 6 3,

3 4 11 11 15 16 22 25.

Example of the application of an inclusive scan

- Suppose we have a 100 cm sandwich to feed 10 people
- We know how many inches each person wants:
[3, 5, 2, 7, 28, 4, 3, 0, 8, 1]
- How can we quickly cut the sandwich?
- How many will be left?

- Method 1 : cut the sections sequentially : first 3cm, then 5cm then 2cm ...

- Method 2: Calculate the prefixed sum
[3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (remaining 39cm)

Typical use case of the scan

- The scan is a simple and practical building block for parallelism
 - Convert recurrences sequentially :
`for (j=1;j<n;j++) out[j] = out[j-1] + f(j);`
 - In parallel : `forall(j) temp[j] = f(j) ; scan (out,temp);`

Use a lot of parallel algorithms

- Sort by base
- Quick sort
- Character comparison
- Lexical analysis
- Compression stream
- Polynomial evaluation
- Solve recurrences
- Operation on trees
- Histograms
- ...

Practical use cases

- Assigning places in a campsite
- Assigning places in a market
- Assigning places in fields
- Allocate memory to parallel threads
- Allocate memory buffers for these communication channels
- ...

An inclusive scan for sequential addition

For a sequence $[x_0, x_1, x_2, \dots]$

Compute the result $[y_0, y_1, y_2, \dots]$

Such as

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

Using the recursive definition

$$y_i = y_{i-1} + x_i$$

An efficient implementation in C

```
y[0] = x[0];  
for (i = 1; i < Max_i; i++) y[i] = y[i-1] + x[i];
```

Efficient from a computational point of view:

- N additions needed for N elements: $O(N)$
- Only slightly more expensive than a sequential reduction!

A naive algorithm for parallel inclusive scanning

- Assign a thread to calculate each element of y .
- Make all threads add all elements of x needed to calculate the element of y .
 $y_0 = x_0$

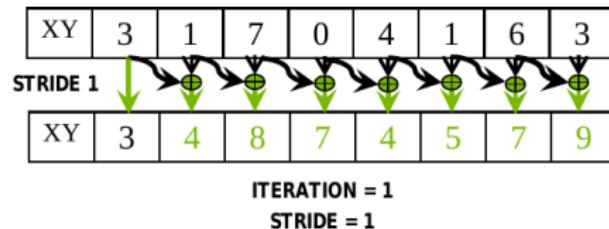
$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

Parallel programming is easy as long as you don't care about performance!

A better parallel scanning algorithm

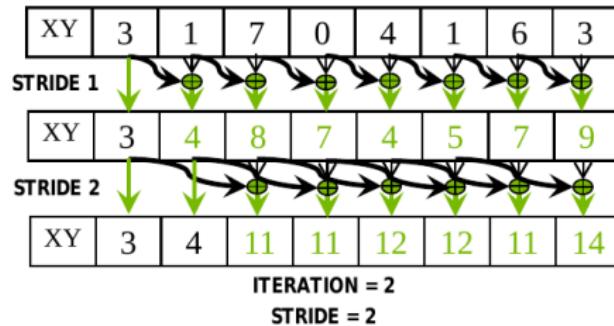
- ① Read input from global memory to shared memory
- ② Repeat $\log(n)$ times: traverse from 1 to $n - 1$: double traverse for each iteration



- Active threads from $stride$ to $n - 1$ ($n - stride$ threads)
- The thread j adds the elements j and $j - stride$ from shared memory and writes the result to the box j in shared memory
- Requires a synchronization barrier, once the read is done and before the write is done

