Introduction to GPU programming with CUDA

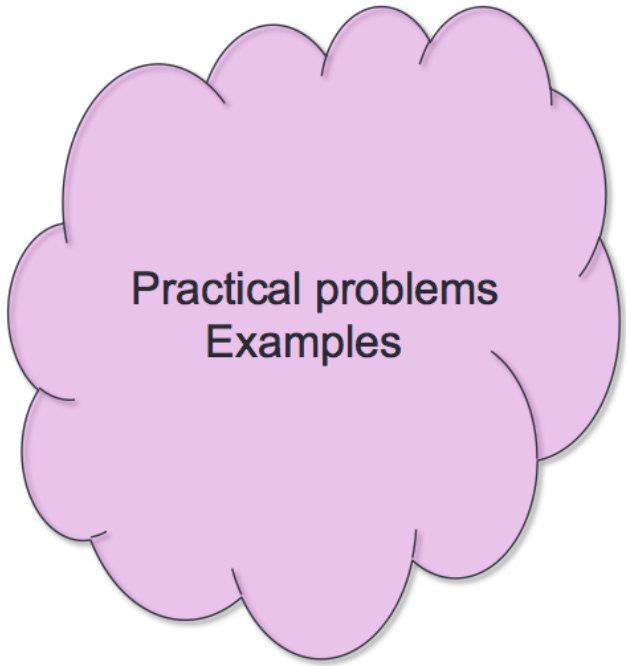
Dr. Juan C Zuniga
University of Saskatchewan, WestGrid

UBC Summer School, Vancouver. June 12th, 2018

Outline

1. Overview of GPU computing
   a. what is a GPU?
   b. GPU architecture
   c. GPU programming approaches
2. CUDA programming model
   a. data parallelism
   b. thread hierarchy
   c. memory model
3. Synchronicity in CUDA
   a. task timeline (kernels, transfers, CPU computations)
   b. concurrency
   c. streams and events (synchronization)
4. Profiling and optimization of CUDA kernels
   a. Instrumenting code, and the NVIDIA profiler
   b. occupancy (memory latency, stalls)
   c. branching, iterations, loops

Outline

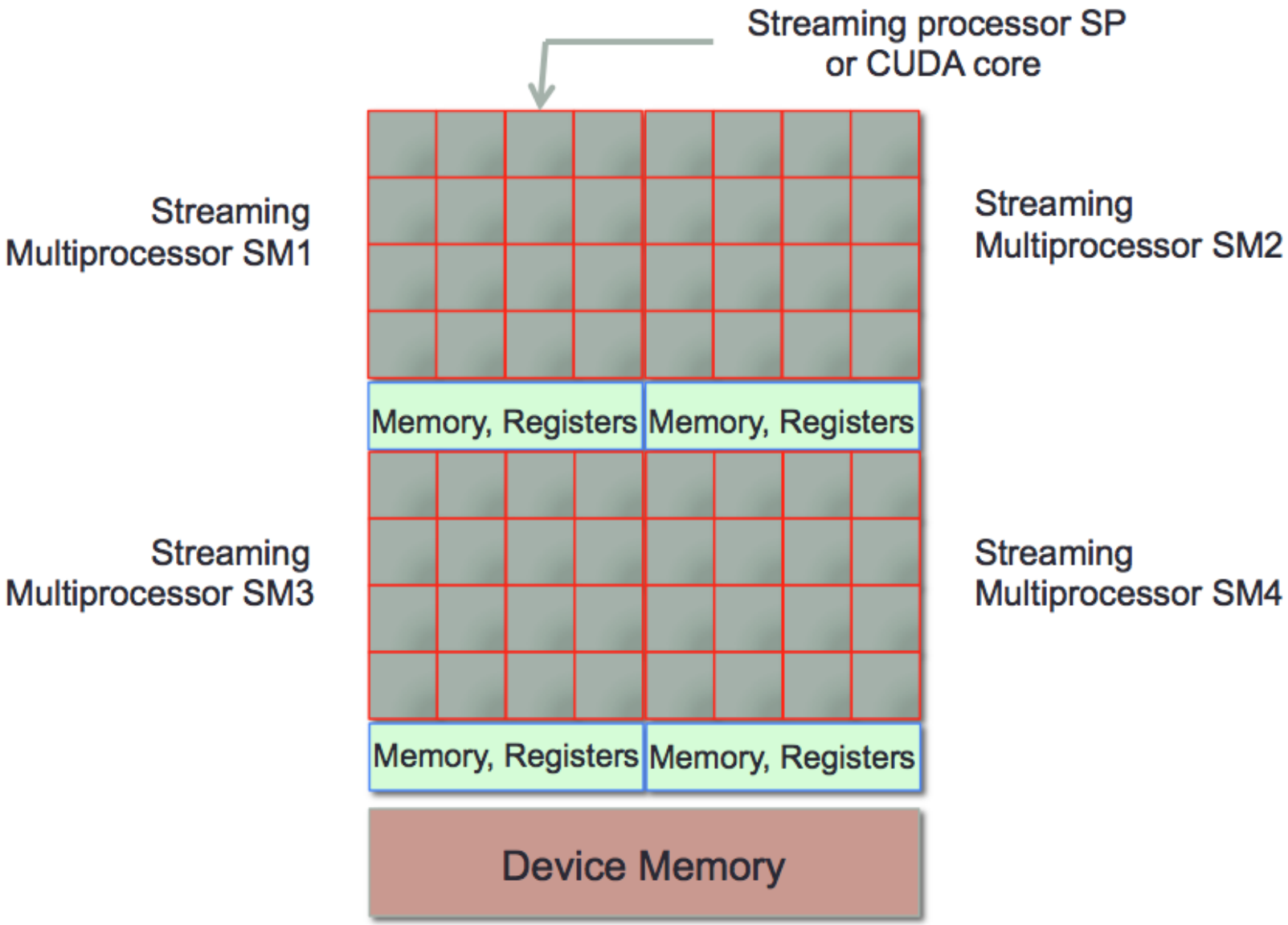


1. Analyze problem
2. Define Algorithm
3. Serial Implementation
4. GPU implementation
5. Profiling/Optimization ?

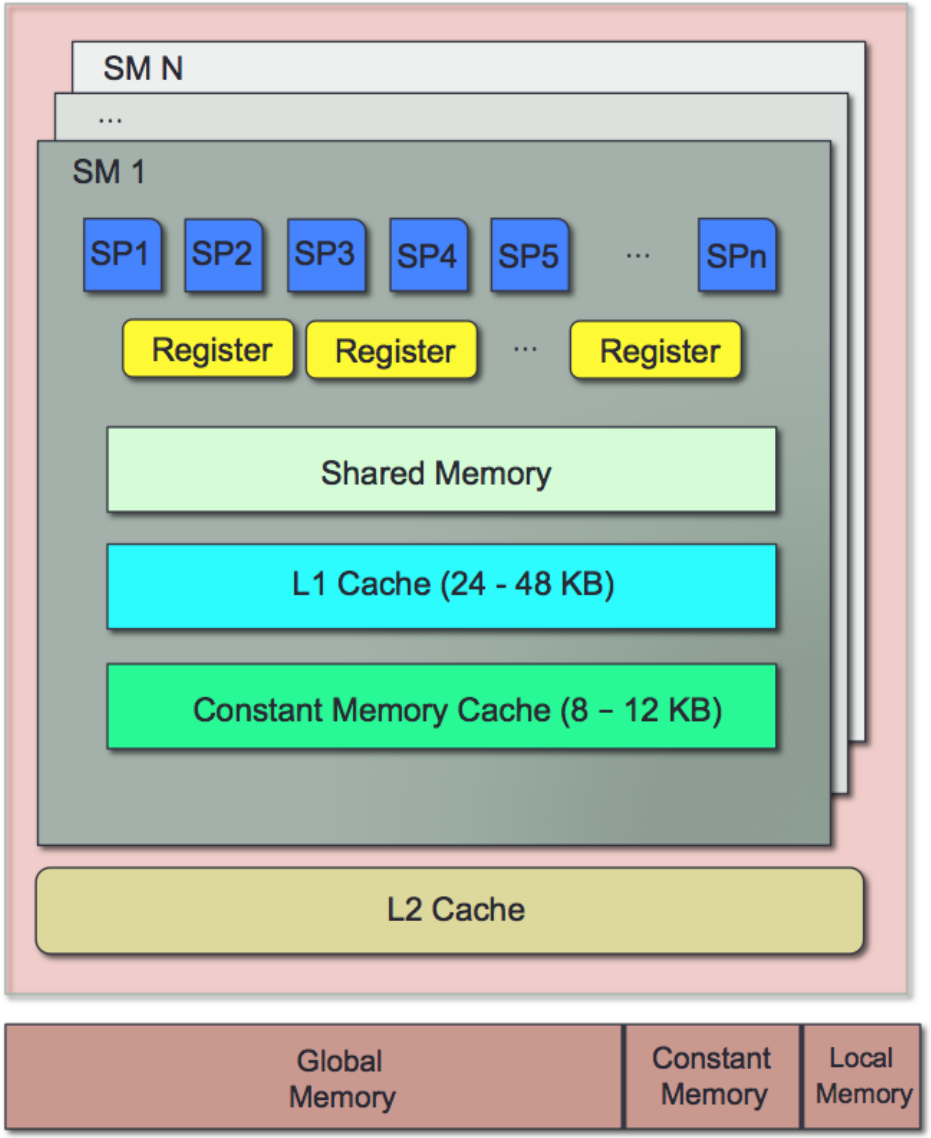
GPU Architecture

GPU Programming Model / Execution Model

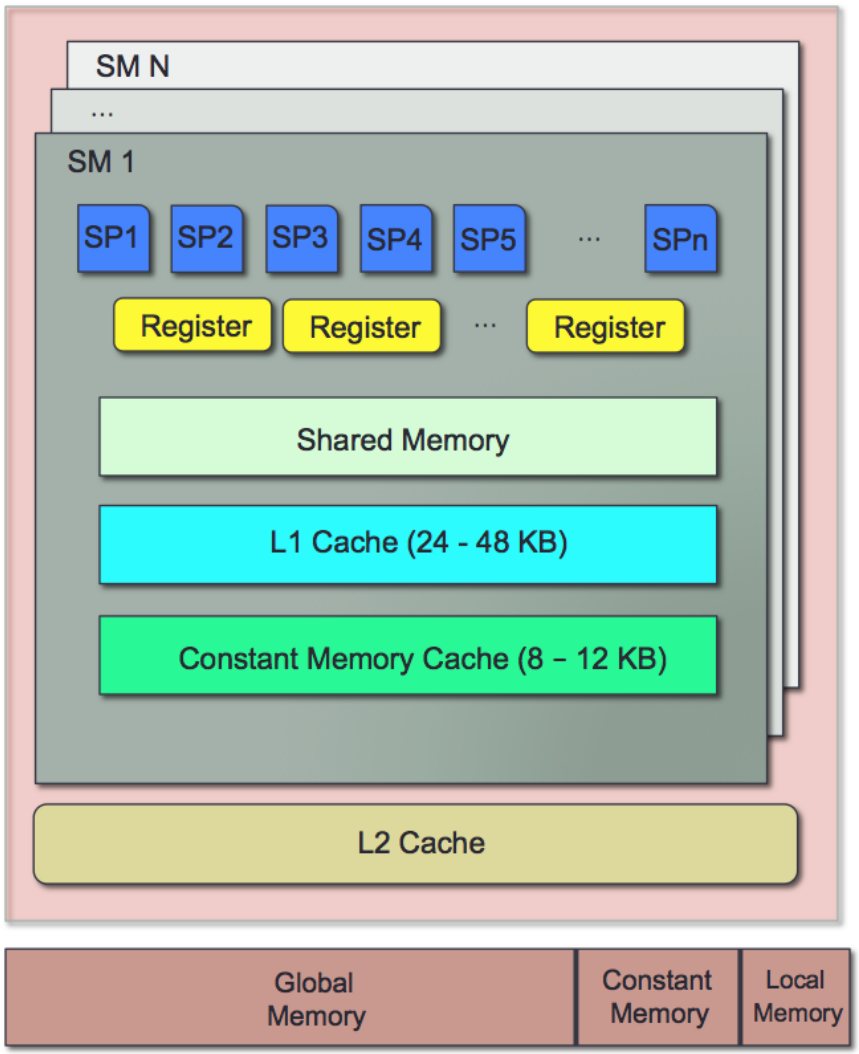
GPU Architecture



GPU Architecture



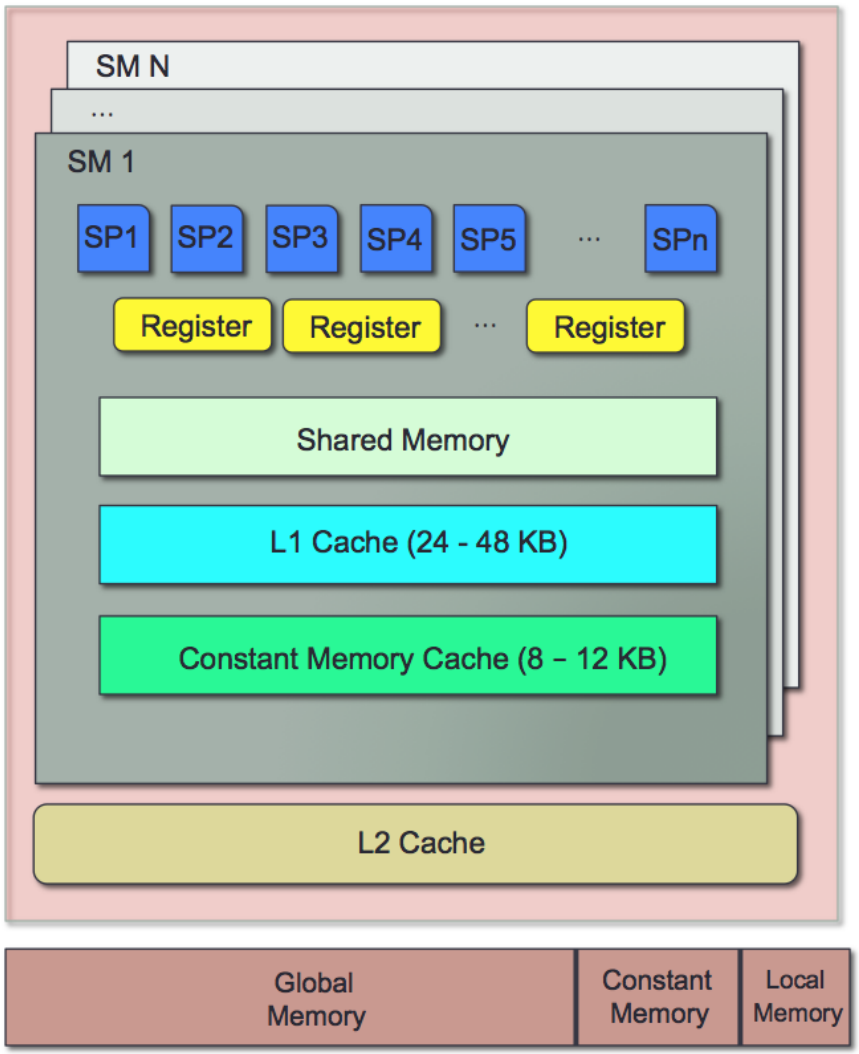
GPU Memory structure



Shared mem and L1 cache:

- The fastest memory you can have
- Shared is managed by the programer, L1 is like CPU cache
- Shared is visible to all compute threads running on SM
- L1 could cache global and/or local memory
- No coherency for L1 cache!!!

