LECTURE 10

General Purpose GPU

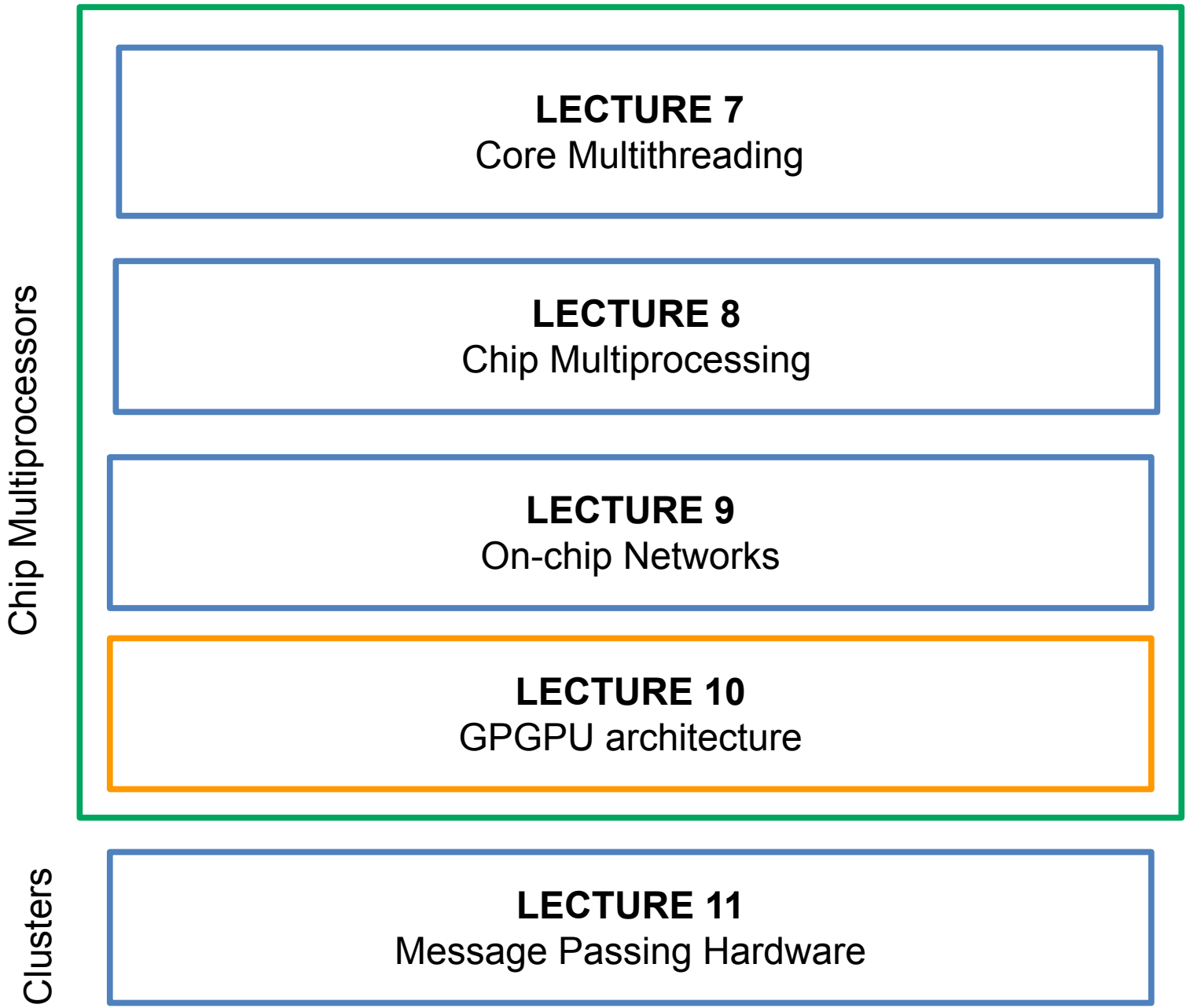
Miquel Pericàs

EDA284/DIT361 - VT 2020

What's cooking

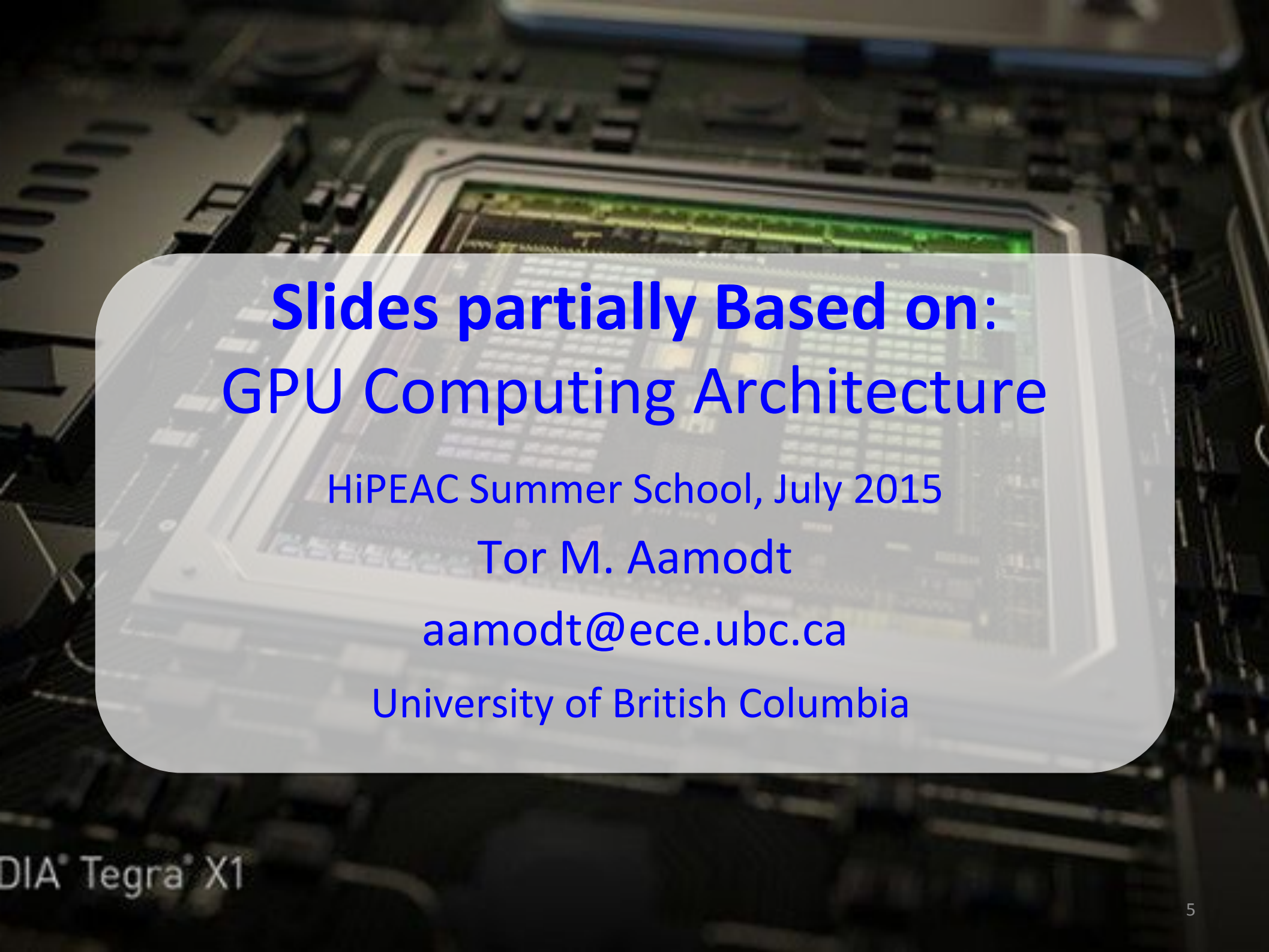
1. **Lectures**
   - Today: **Practice session #2** (8:30h-10h)
   - Today: **GPGPU** (10h-12h)
   - Tuesday Feb 25th (9h-12h) **Message Passing Hardware**
   - Wednesday Feb 26th (8h-10h) **Synchronization**
2. **Lab session**
   - Next Friday (8h-12h @ ED3507), GEM5 + Vector
3. **Project Status**
   - Feb 26th: deadline for first round of peer feedback