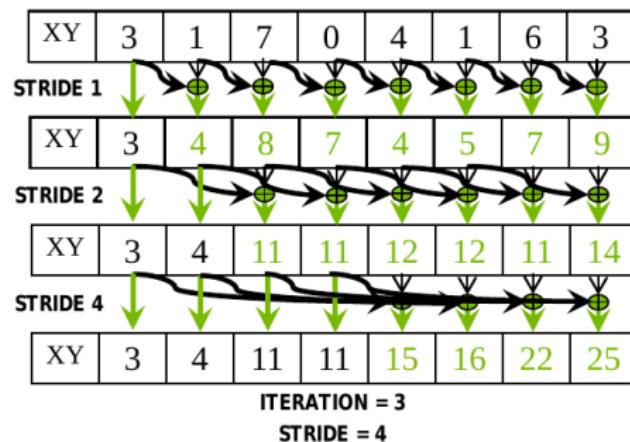
A better parallel scanning algorithm

- ① Read input from global memory to shared memory
- ② Repeat $\log(n)$ times: traverse from 1 to $n - 1$: double traverse for each iteration



A better parallel scanning algorithm

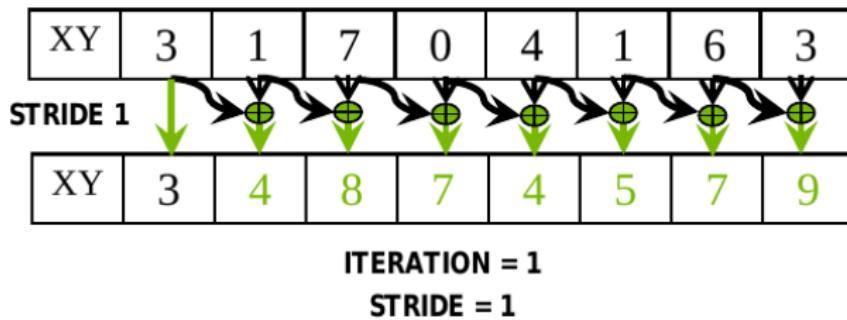
- ① Read input from global memory to shared memory
- ② Repeat $\log(n)$ times: traverse from 1 to $n - 1$: double traverse for each iteration
- ③ Write results from shared memory to global memory



Managing dependencies

During all iterations, each thread can overwrite the input of another thread

- The synchronization barrier ensures that all entries have been written
- All threads secure their inputs that can be overwritten by another thread
- The synchronization barrier is necessary to ensure that all threads have secure inputs
- All threads add and write an output



An inefficient version of the kernel scan

```
--global__ void work_inefficient_scan_kernel(float *X, float *Y,  
    __shared__ float XY[SECTION_SIZE];  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < InputSize) {XY[threadIdx.x] = X[i];}  
    // the code below performs iterative scan on XY  
    for (unsigned int stride = 1; stride <= threadIdx.x; stride *= 2)  
        __syncthreads();  
        float in1 = XY[threadIdx.x - stride];  
        __syncthreads();  
        XY[threadIdx.x] += in1;  
    }  
    __syncthreads();  
    if (i < InputSize) {Y[i] = XY[threadIdx.x];}  
}
```

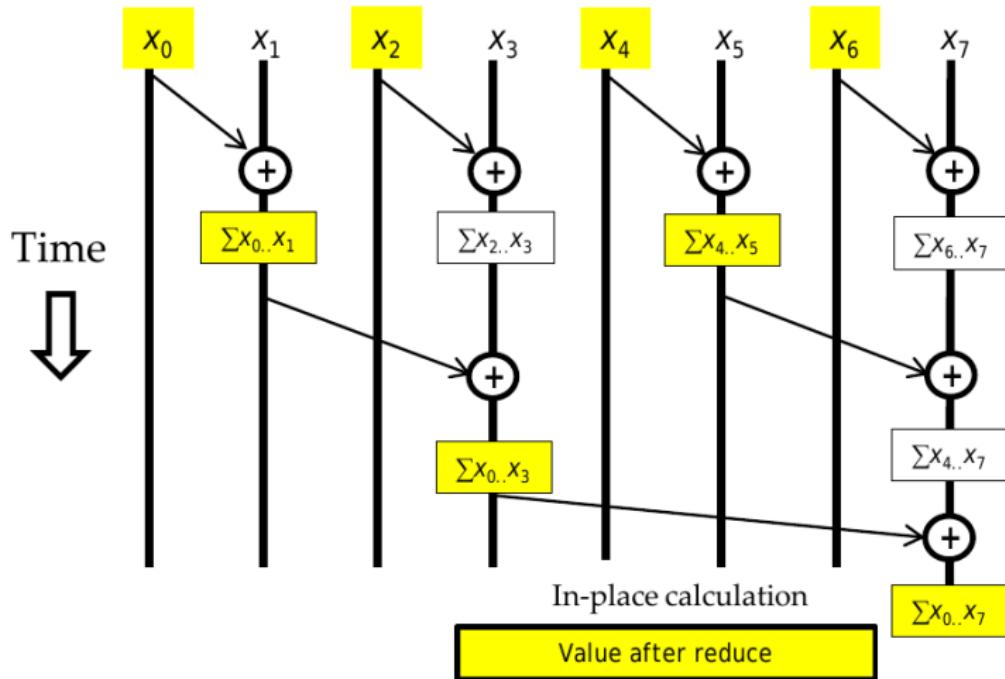
Let's consider the efficiency of this code