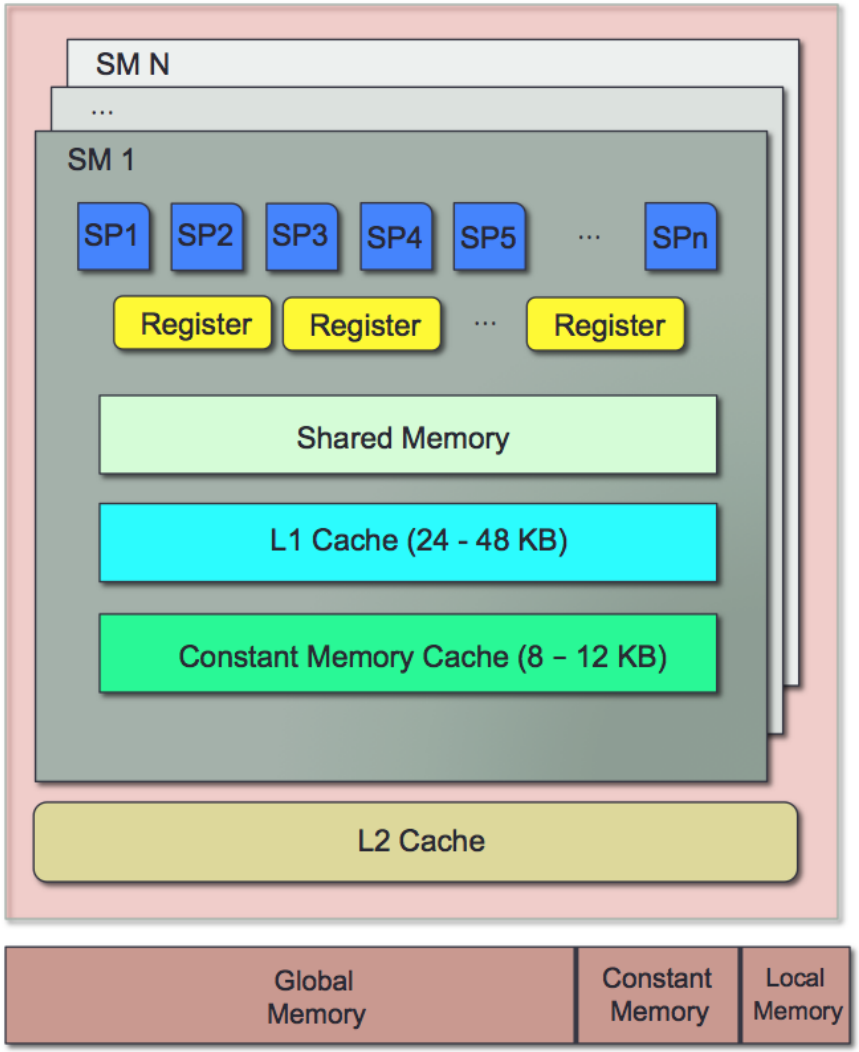
GPU Memory structure



L2 cache:

- Visible to all SMs
- There is coherency for L2
- Global coherency is not guaranteed since L2 caches global and local memory
- Two threads should not modify the same global memory element

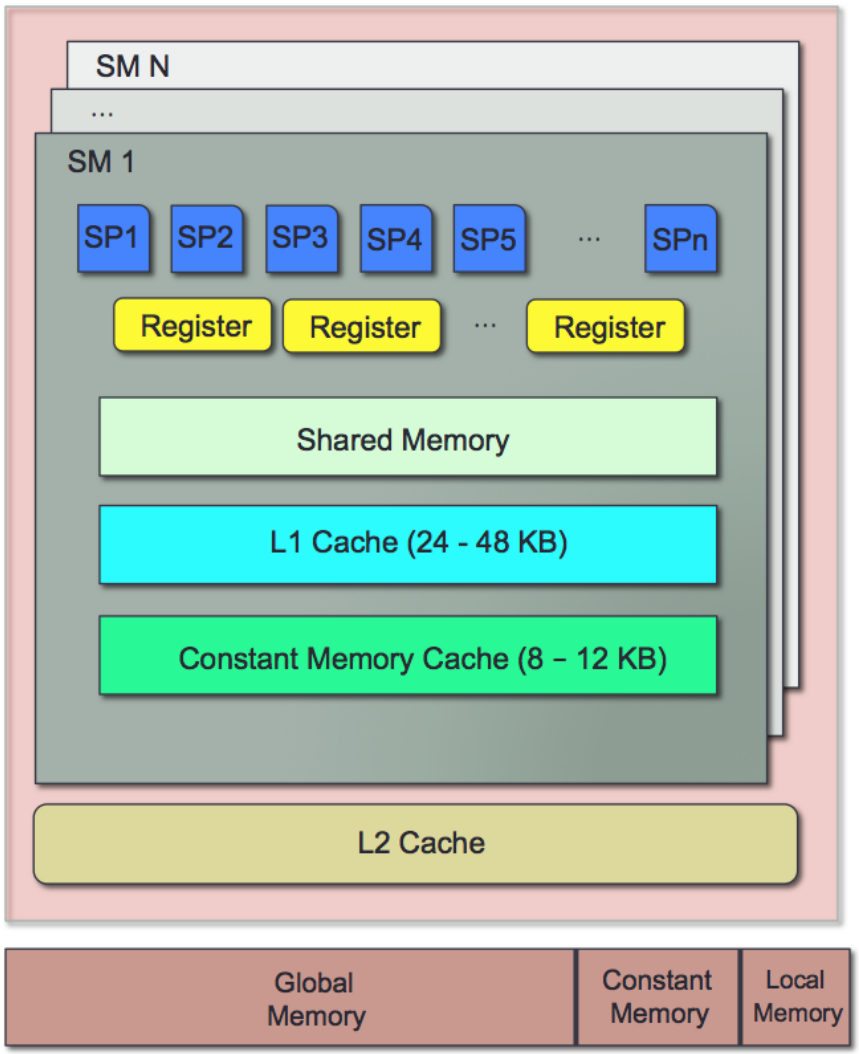
GPU Memory structure



Device memory:

- Global + Local + Constant
- Is off-chip. Like the RAM for CPU (also called VRAM in graphics/video cards)
- Order of magnitude slower than in-chip memory
- Two orders of magnitude more latency

GPU Memory structure / Registers



Registers:

- 32 or 64 bits per register
- Lots of them on each SM
- Assigned at compile time
- Superfast (no latency)
- Local memory used as register spill

GPU Memory structure

What should I care for ?

- For starters: Global Memory and Shared Memory
- Constant memory
- Register count (and local memory), Shared memory - Occupancy
- Your data structure to ensure well use of cache (texture memory)

NVIDIA Pascal P100 specs

We can use CUDA API...
cudaGetDeviceProperties()

Programming Models

Programming Model	Execution Model
• Sequential (SISD)	• Pipeline, Out of order
• Data Parallelism (SIMD)	• SIMD machines
• Task Parallelism (MIMD)	• Multi threads/cores

The programming model could respond to 2 questions:

1. How many streams of instructions you have
2. How many streams of data you have

The execution model = the way the architecture is organized to implement the programming model

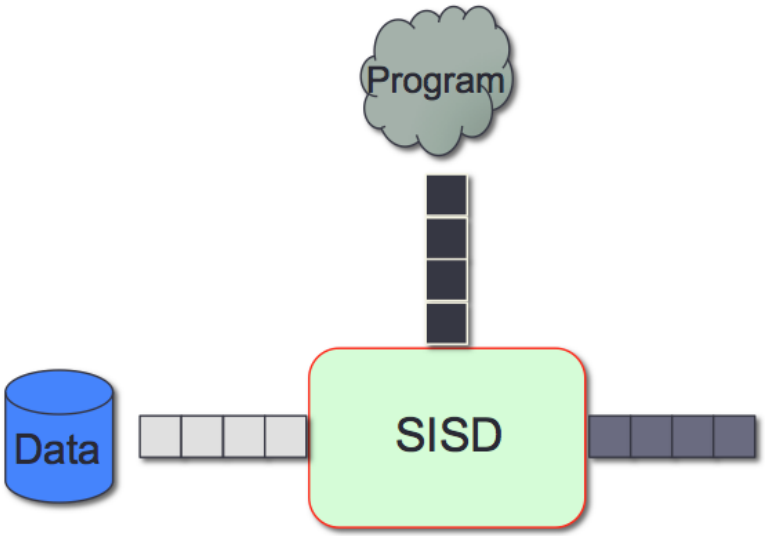
Programming Models / SISD / SIMD

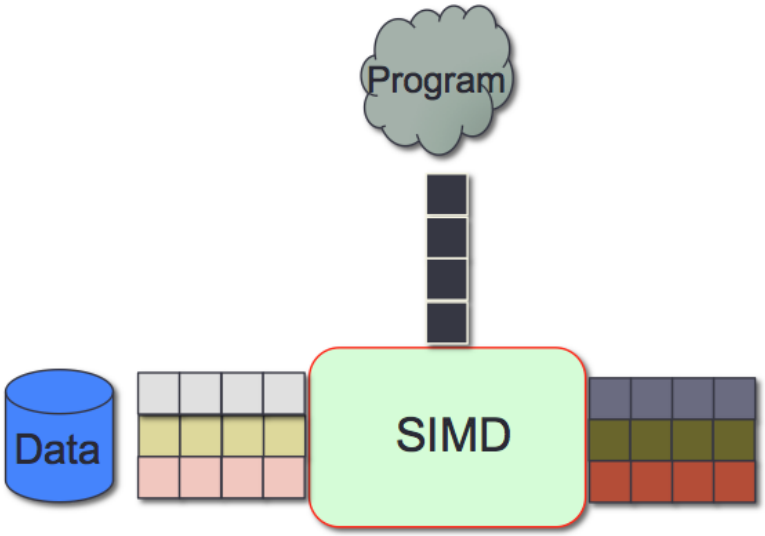
Programming Model

- Sequential (SISD)
- Data Parallelism (SIMD)
- Task Parallelism (MIMD)

Execution Model

- Pipeline, Out of order
- SIMD machines
- Multi threads/cores





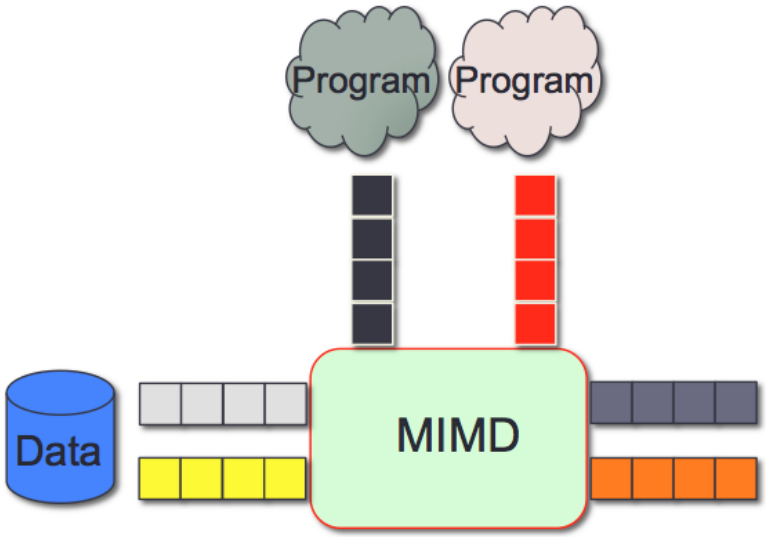
Programming Models / MIMD / SPMD

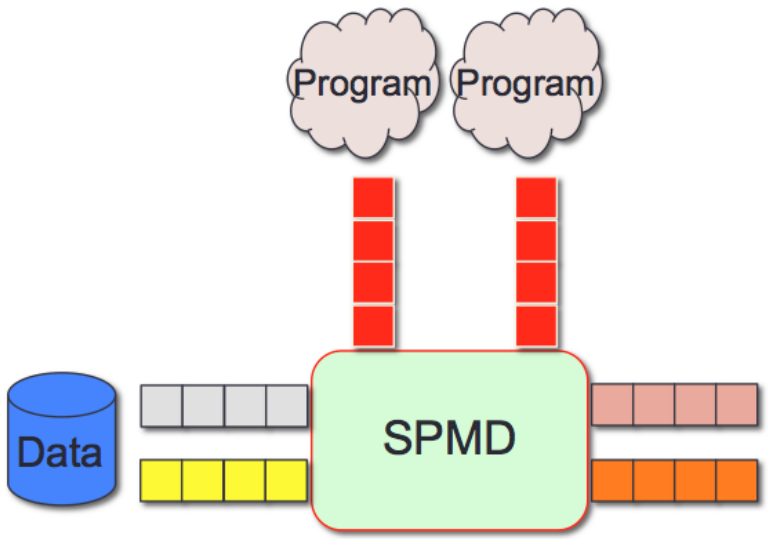
Programming Model

- Sequential (SISD)
- Data Parallelism (SIMD)
- Task Parallelism (MIMD)

Execution Model

- Pipeline, Out of order
- SIMD machines
- Multi threads/cores





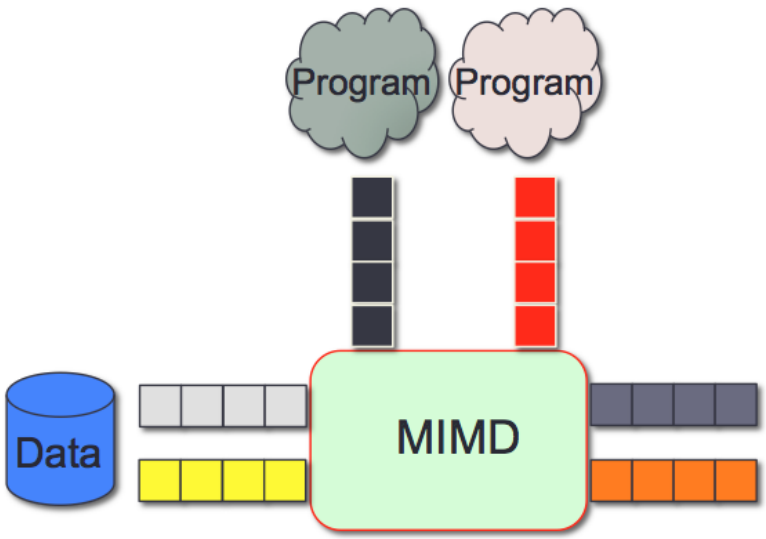
Programming Models / MIMD / SPMD

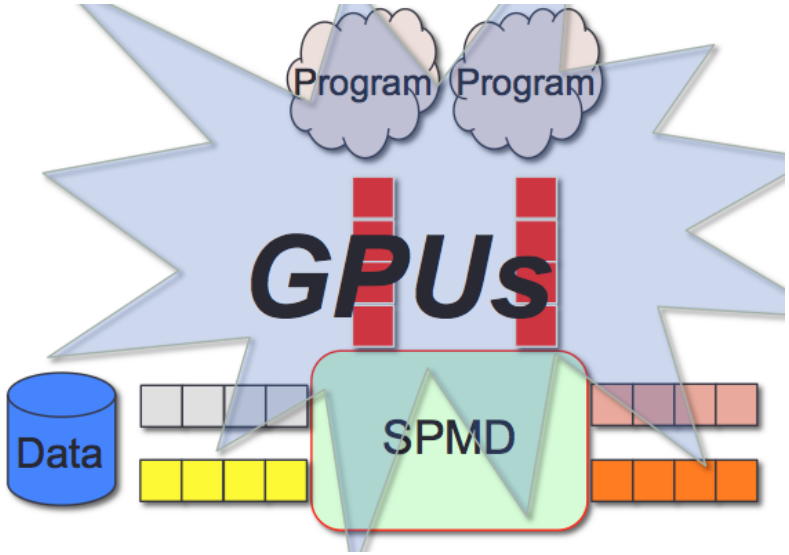
Programming Model

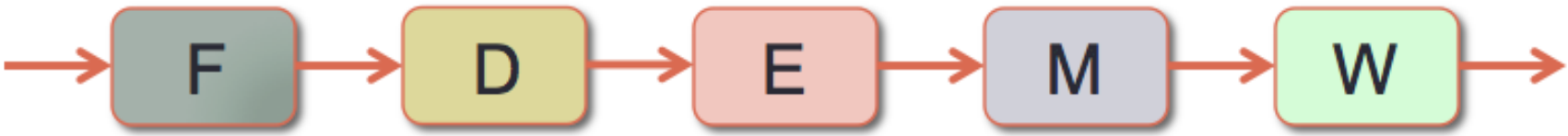
- Sequential (SISD)
- Data Parallelism (SIMD)
- Task Parallelism (MIMD)