Lectures 7 - 11 Overview



Lecture 10: Outline

- Motivation for GPGPU
- SIMT Programming Model
- Two-stage compilation
- GPU Microarchitecture
- GPU Memory System



Slides partially Based on:
GPU Computing Architecture

HiPEAC Summer School, July 2015

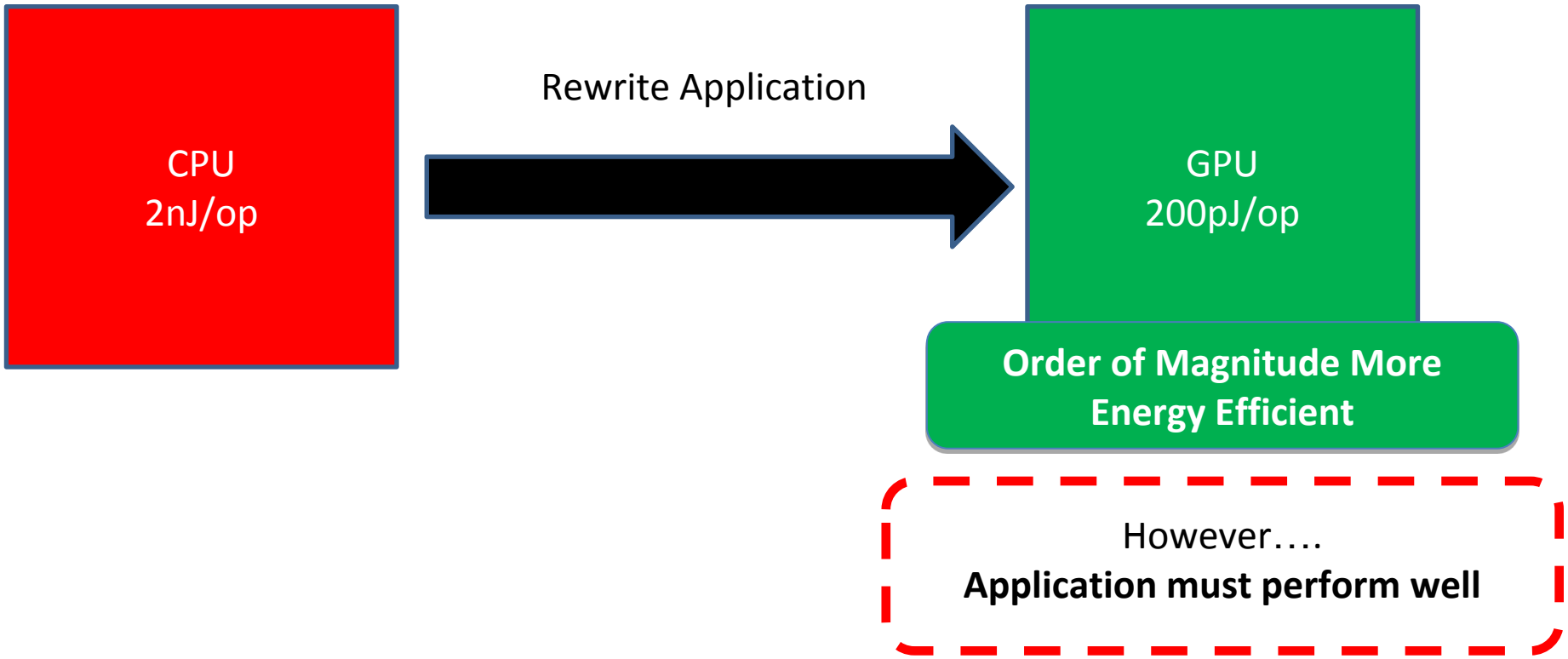
Tor M. Aamodt

aamodt@ece.ubc.ca

University of British Columbia

Why use a GPU for computing?