- The scan executes $\log(n)$ iterations in parallel
 - The iterations are respectively $(n - 1), (n - 2), (n - 4), \dots, (n - \frac{n}{2})$ addition operations
 - For a total operation addition: $n * \log(n) \rightarrow O(n * \log(n))$
- This scanning algorithm is not efficient
 - The sequential scan algorithm performs n additions
 - A $\log(n)$ factor can be expensive: 10x for 1024 elements!!!
- A parallel algorithm can be slower than its sequential equivalent when computational resources are saturated by low efficiency!

Improve efficiency

- Balanced trees
 - Build balanced binary trees for input data and browse to and from the root
 - The tree is not really the data structure used but a concept to determine what each thread does at each step
- For scan
 - Traverse the tree from the leaves to the root by building partial sums at each internal node of the tree
 - The root contains the sum of all leaves
 - Cross the tree in the reverse direction to build the results from the partial sums

Parallel scan: The reduction phase



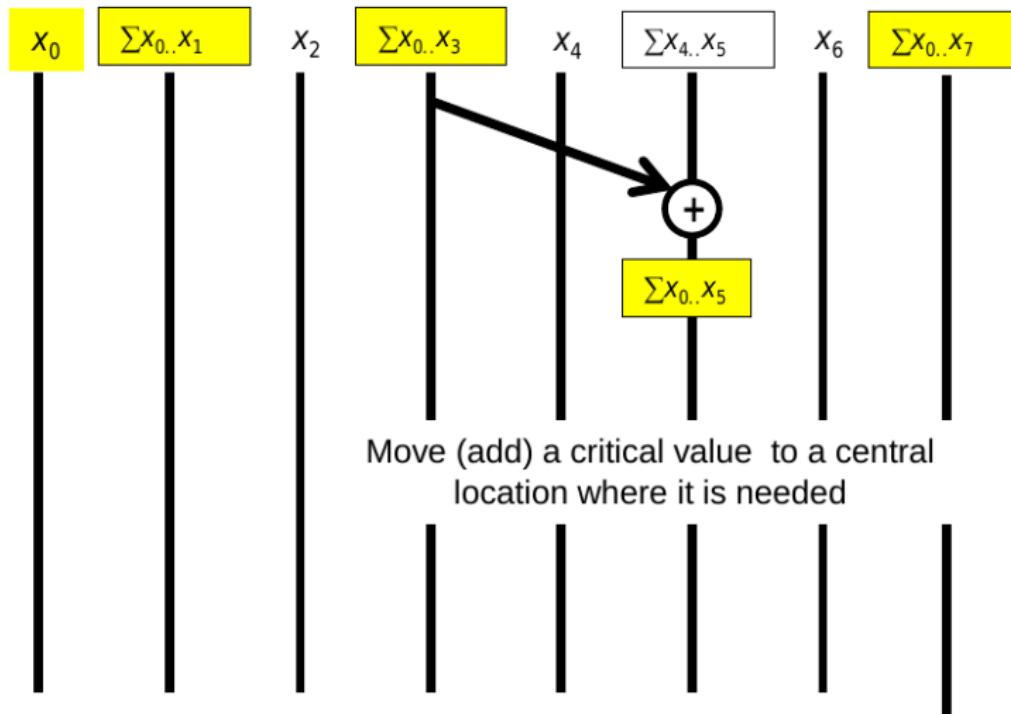
Kernel code: Reduction phase

```
// XY[2*BLOCK_SIZE] is in shared memory
for (unsigned int stride = 1; stride <= BLOCK_SIZE; stride *= 2)
{
    int index = (threadIdx.x+1)*stride*2 - 1;

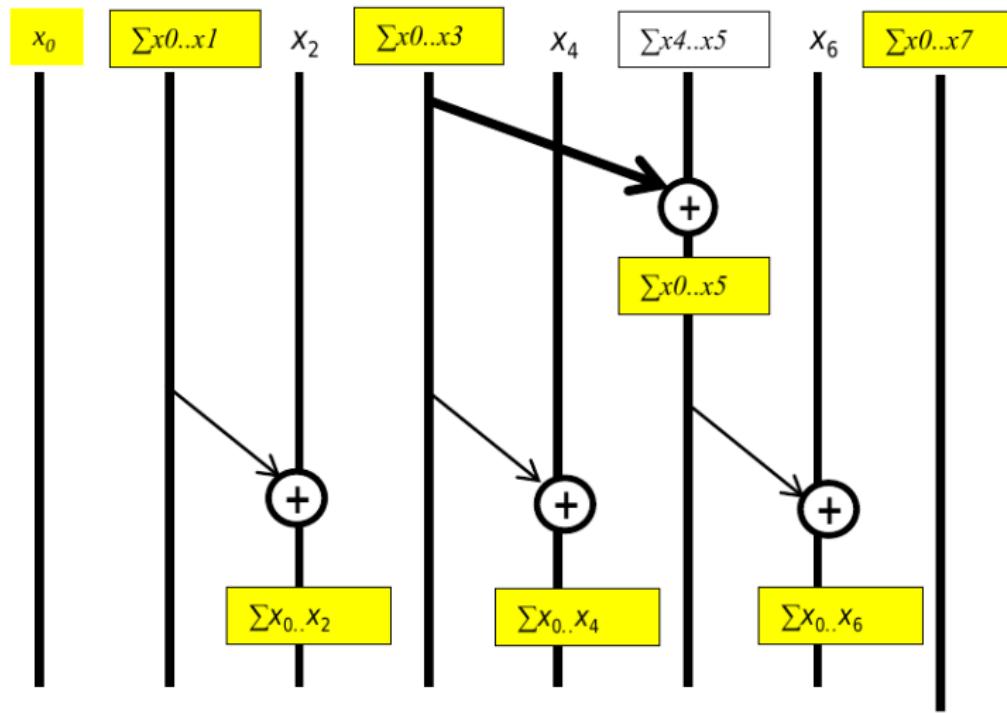
    if(index < 2*BLOCK_SIZE)
        XY[index] += XY[index-stride];

    __syncthreads();
}
```