Execution Model

- Pipeline, Out of order
- SIMD machines
- Multi threads/cores



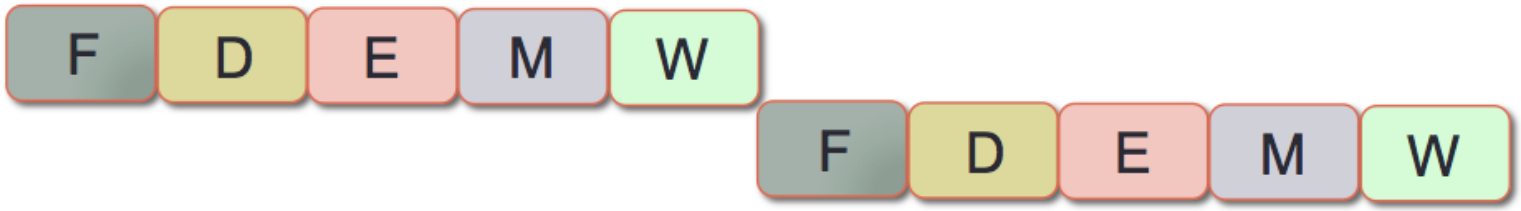




Instruction Cycle:

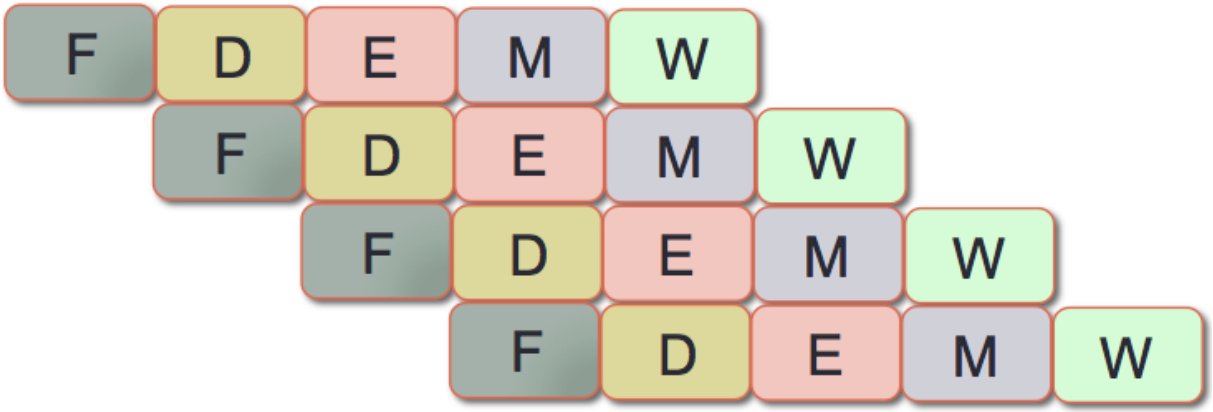
1. Fetch instruction
2. Decode instruction
3. Execute with registers
4. Access memory
5. Write back to registers

Latency: amount of time that an instruction takes to complete
Throughput: number of instructions completed per unit of time



Latency = 5 cycles, Throughput = 0.2 IPC

Functional units for every stage of the instruction cycle allows Pipelining (Instruction level Parallelism)



Latency = 5 cycles, Throughput = 1 IPC !!!

Execution Models / Pipelining

Pipeline Hazards...

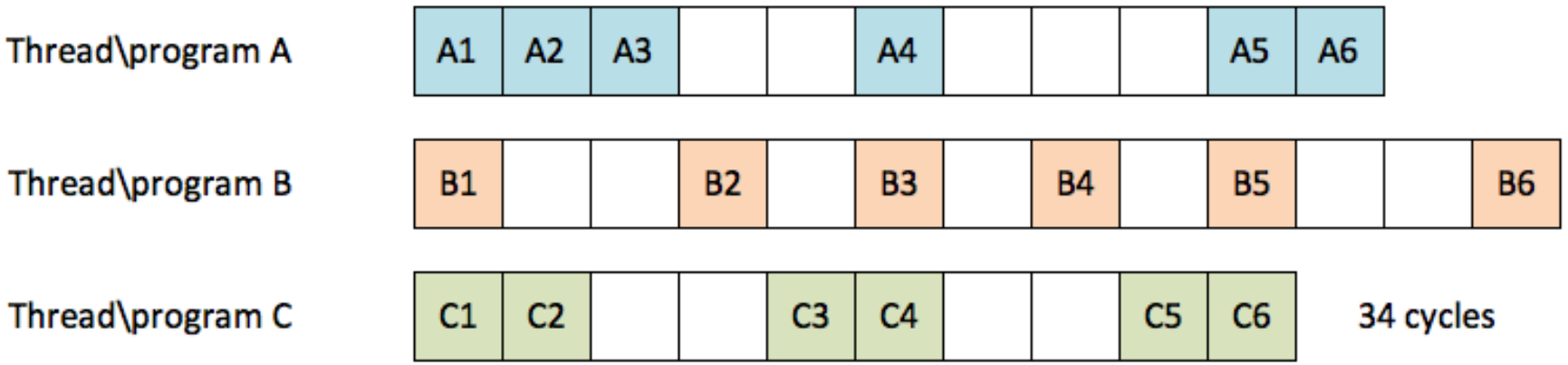
- Control Hazards
- Data Hazards
- Structural Hazards

All sort of creative solutions:

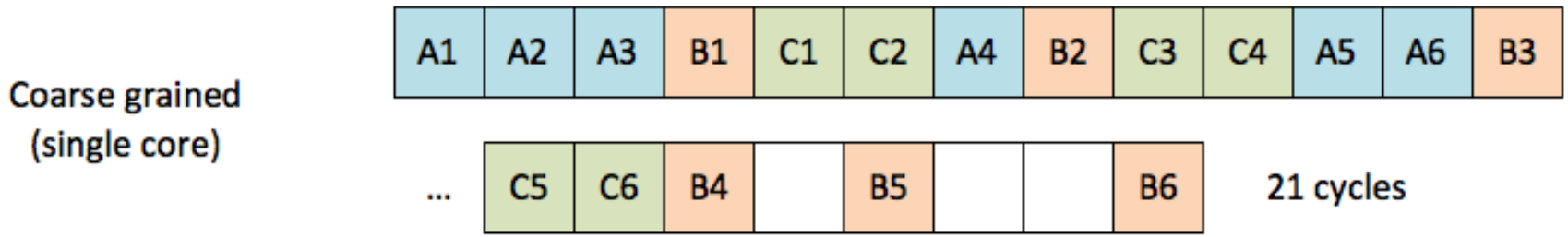
- Multiple functional units (specialized)
- Cache for data and cache for instructions
- Our of order execution
- Forwarding
- Branch prediction

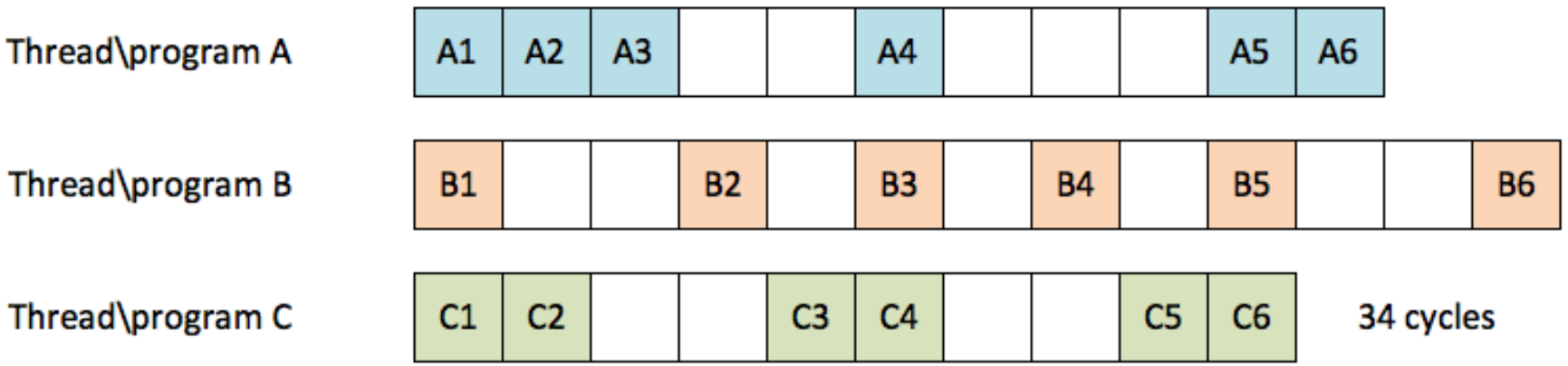
Still very hard to keep the pipeline without stalls !!! The alternative: to introduce another level of parallelism,

Thread level parallelism

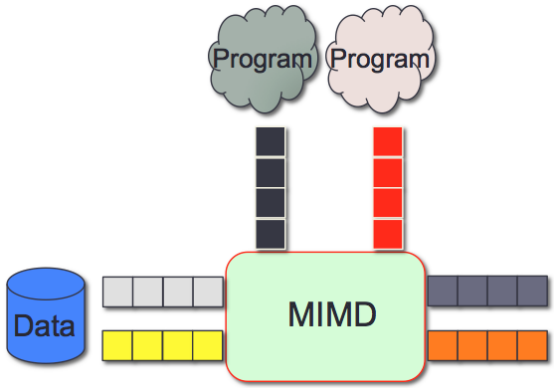


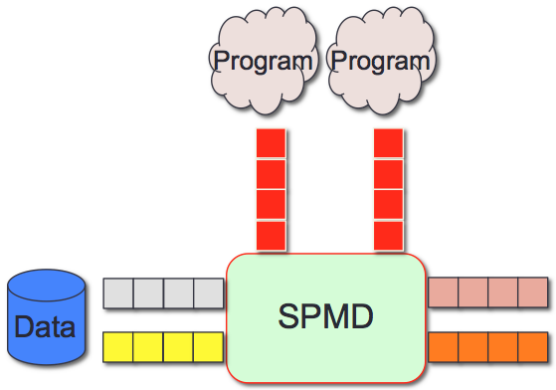
Multithreading can be: coarse-grained, fine-grained, or simultaneous

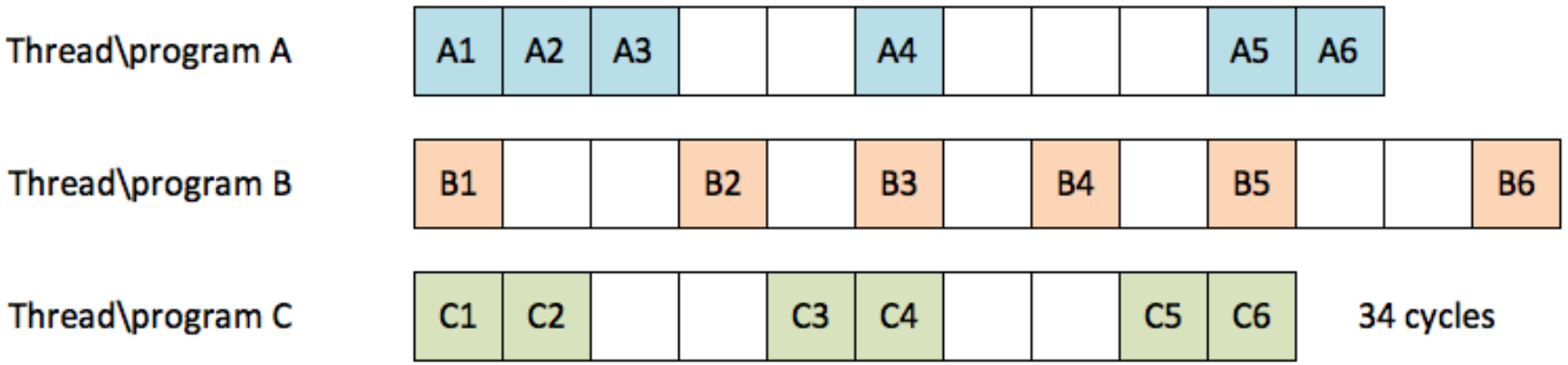




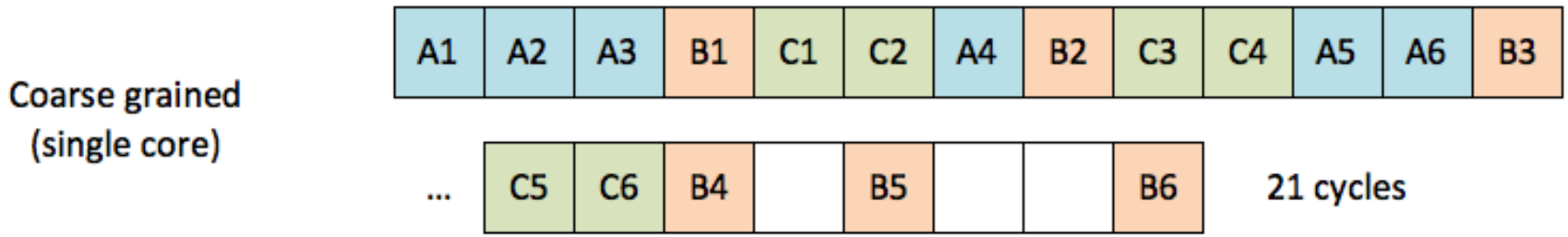
Multithreading can be: coarse-grained, fine-grained, or simultaneous

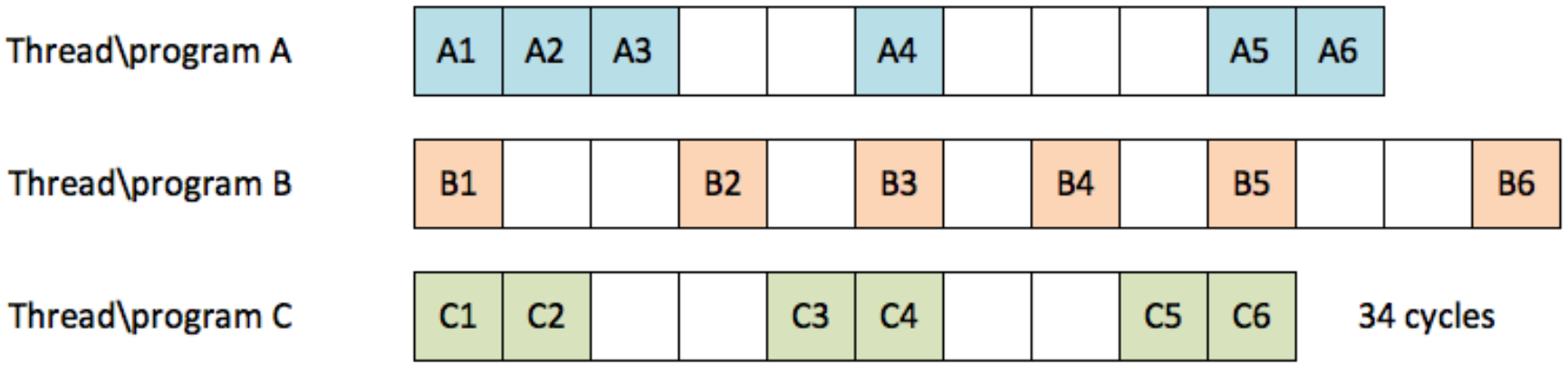




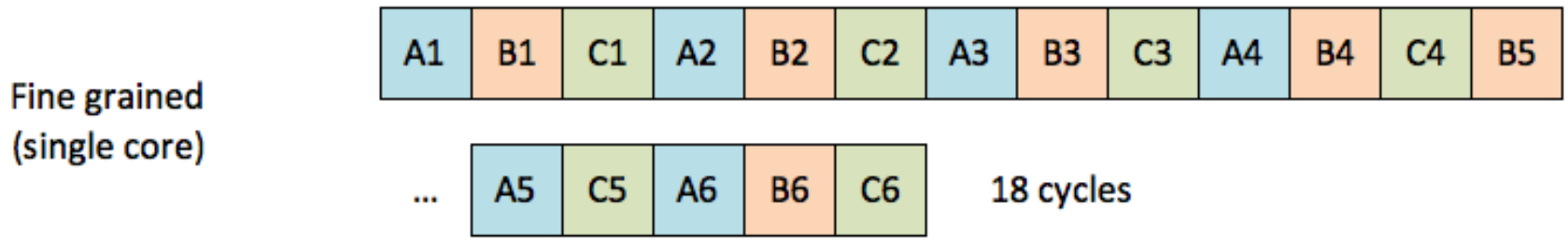


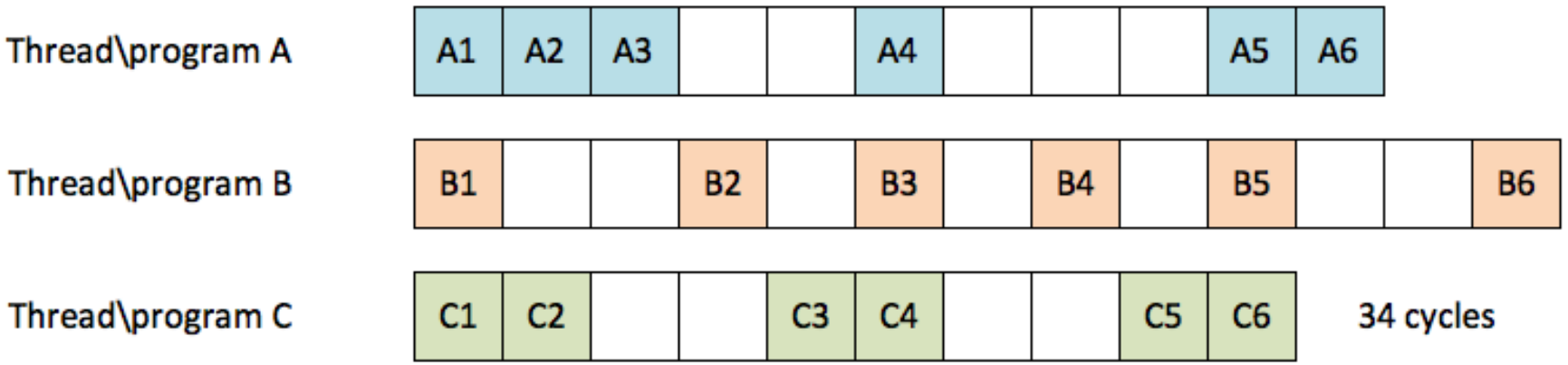
Multithreading can be: coarse-grained, fine-grained, or simultaneous