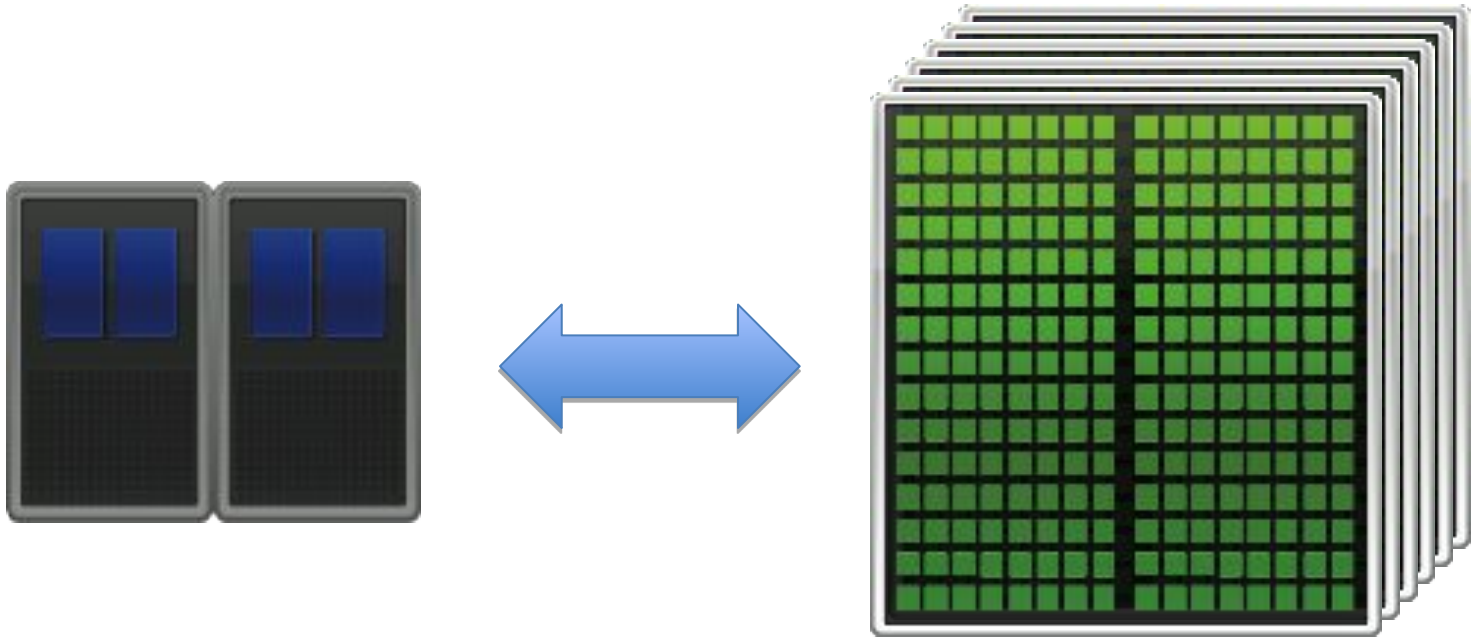
- GPU uses larger fraction of silicon for computation than CPU.
- At peak performance GPU uses order of magnitude less energy per operation than CPU.



What is a GPU? What is GPGPU?

- GPU = Graphics Processing Unit
  - Accelerator for raster based graphics (OpenGL, DirectX)
  - Highly programmable (Turing complete)
  - Commodity hardware
  - 100's of ALUs; 10's of 1000s of concurrent threads
- History of GPGPU computing
  - GPGPU = General-purpose GPU: Use GPU for General Purpose
  - First attempts: express programs using **OpenGL**, **DirectX**...
  - **Brook**, **Sh** (2002) add stream extensions to C language to remove graphics hardware details
  - **GPU Accelerator** (2006) added data-parallel arrays
  - Nvidia Compute Unified Device Architecture (**CUDA**, 2007)
  - **OpenCL** (2009), **OpenACC** (2011), **OpenMP 4.x+** (2013)