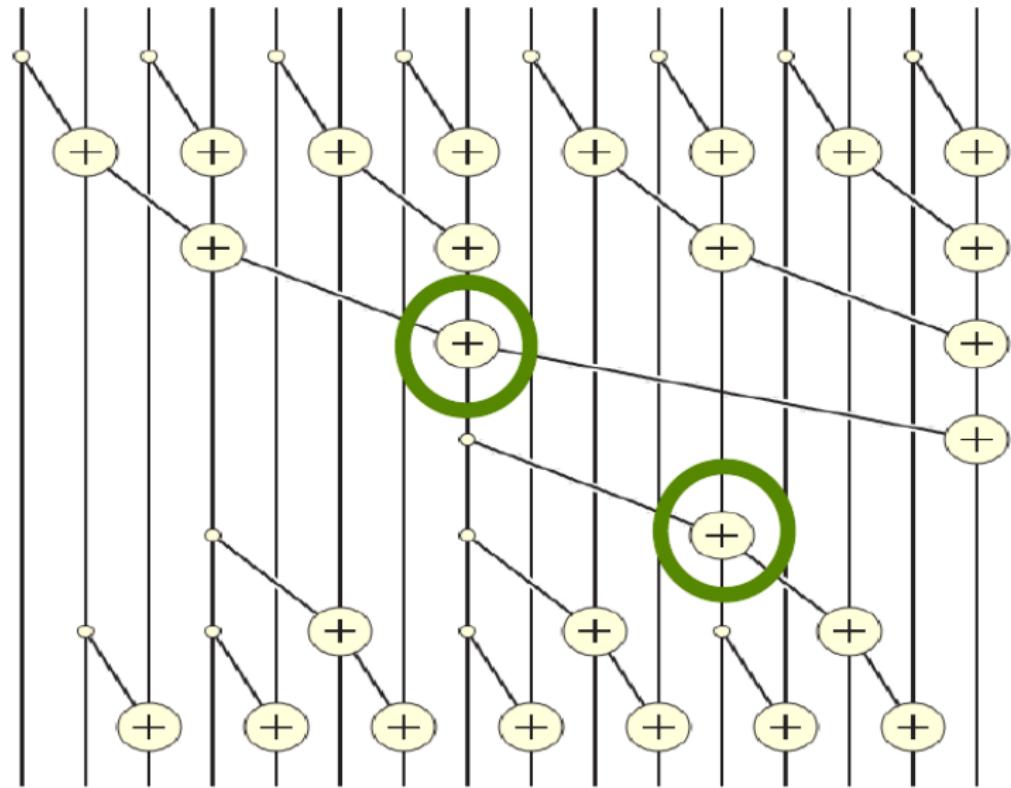
Parallel scan: after the reduction



Parallel scan: after the reduction



Overview



Kernel code: after the reduction phase

```
for (unsigned int stride = BLOCK_SIZE/2; stride > 0; stride /= 2)
    __syncthreads();
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index+stride < 2*BLOCK_SIZE) {
        XY[index + stride] += XY[index];
    }
}
__syncthreads();

if (i < InputSize) Y[i] = XY[threadIdx.x];
```

Analysis of the efficiency of the effective core

- The efficient kernel performs $\log(n)$ parallel iterations for the reduction step
 - The iterations respectively perform $\frac{n}{2}, \frac{n}{4}, \dots, 1$ addition operations
 - Overall: $(n - 1) \rightarrow O(n)$
- The efficient kernel performs $\log(n) - 1$ parallel iterations for the post-reduction step
 - The iterations respectively perform $2 - 1, 4 - 1, \dots, \frac{n}{2-1}$ addition operations
 - Overall: $(n - 2) - (\log(n) - 1) \rightarrow O(n)$
- The 2 phases perform up to but not more than $2 * (n - 1)$ addition operations
- The total number of additions is not more than 2 times what is done by the efficient sequential algorithm
 - The benefit of parallel execution can largely overcome this doubling of work when there are sufficient computing resources