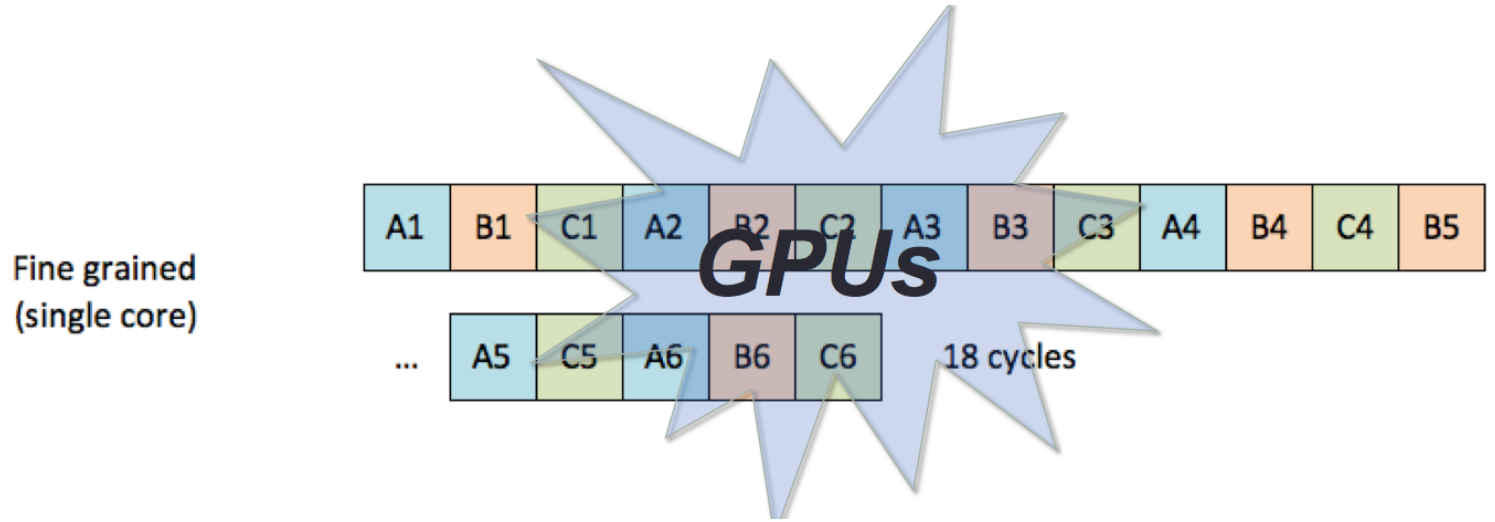


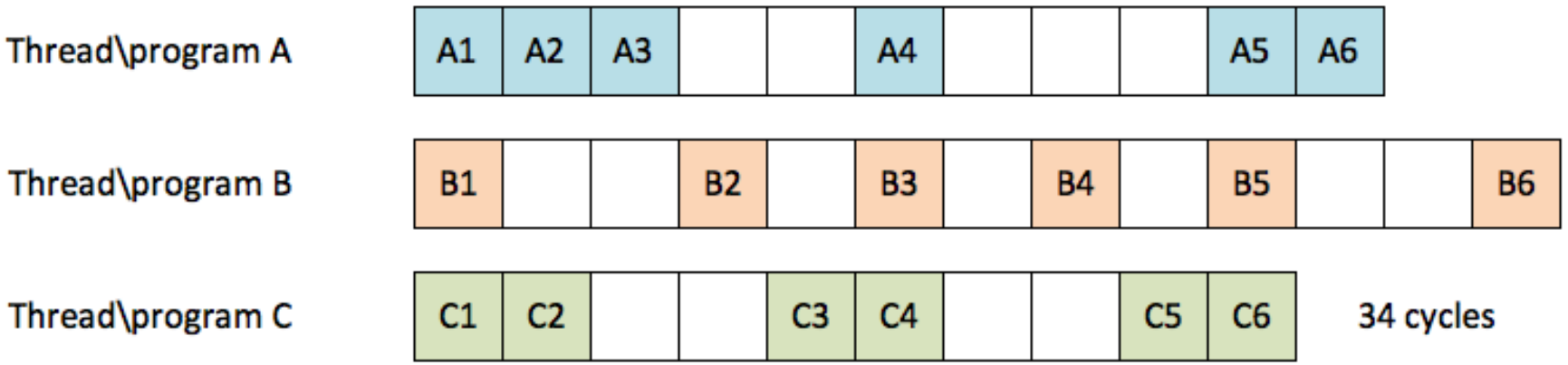
Multithreading can be: coarse-grained, fine-grained, or simultaneous



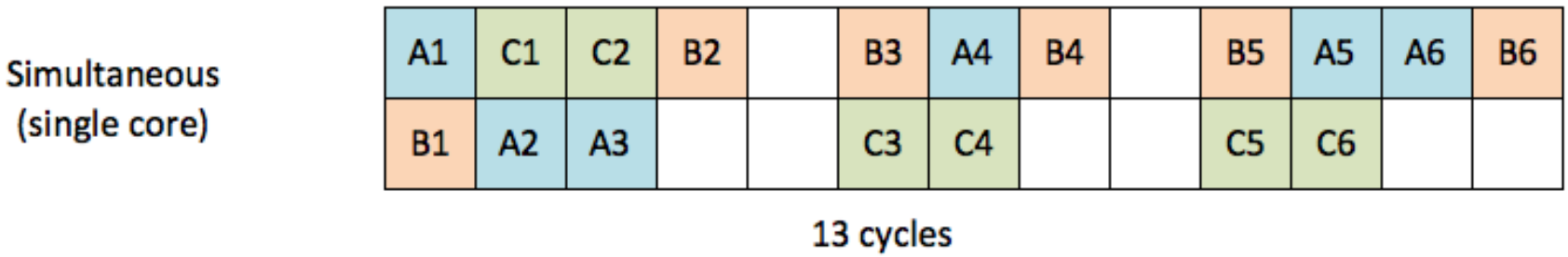


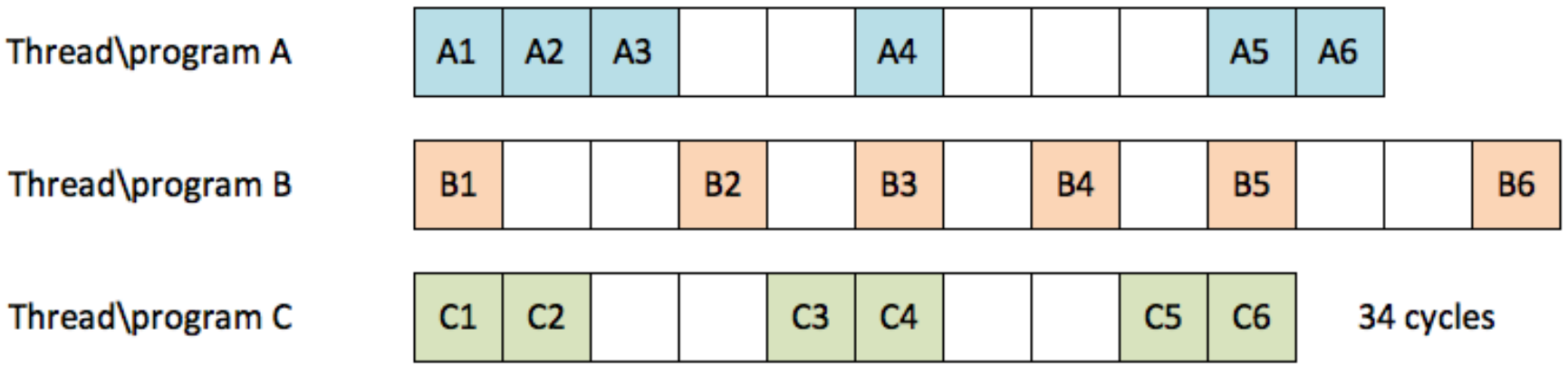
Multithreading can be: coarse-grained, fine-grained, or simultaneous



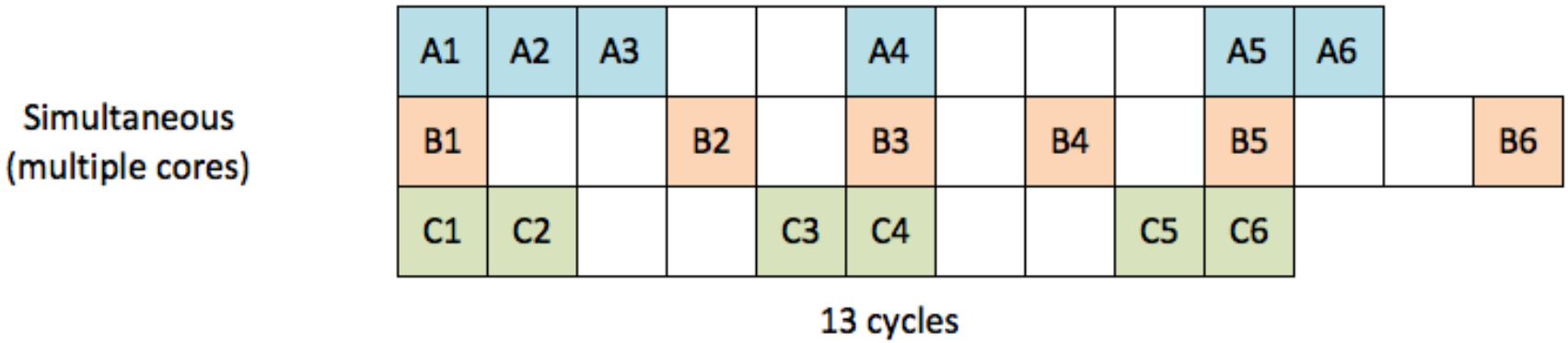


Multithreading can be: coarse-grained, fine-grained, or simultaneous





Multithreading can be: coarse-grained, fine-grained, or simultaneous



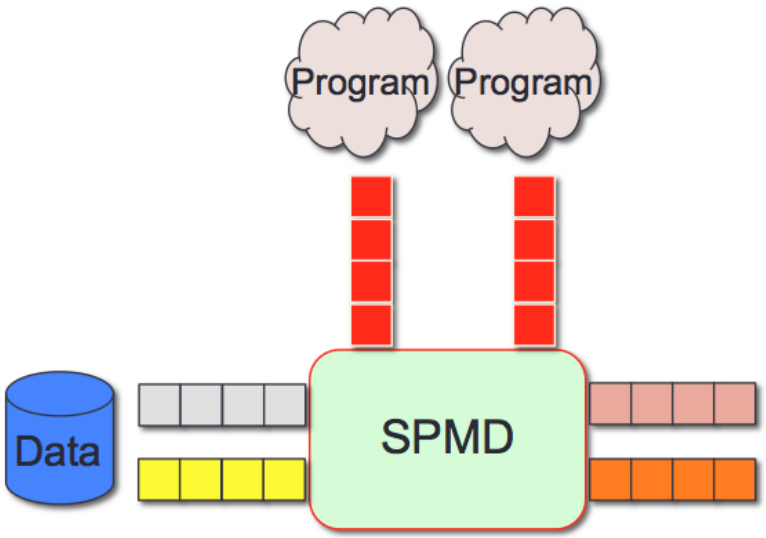
Back to GPUs...

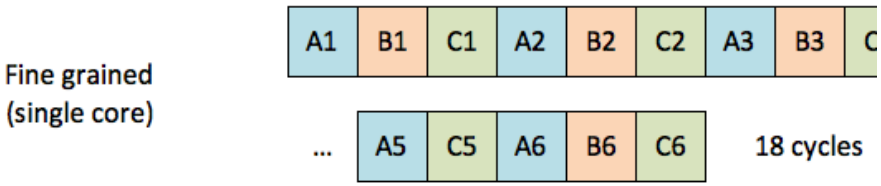
Programming Model

- Sequential (SISD)
- Data Parallelism (SIMD)
- Task Parallelism (MIMD)

Execution Model

- Pipeline, Out of order
- SIMD machines
- Multi threads/cores





$$A = [a1, a2, a3, a4]$$

$$B = [b1, b2, b3, b4]$$

$$C = A + B$$

$$A = [a1, a2, a3, a4]$$

$$B = [b1, b2, b3, b4]$$

$$C = A + B$$

```
data_type A[4]={...}
data_type B[4]={...}
data_type C[4]
For k = 1..4:
C[k] = A[k] + B[k]
```

$$A = [a1, a2, a3, a4]$$

$$B = [b1, b2, b3, b4]$$

$$C = A + B$$

```
data_type A[4]={...}
data_type B[4]={...}
data_type C[4]
For k = 1..4:
C[k] = A[k] + B[k]
```

```
Ld R1,mem(Ak)
Ld R2,mem(Bk)
Add R0,R1,R2
St R0,mem(Ck)
```

repeat four times...

$$A = [a1, a2, a3, a4]$$

$$B = [b1, b2, b3, b4]$$

$$C = A + B$$

```
data_type A[4]={...}
data_type B[4]={...}
data_type C[4]
For k = 1..4:
C[k] = A[k] + B[k]
```

```
Ld R1,mem(Ak)
Ld R2,mem(Bk)
Add R0,R1,R2
St R0,mem(Ck)
```

repeat four times...

```
VLd VR1,mem(A)
VLd VR2,mem(B)
VAdd VR0,VR1,VR2
VSt VR0,mem(C)
```

Execution Models / SIMD Machines

$$A = [a1, a2, a3, a4]$$

$$B = [b1, b2, b3, b4]$$

$$C = A + B$$

```
data_type A[4]={...}
data_type B[4]={...}
data_type C[4]
For k = 1..4:
C[k] = A[k] + B[k]
```

```
Ld R1,mem(Ak)
Ld R2,mem(Bk)
Add R0,R1,R2
St R0,mem(Ck)
repeat four times...
```

```
VLd VR1,mem(A)
VLd VR2,mem(B)
VAdd VR0,VR1,VR2
VSt VR0,mem(C)
```

The compilers could generate vector instructions (AVX, AVX2)

Automatic vectorization, optimization -O2 -O3

Execution Models / SIMD Machines

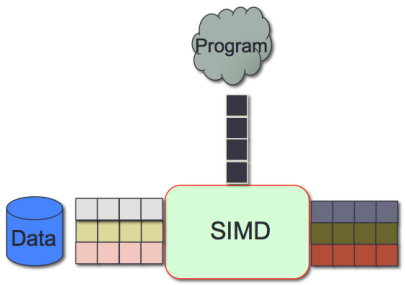
$$A = [a1, a2, a3, a4]$$
$$B = [b1, b2, b3, b4]$$
$$C = A + B$$

```
data_type A[4]={...}
data_type B[4]={...}
data_type C[4]
For k = 1..4:
C[k] = A[k] + B[k]
```

```
Ld R1,mem(Ak)
Ld R2,mem(Bk)
Add R0,R1,R2
St R0,mem(Ck)
```
repeat four times...

```
VLd VR1,mem(A)
VLd VR2,mem(B)
VAdd VR0,VR1,VR2
VSt VR0,mem(C)
```

Or you could program thinking on vectors and/or data parallelism...