GPU Compute Programming Model

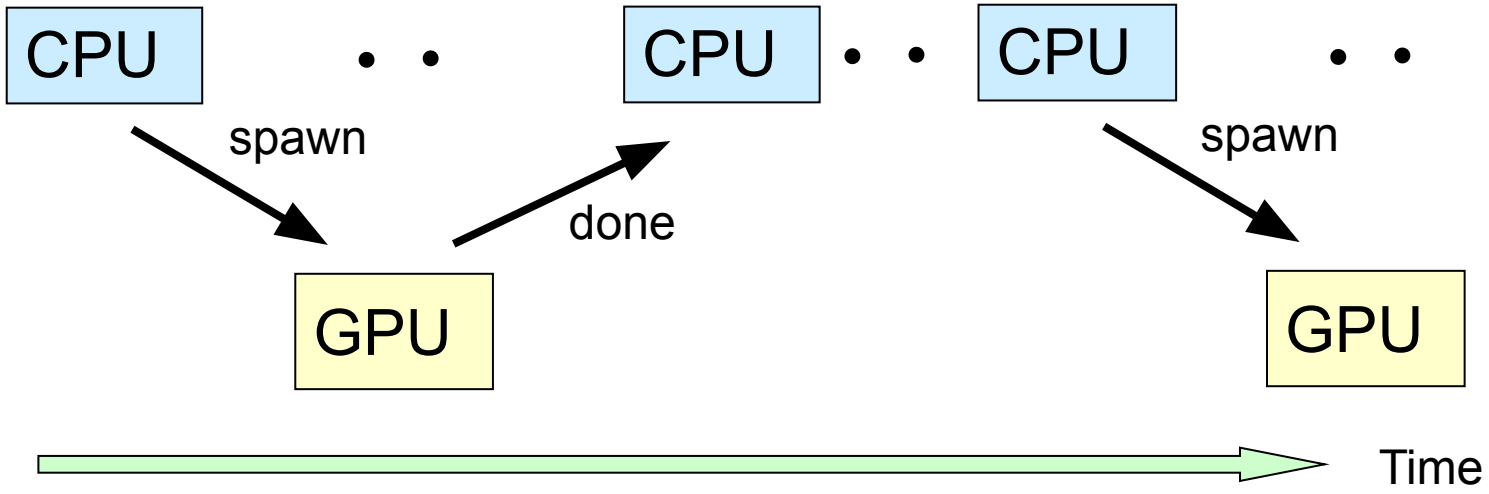


CPU GPU

How is this system programmed (today)?