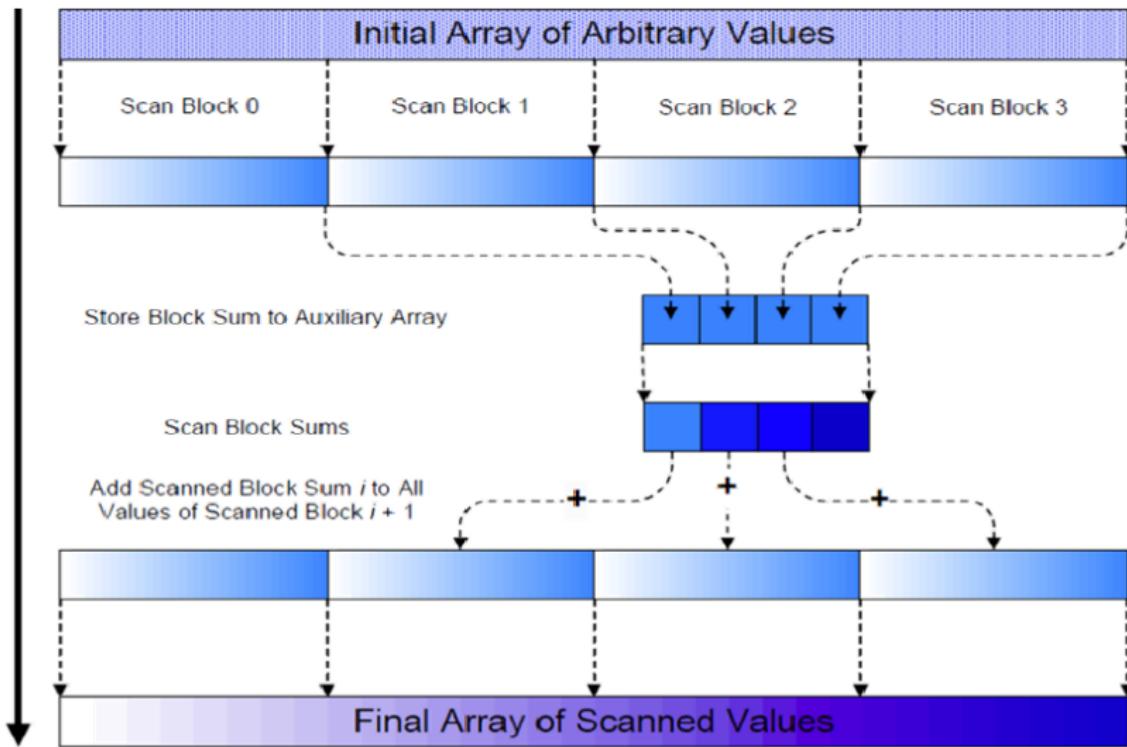
Some trade-offs

- The efficient scan kernel is normally more desirable
 - Better energy efficiency
 - Less resources are needed for its execution
- Nevertheless, the inefficient version of the kernel could be better in absolute terms of performance thanks to its single phase
 - In this case you need sufficient resources

Processing of large input tables

- Based on the efficient kernel scan
- Have each section of $2 * blockDim.x$ elements assigned to a block
 - Perform a parallel scan on each section
- Each block writes the sum of its section to a sum array $Sum[]$ using its index $blockIdx.x$
- Run the kernel scan on the $Sum[]$ array
- Add the values of the array $Sum[]$ to all elements of the corresponding sections
- The adaptation of the inefficient core is done in a similar way

Diagram of the complete execution of scan



Exclusive scan definition

Definition : The operation exclusive scan takes an associative binary operator \oplus (plus circle) and an array of n elements.

$[x_0, x_1, \dots x_n]$

and returns an array $[0, x_0, x_0 \oplus x_1, \dots x_0 \oplus x_1 \oplus \dots \oplus x_{n-2}]$

Example : if *oplus* is the addition, then the operation scan on the array should return

$[3, 1, 7, 0, 4, 1, 6, 3],$

$[0, 3, 4, 11, 11, 15, 16, 22].$

Why use an exclusive scan

- To find the buffer start address
- The inclusive and exclusive scans can easily be deduced from each other, it is a matter of convenience

	[3, 1, 7, 0, 4, 1, 6, 3]
Exclusif	[0, 3, 4, 11, 11, 15, 16, 22]
Inclusif	[3, 4, 11, 11, 15, 16, 22, 25]

A simple version of the kernel scan exclusive

- Adaptation of the inefficient version of the kernel scan inclusive
- Block 0:
 - Thread 0 puts a 0 in $XY[0]$.
 - All other threads put the value of $X[threadIdx.x - 1]$ into $XY[threadIdx.x]$
- All other blocks
 - All threads put the value of $X[blockIdx.x * blockDim.x + threadIdx.x - 1]$ into $XY[threadIdx.x]$