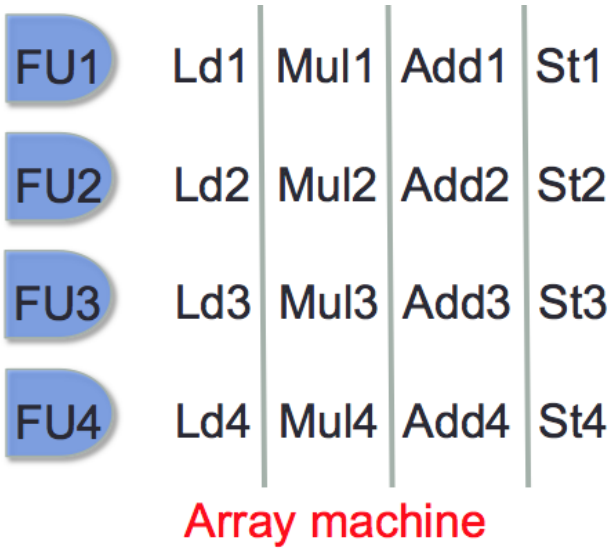


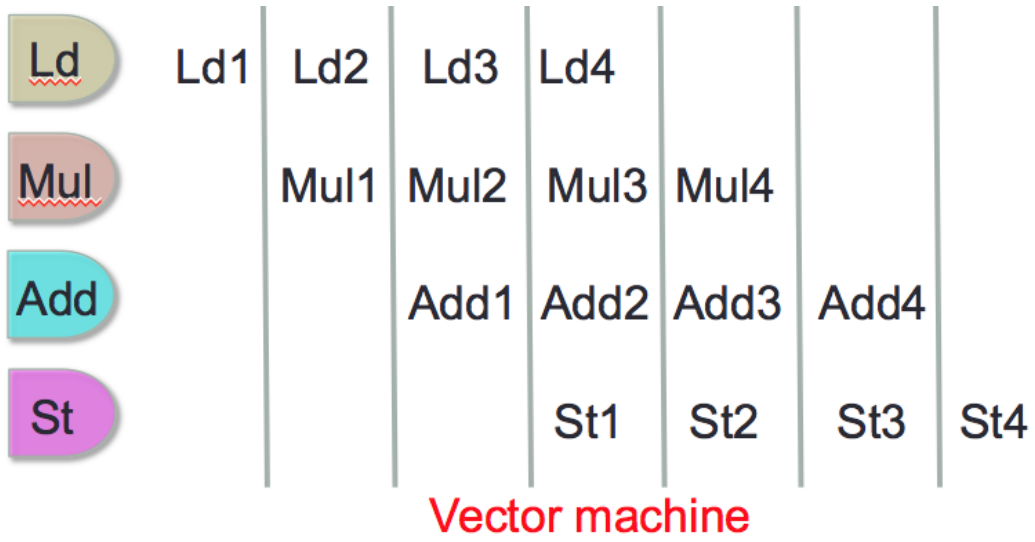
Execution Models / SIMD Machines

$$A = [a1, a2, a3, a4]$$

$$A = 2A + 3$$

```
VLd VR0,mem(A)
VMul VR0,VR0,2
VAdd VR0,VR0,3
VSt VR0,mem(A)
```



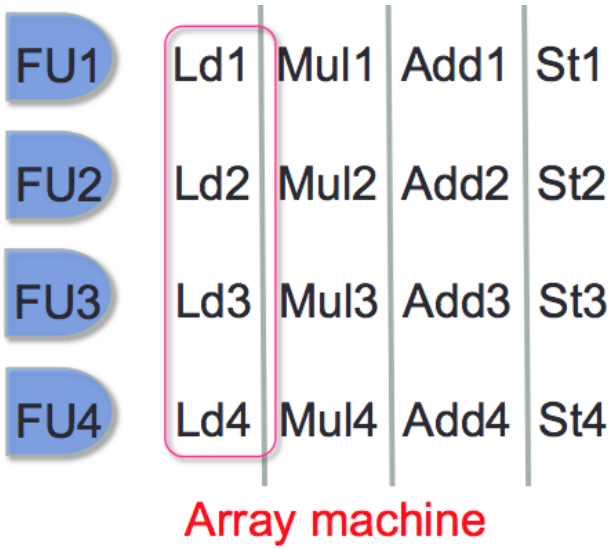


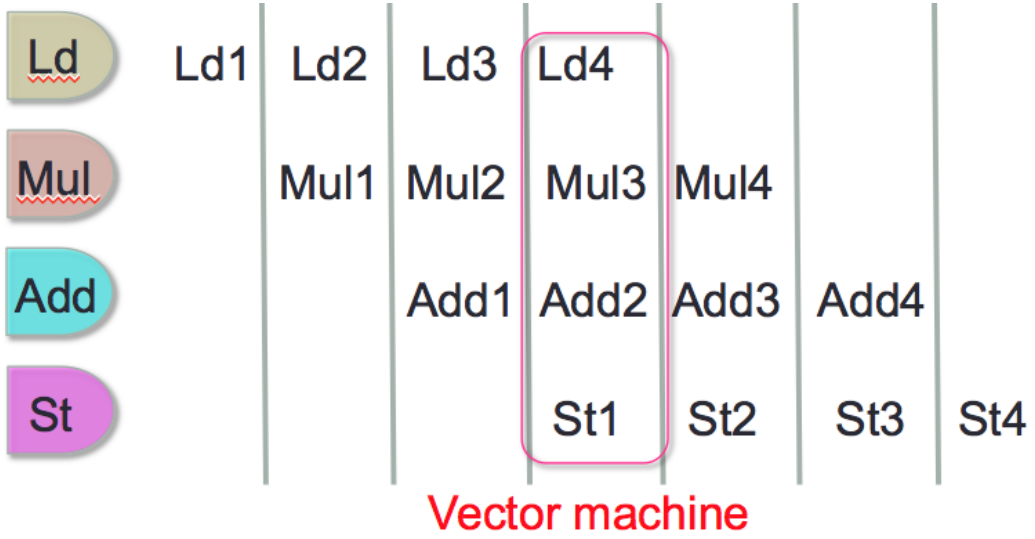
Execution Models / SIMD Machines

$$A = [a1, a2, a3, a4]$$

$$A = 2A + 3$$

```
VLd VR0,mem(A)
VMul VR0,VR0,2
VAdd VR0,VR0,3
VSt VR0,mem(A)
```

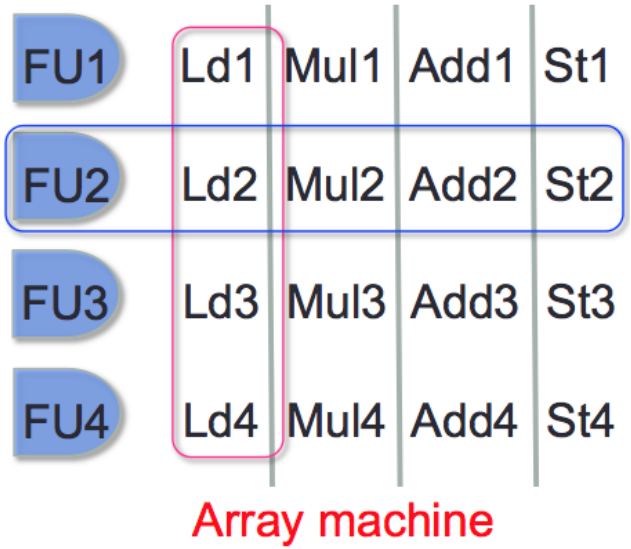


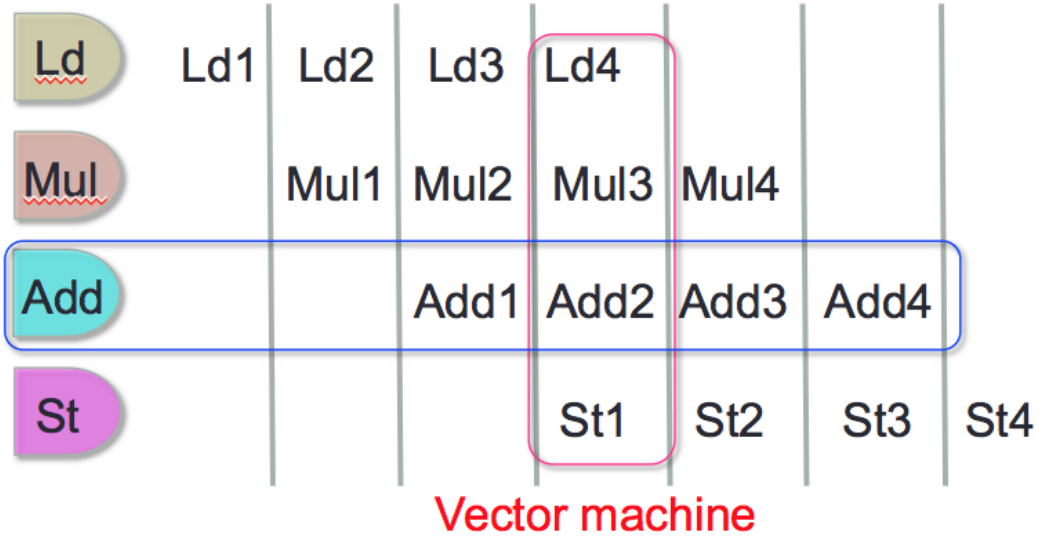


$$A = [a1, a2, a3, a4]$$

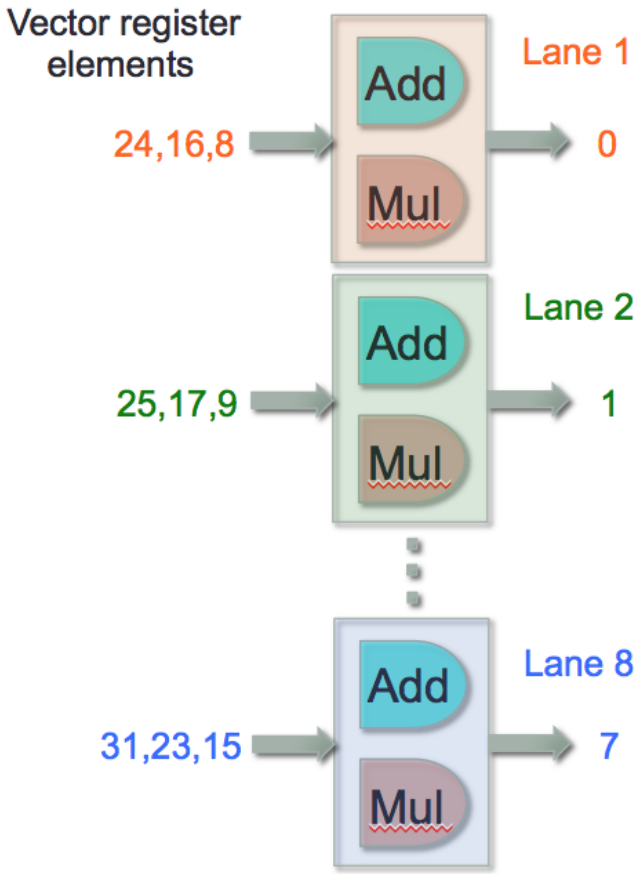
$$A = 2A + 3$$

```
VLd VR0,mem(A)
VMul VR0,VR0,2
VAdd VR0,VR0,3
VSt VR0,mem(A)
```

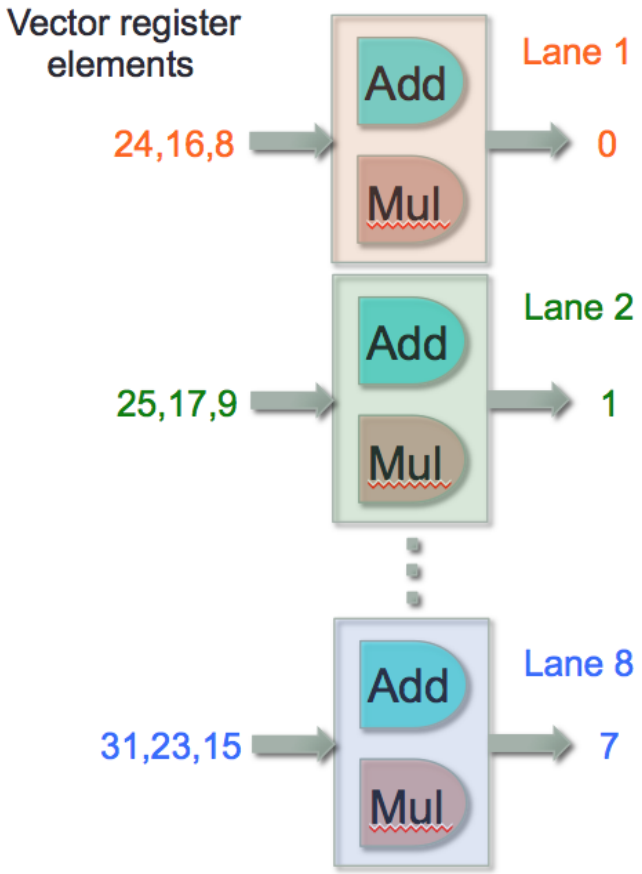




Execution Models / SIMD Machines



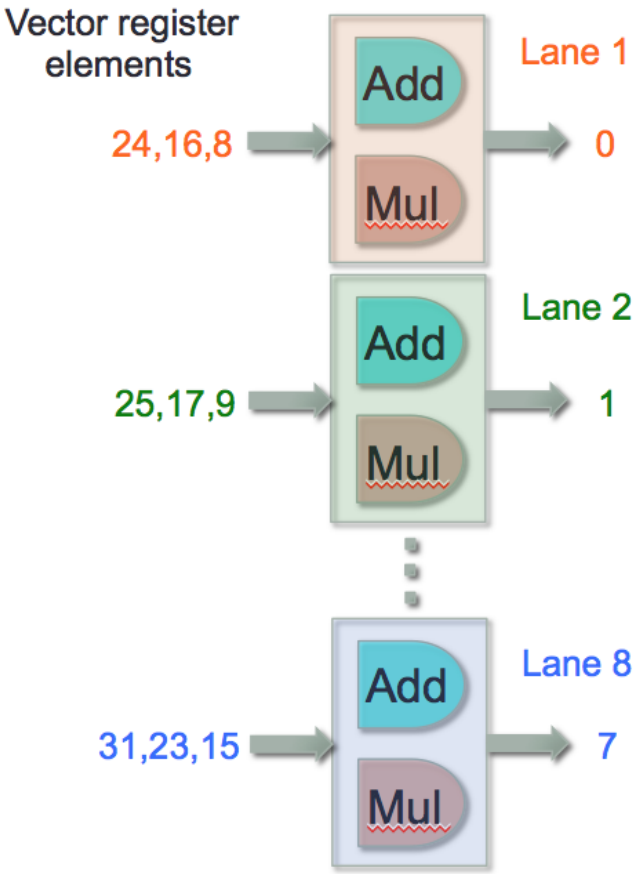
Execution Models / SIMD Machines



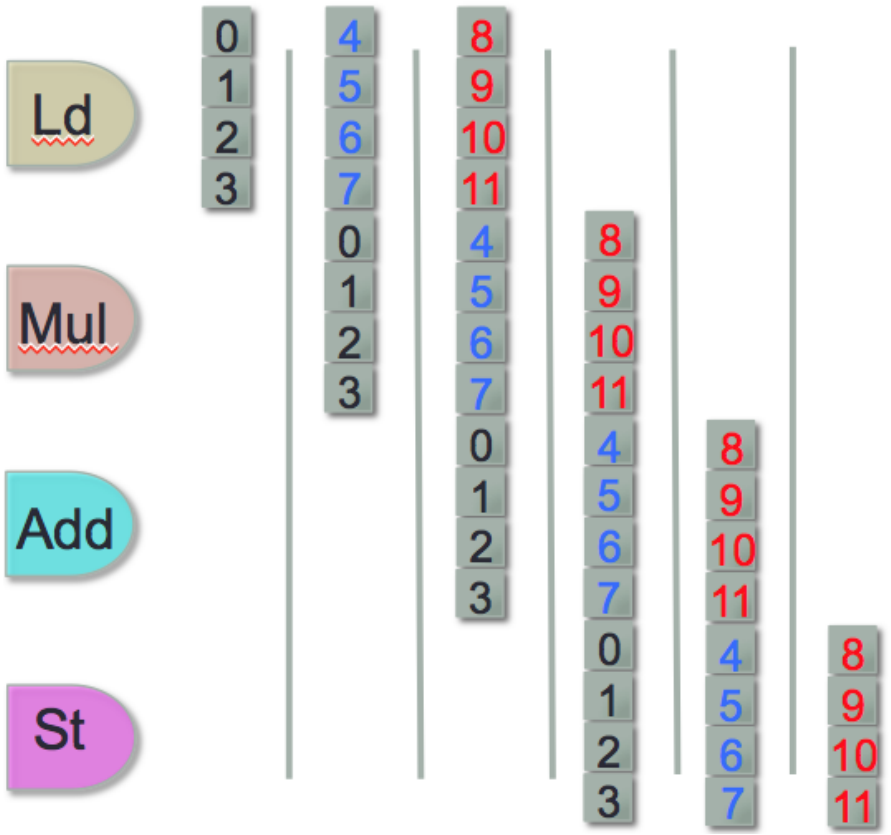
12 elements/vector. 4 lanes.