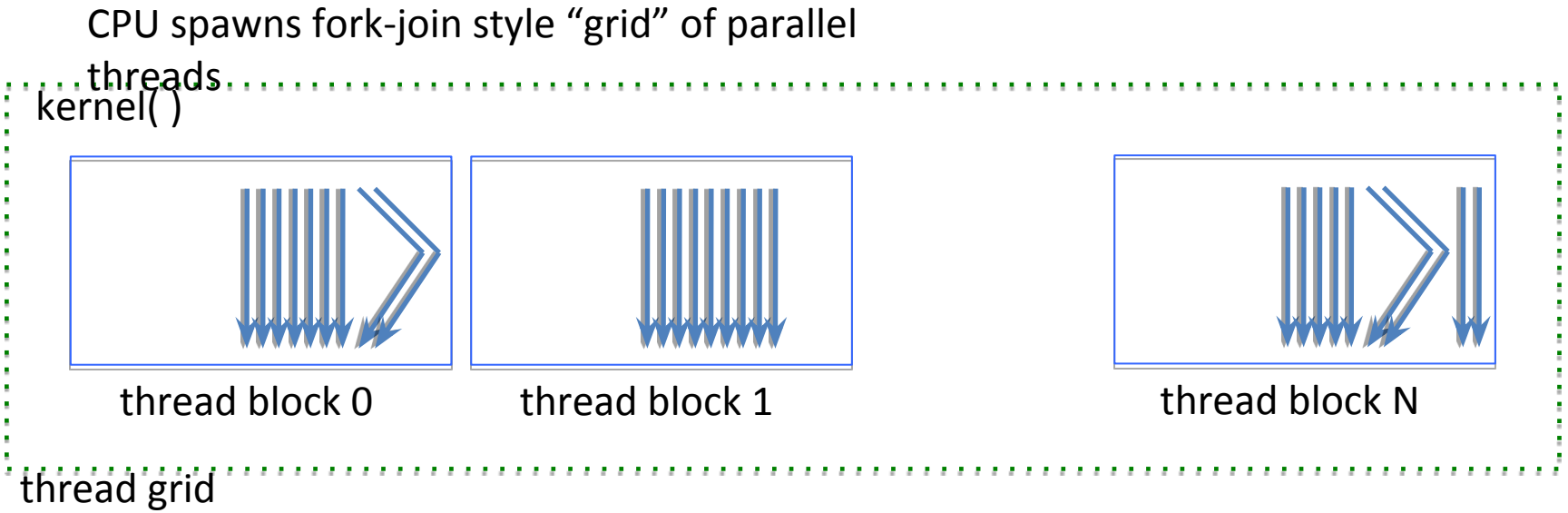
CPU+GPU Programming Model

- CPU “Off-load” parallel kernels to GPU



  - Transfer data to GPU memory
  - GPU HW spawns threads
  - Need to transfer result data back to CPU main memory

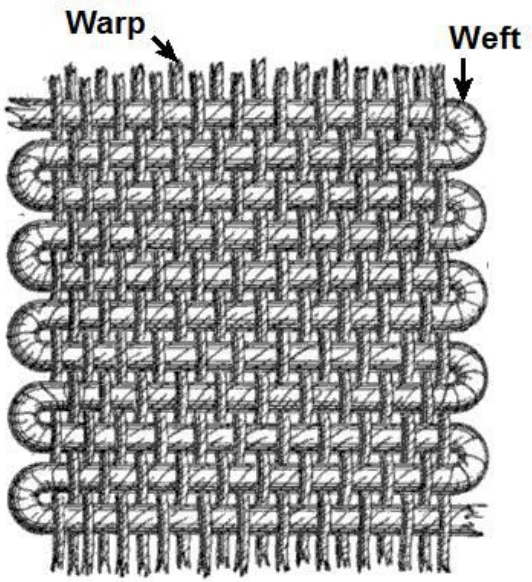
CUDA/OpenCL Threading Model



- Spawns more threads than GPU can run (some may wait)
- Organize threads into “blocks” (up to 1024 threads per block)
- Threads can communicate/synchronize with other threads in block
- Threads/Blocks have an identifier (can be 1, 2 or 3 dimensional)
- Each kernel spawns a “grid” containing 1 or more thread blocks.
- Motivation: Write parallel software once and run on future hardware

SIMT Execution Model

- Single-Instruction Multiple-Thread (SIMT)
- Programmer sees MIMD threads (scalar) in SPMD (single-program multiple-data) programming model
- GPU bundles threads into warps (wavefronts) and runs them in lockstep on SIMD hardware
- An NVIDIA *warp* groups 32 consecutive threads together (AMD *wavefronts* group 64 threads together)
- Aside: the term warp originates from the textile industry



[https://en.wikipedia.org/wiki/Warp_and_weft]

CUDA Syntax Extensions

- Declaration specifiers

__global__ void foo(...); // kernel entry point (runs on GPU)
__device__ void bar(...); // function callable from a GPU thread

- Syntax for kernel launch

foo<<<500, 128>>>(...); // 500 thread blocks, 128 threads each

- Built in variables for thread identification

dim3 threadIdx; dim3 blockIdx; dim3 blockDim;

Example: Original C Code

SAXPY stands for "Single-Precision A * X Plus Y". It is a function in the standard Basic Linear Algebra Subroutines (BLAS) library

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}

int main() {
  // omitted: allocate and initialize memory
  saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY kernel
  // omitted: using result
}
```

CUDA Code

```
__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]+y[i];
}
```

Runs on GPU

```
int main() {
  // omitted: allocate and initialize memory
  int nblocks = (n + 255) / 256;

  cudaMalloc((void**) &d_x, n);
  cudaMalloc((void**) &d_y, n);
  cudaMemcpy(d_x,h_x,n*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_y,h_y,n*sizeof(float),cudaMemcpyHostToDevice);
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  cudaMemcpy(h_y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
  // omitted: using result
}
```

h_x, h_y: host addresses
d_x, d_y: device addresses

OpenCL Code