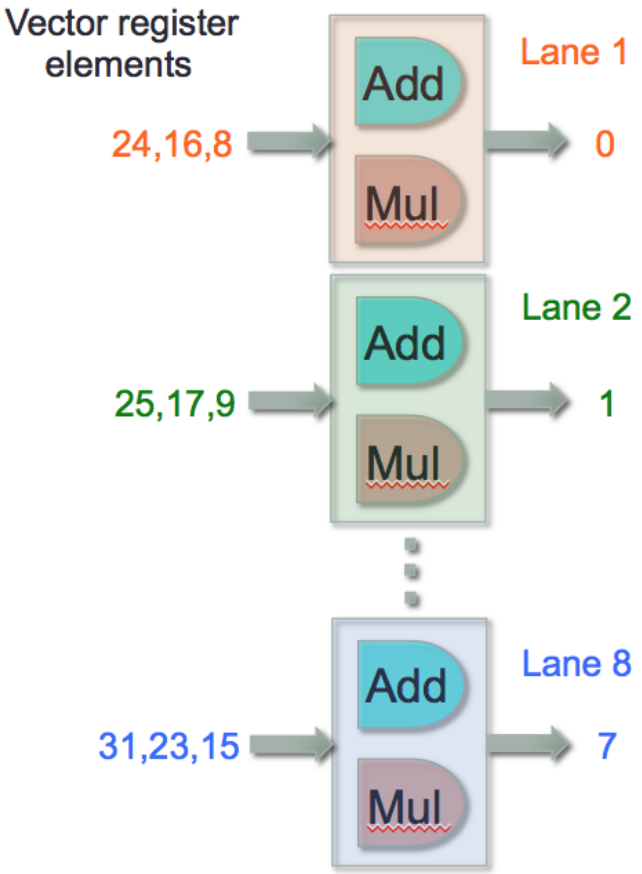
Execute `Ld`, `Mul`, `Add` and `St`

Execution Models / SIMD Machines

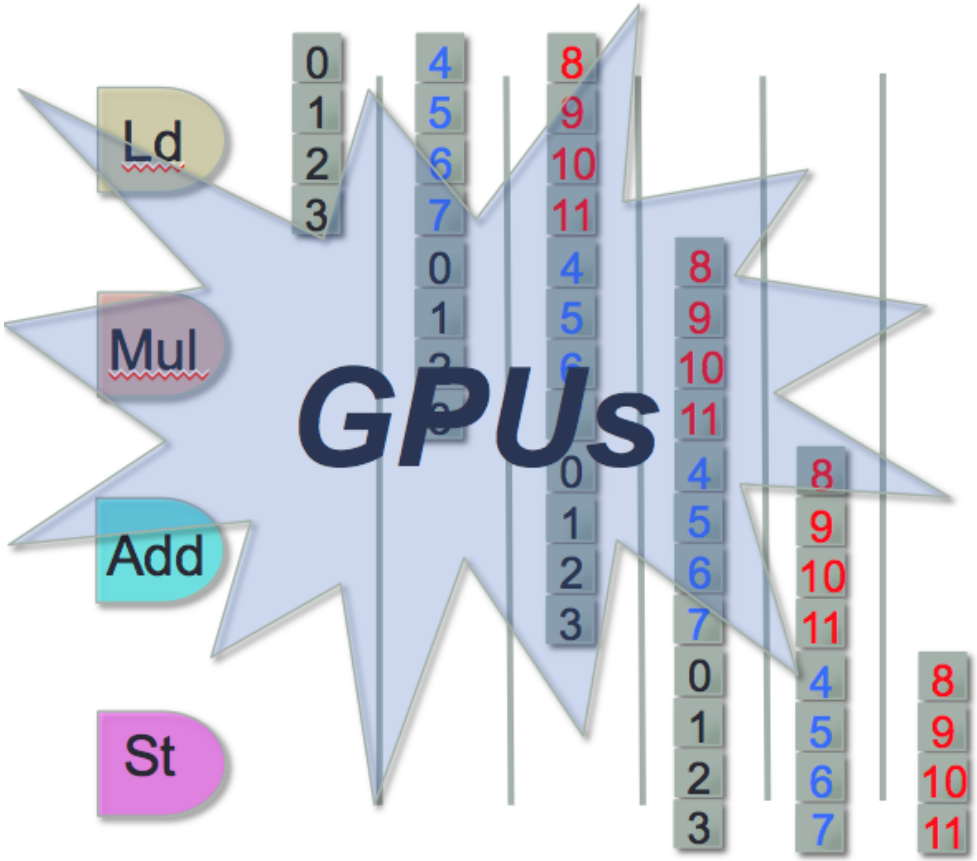


12 elements/vector. 4 lanes.
Execute `Ld`, `Mul`, `Add` and `St`





12 elements/vector. 4 lanes.
Execute `Ld`, `Mul`, `Add` and `St`



Back to GPUs...

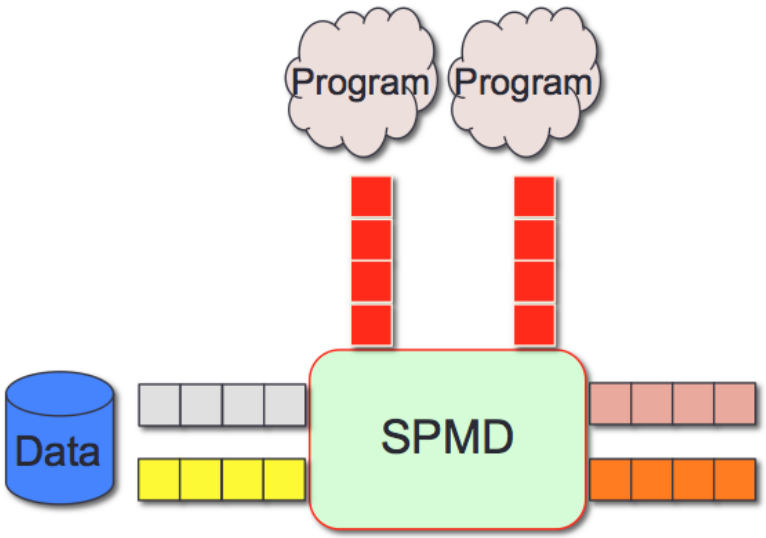
Back to GPUs...

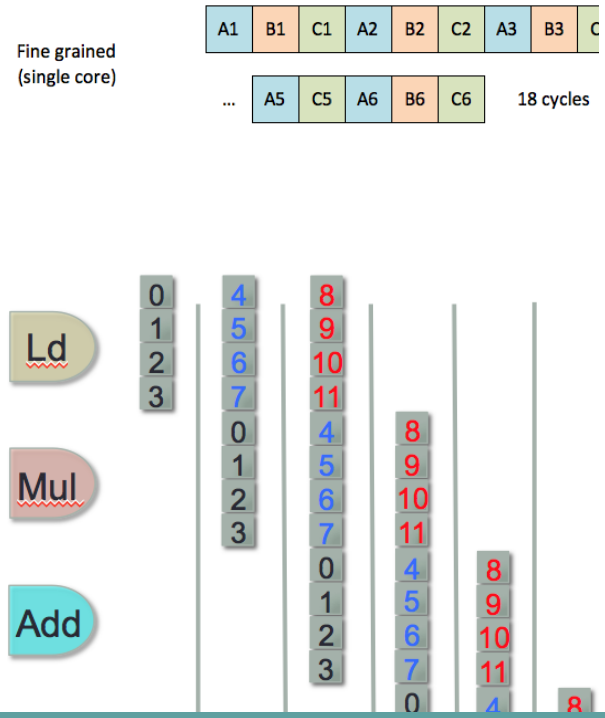
Programming Model

- Sequential (SISD)
- Data Parallelism (SIMD)
- Task Parallelism (MIMD)

Execution Model

- Pipeline, Out of order
- SIMD machines
- Multi threads/cores





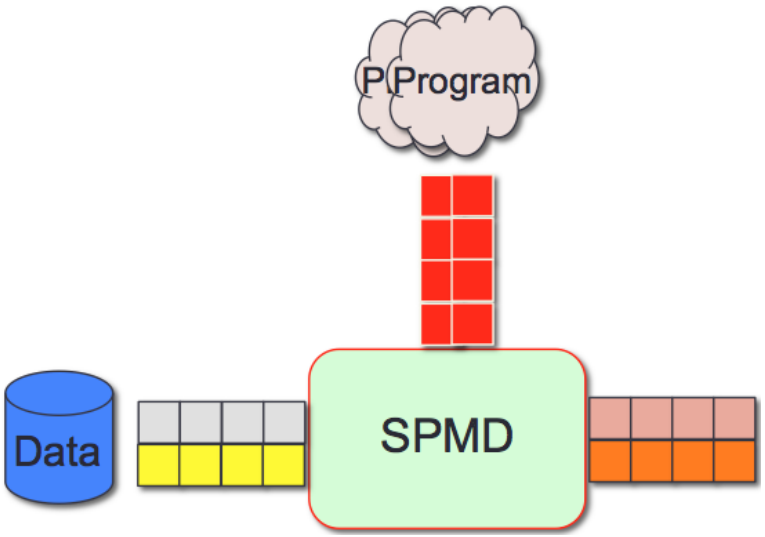
Back to GPUs...

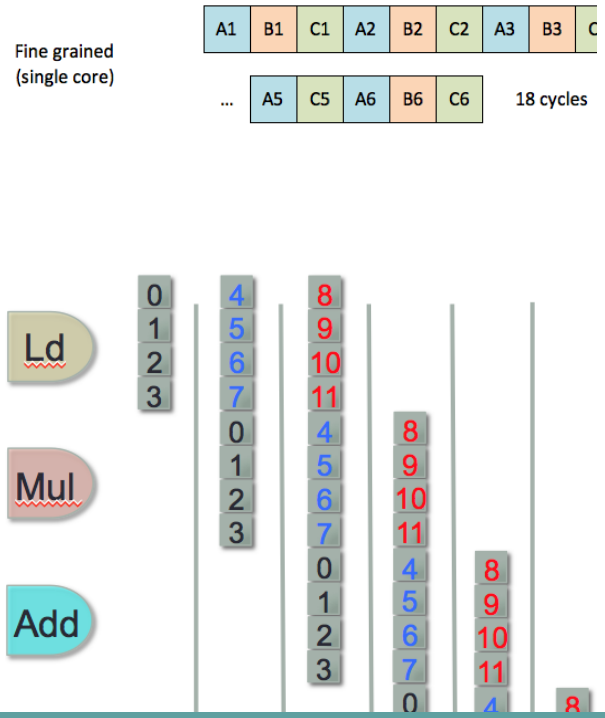
Programming Model

- Sequential (SISD)
- Data Parallelism (SIMD)
- Task Parallelism (MIMD)

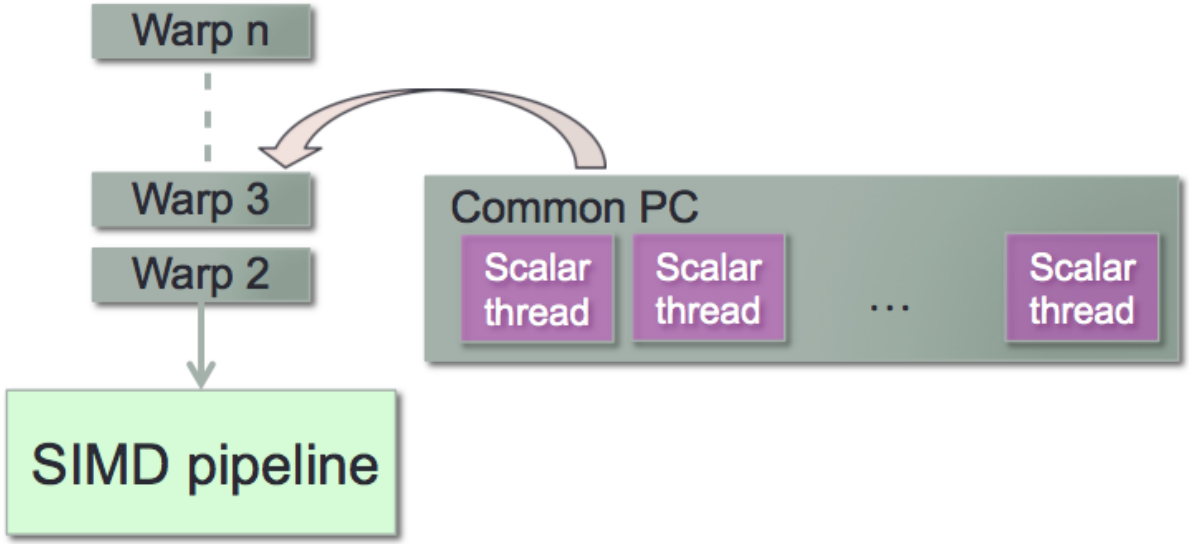
Execution Model

- Pipeline, Out of order
- SIMD machines
- Multi threads/cores



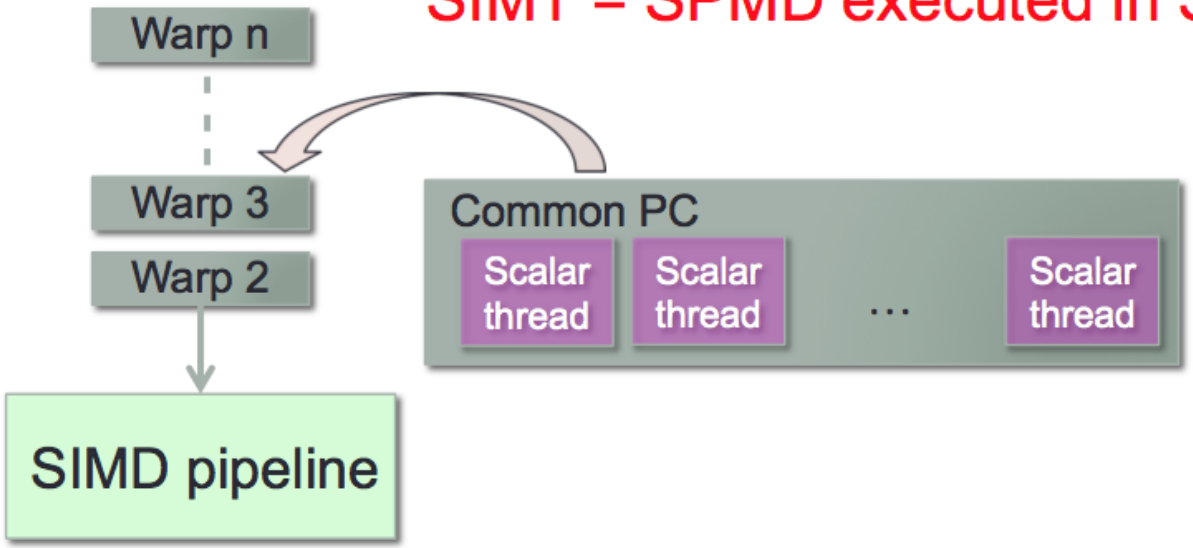


SIMT programming/execution model



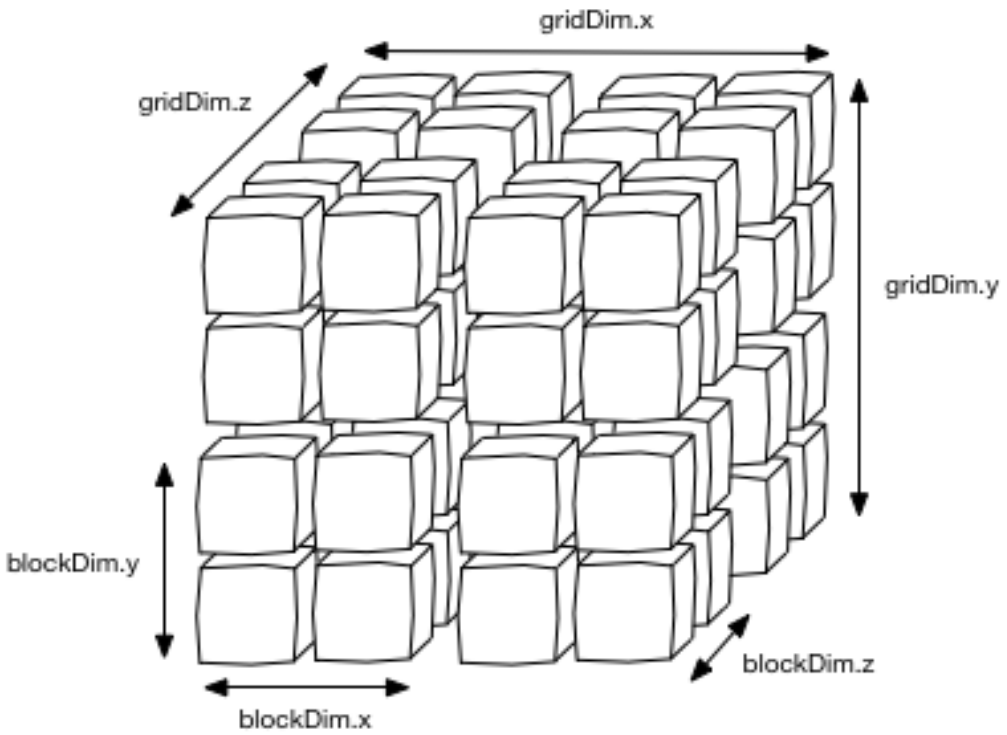
SIMT programming/execution model

SIMT = SPMD executed in SIMD



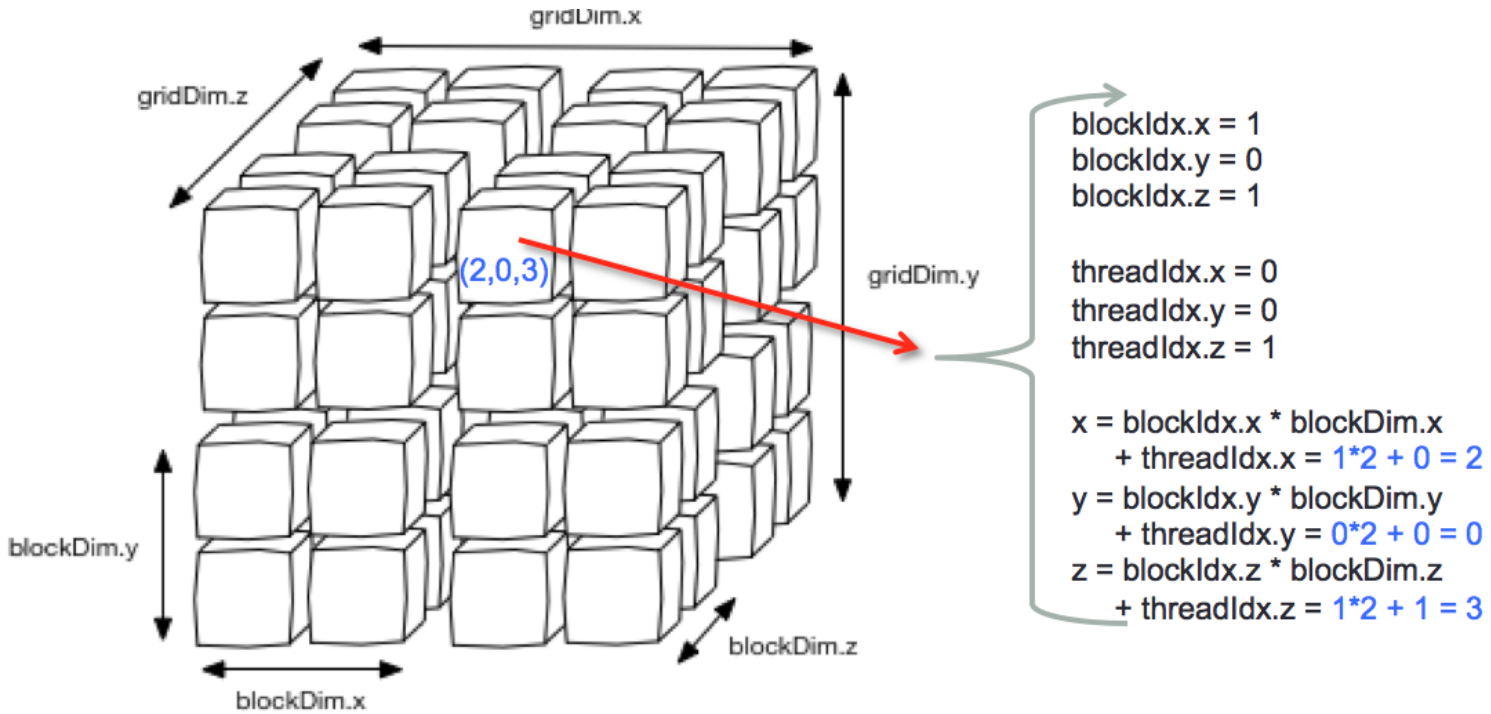
Thread hierarchy

- Threads are organized in three-dimensional blocks.
- A block is entirely assigned to a single streaming Multiprocessor.
- A block is subdivided in warps of 32 threads.
- All threads in a warp run in lockstep.
- Blocks are organized in a three-dimensional grid of blocks.



Thread hierarchy

- Threads are organized in three-dimensional blocks.
- A block is entirely assigned to a single streaming Multiprocessor.
- A block is subdivided in warps of 32 threads.
- All threads in a warp run in lockstep.
- Blocks are organized in a three-dimensional grid of blocks.



Thread hierarchy and memory model

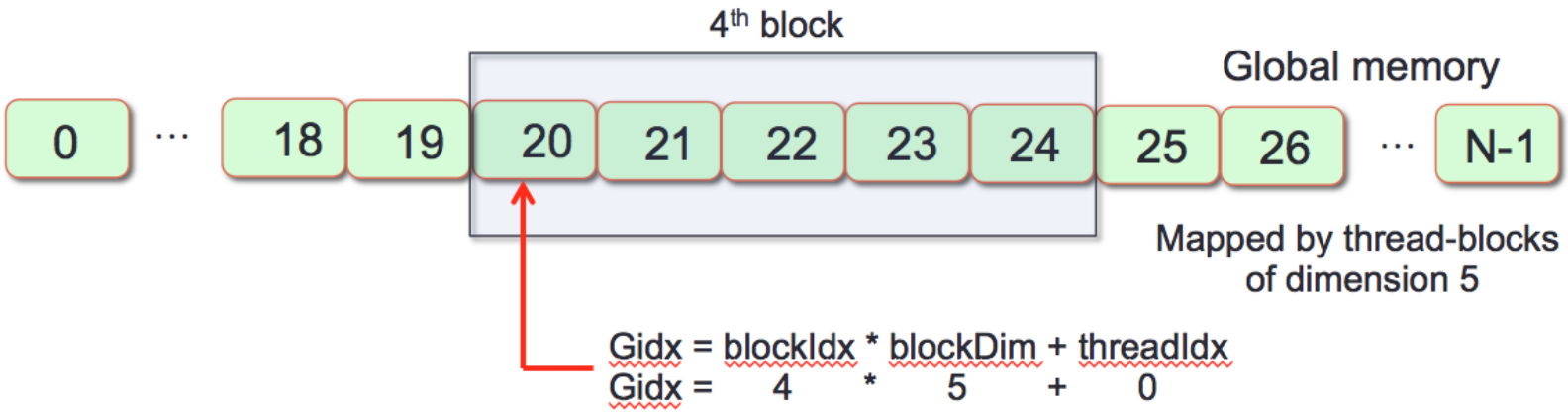
Memory type	Lives in	Managed by	Scope in host	Scope in device	Lifetime
Global	Device memory L1/L2 cache	programer (and the system)	read & write cudaMemCpy	All threads	Application lifetime
Constant	Device memory constant cache	programer (and the system)	read & write cudaMemCpy	All threads (read only)	Application lifetime
Registers	SM register file	Compiler	not visible	Per thread	Lifetime of Thread
local	Device memory L1/L2 cache	Compiler (as register spill)	not visible	Per thread	Lifetime of Thread
Shared	SM shared Memory	programer	not visible	All threads in the block	Lifetime of the block

NVIDIA Pascal P100 specs

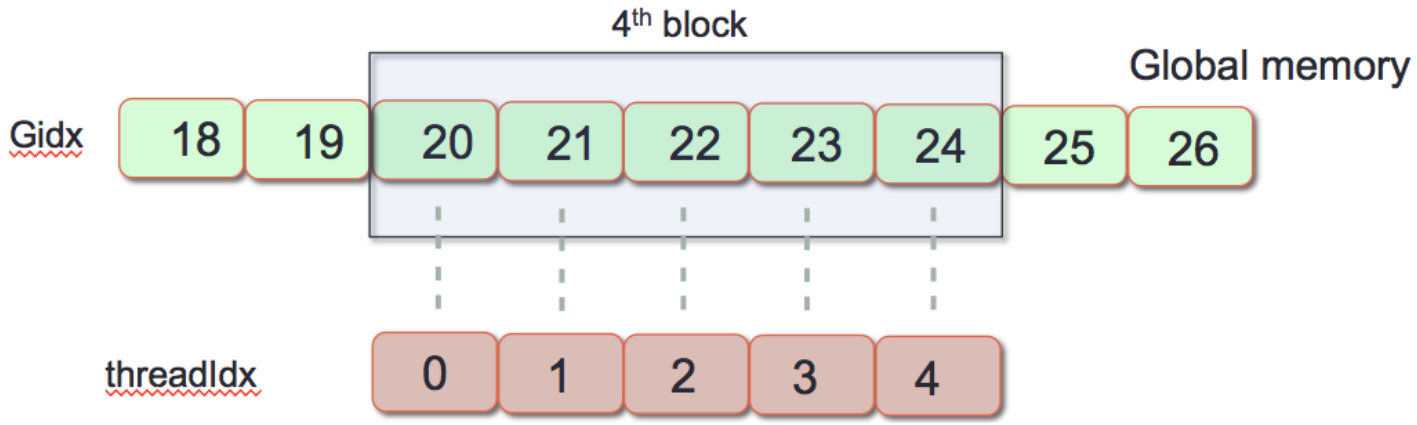
We can use CUDA API...
cudaGetDeviceProperties()

Using shared memory

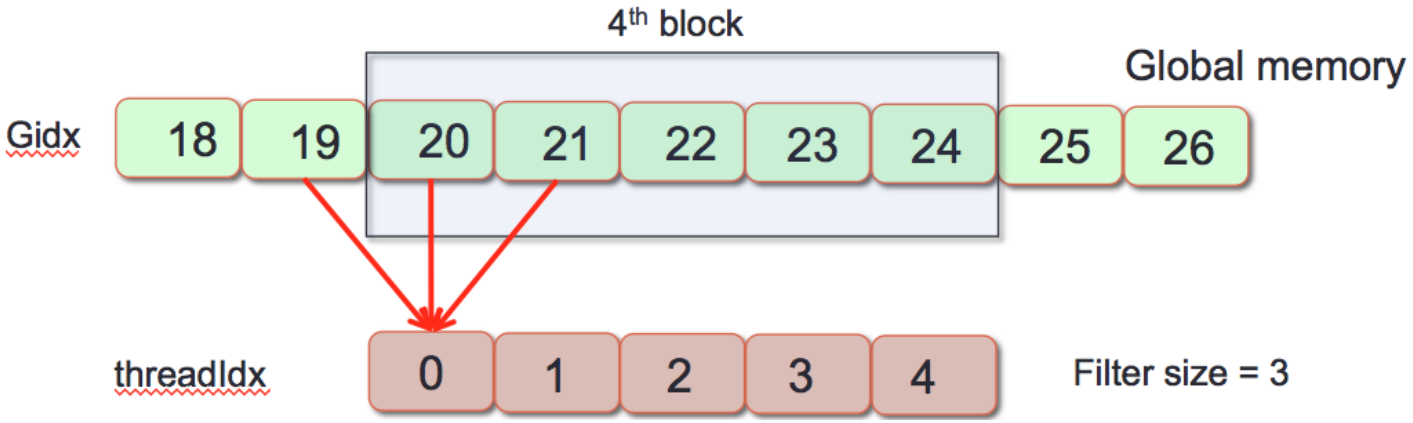
Using shared memory



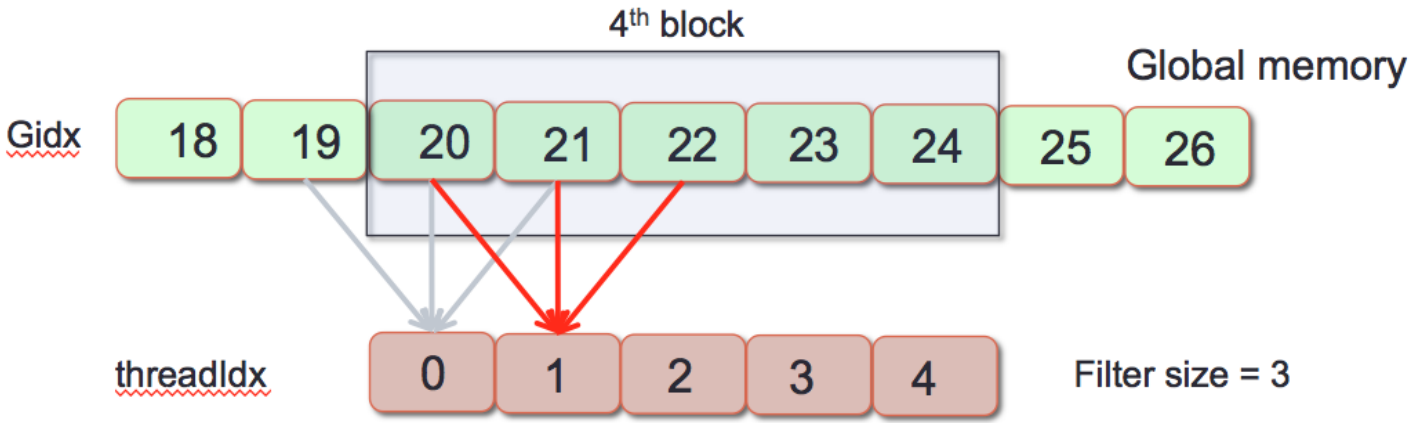
Using shared memory



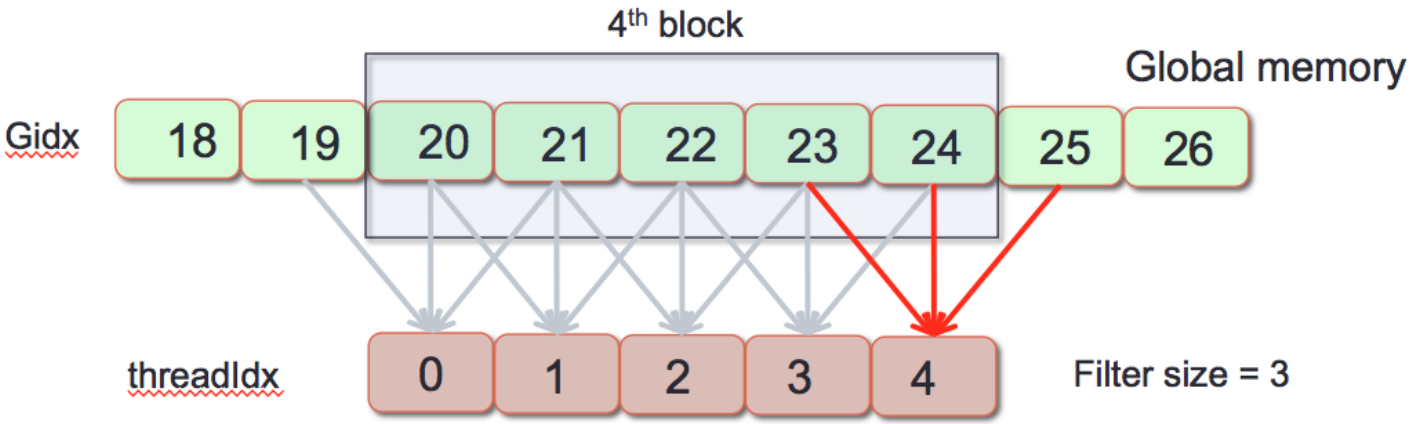
Using shared memory



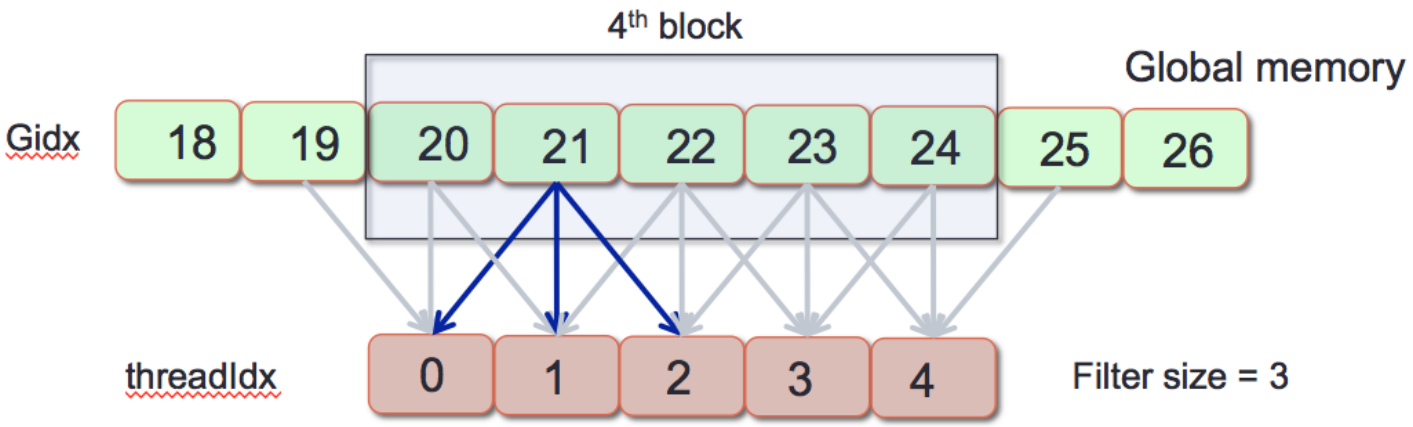
Using shared memory



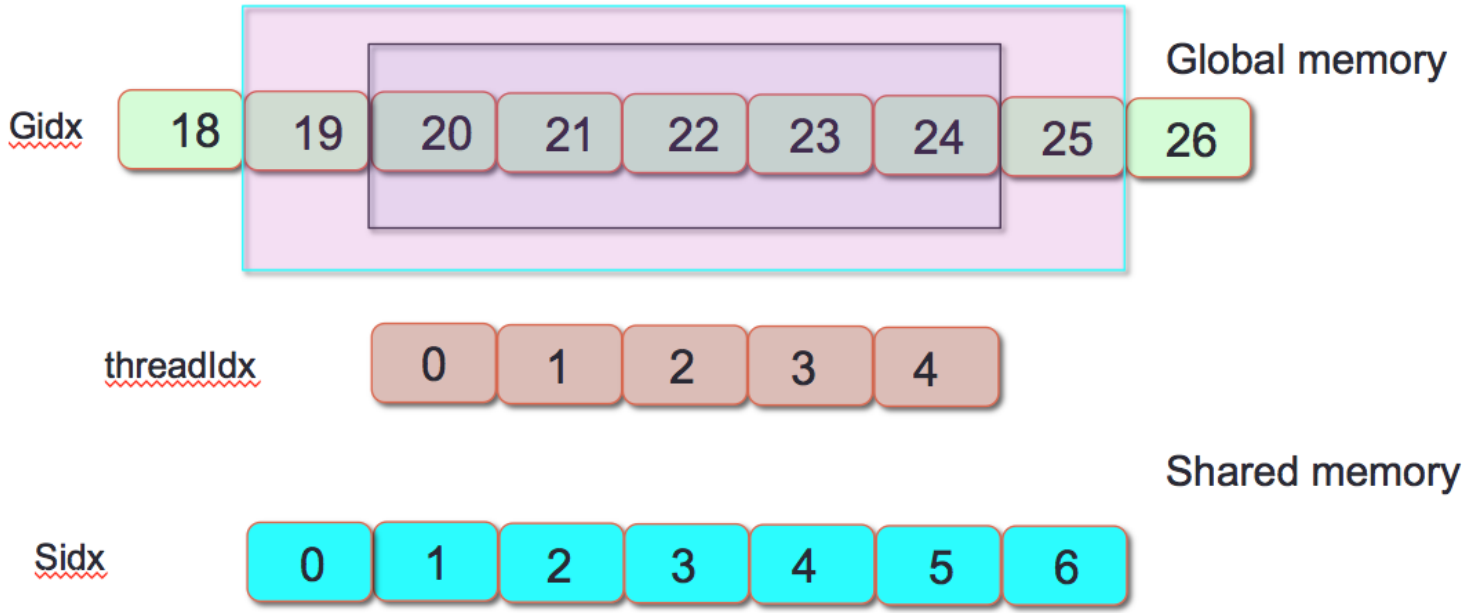
Using shared memory



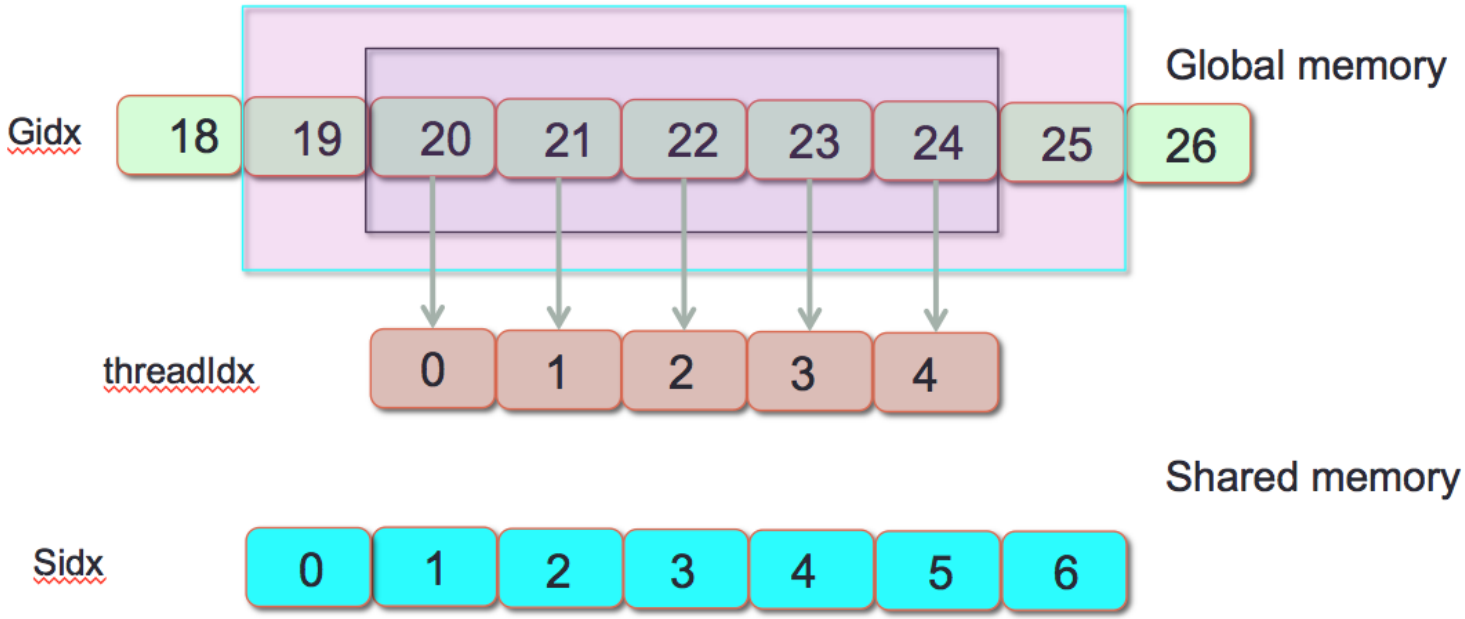
Using shared memory



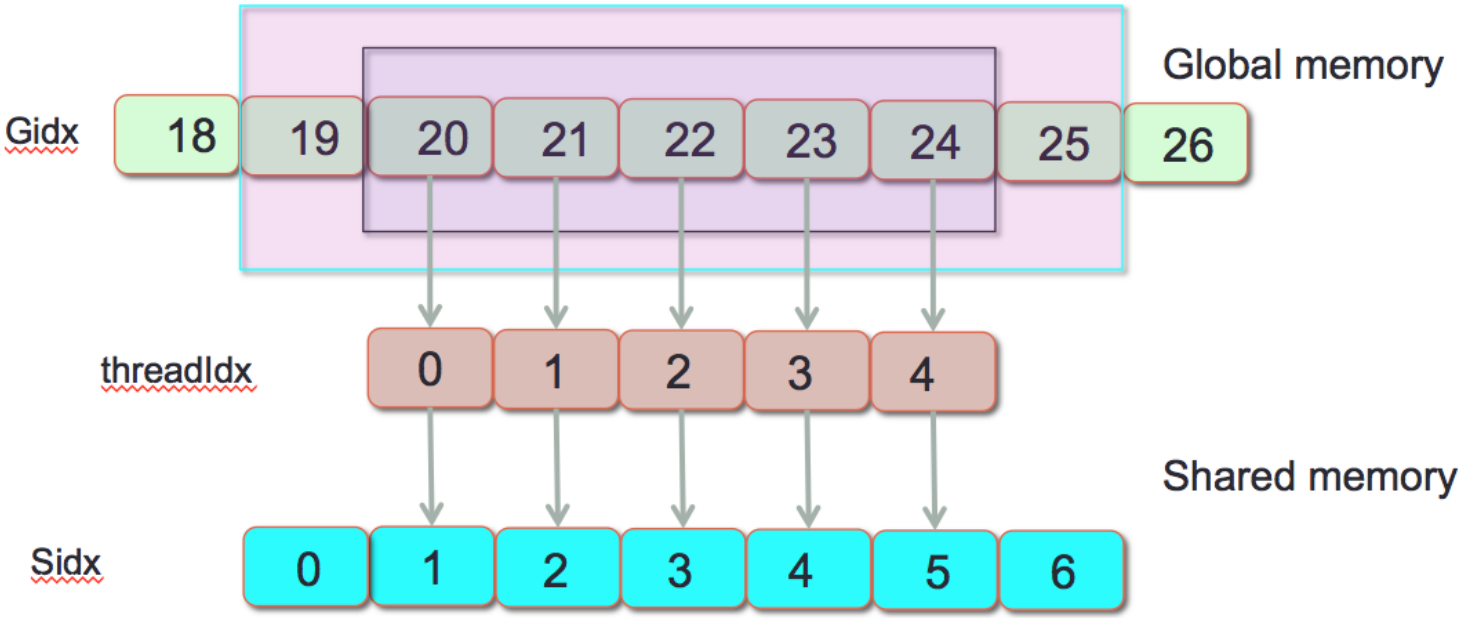
Using shared memory



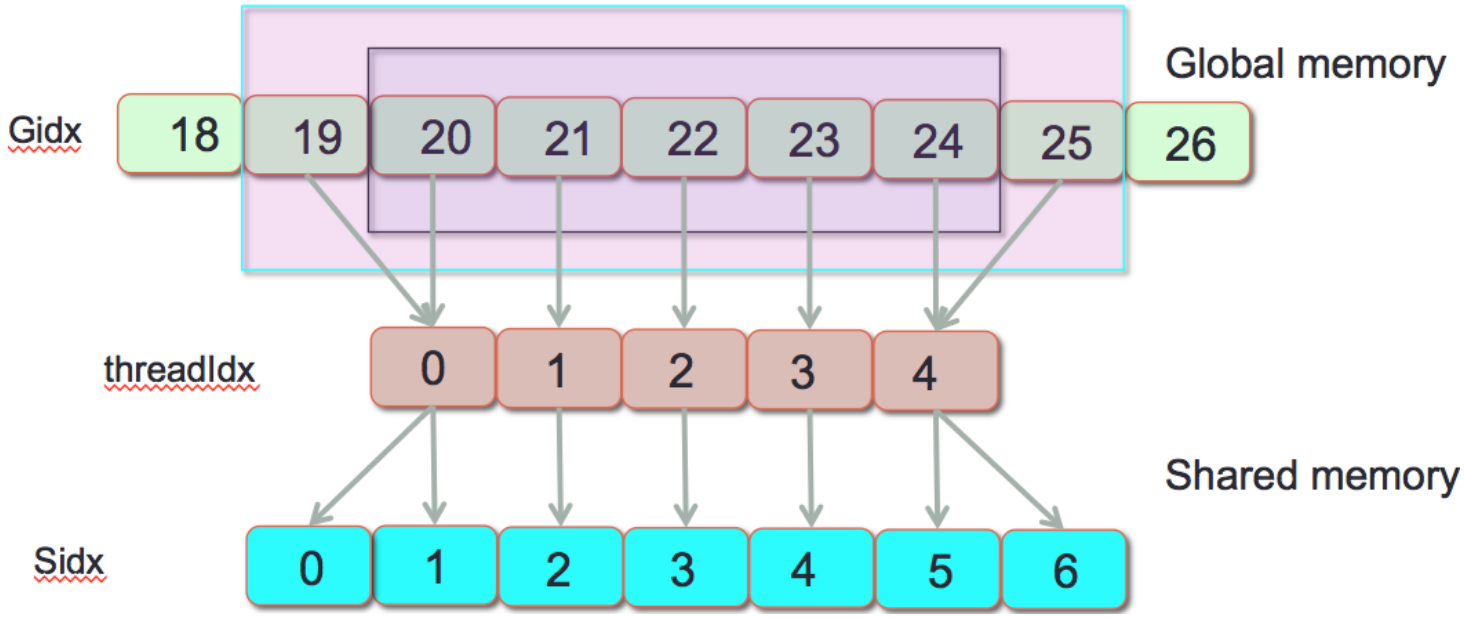
Using shared memory



Using shared memory



Using shared memory



Using shared memory