```
__kernel void saxpy(int n, float a, __global float *x, __global float *y) {
   int i = get_global_id(0);
   if(i<n) y[i]=a*x[i]+y[i];
}
```

Runs on GPU

```
int main() {
  // omitted: allocate and initialize memory on host, variable declarations

  int nblocks = (n + 255) / 256;
  int blocksize = 256;

  clGetPlatformIDs(1, &cpPlatform, NULL);
  clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);
  cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
  cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr1);
  dx = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float) * n, NULL, &ciErr1);
  dy = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE, sizeof(cl_float) * n, NULL, &ciErr1);

  // omitted: loading program into char string cSourceCL
  cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char **)&cSourceCL, &szKernelLength,
    &ciErr1);
  clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL);
  ckKernel = clCreateKernel(cpProgram, "saxpy_serial", &ciErr1);

  clSetKernelArg(ckKernel, 0, sizeof(cl_int),   (void*)&n);
  clSetKernelArg(ckKernel, 1, sizeof(cl_float), (void*)&a);
  clSetKernelArg(ckKernel, 2, sizeof(cl_mem),   (void*)&dx);
  clSetKernelArg(ckKernel, 3, sizeof(cl_mem),   (void*)&dy);

  clEnqueueWriteBuffer(cqCommandQueue, dx, CL_FALSE, 0, sizeof(cl_float) * n, x, 0, NULL, NULL);
  clEnqueueWriteBuffer(cqCommandQueue, dy, CL_FALSE, 0, sizeof(cl_float) * n, y, 0, NULL, NULL);
  clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL, &nblocks, & blocksize, 0, NULL, NULL);
  clEnqueueReadBuffer(cqCommandQueue, dy, CL_TRUE, 0, sizeof(cl_float) * n,   y, 0, NULL, NULL);

  // omitted: using result
}
```

OpenMP 4.0 & SYCL Example

```
// OpenMP: directive-based parallel programming model
void saxpy_openmp(int n, float a, float *x, float *y)
{
  #pragma omp target teams distribute parallel for
  for (int i = 0; i < n; ++i)
     y[i] = a*x[i] + y[i];
}
```

Runs on GPU

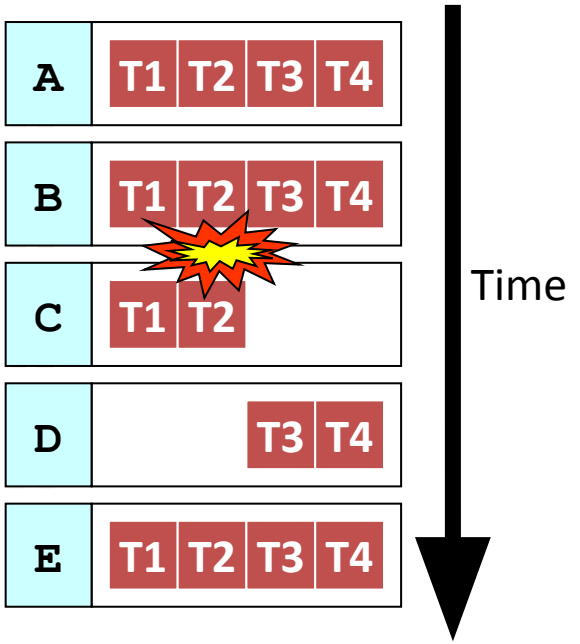
```
// SYCL: single-source domain specific embedded language (DSEL)
// based on pure C++11
void saxpy_sycl(int n, float a, float *x, float *y)
{
  …
  cgh.parallel_for(range<1>(n),
  [=](id<1> idx){ y[i] = a * x[i] + y[i]; });
  …
}
```

Runs on GPU

SIMT Execution Model

- Challenge: How to handle branch operations when different threads in a warp follow a different path through program? This is called branch divergence: e.g. { `if(i < n) … }`
- Solution: Serialize different paths.

```
 foo[] = {4,8,12,16};
A: v = foo[threadIdx.x];
B: if (v < 10)
C:     v = 0;
   else
D:     v = 10;
E: w = bar[threadIdx.x]+v;
```

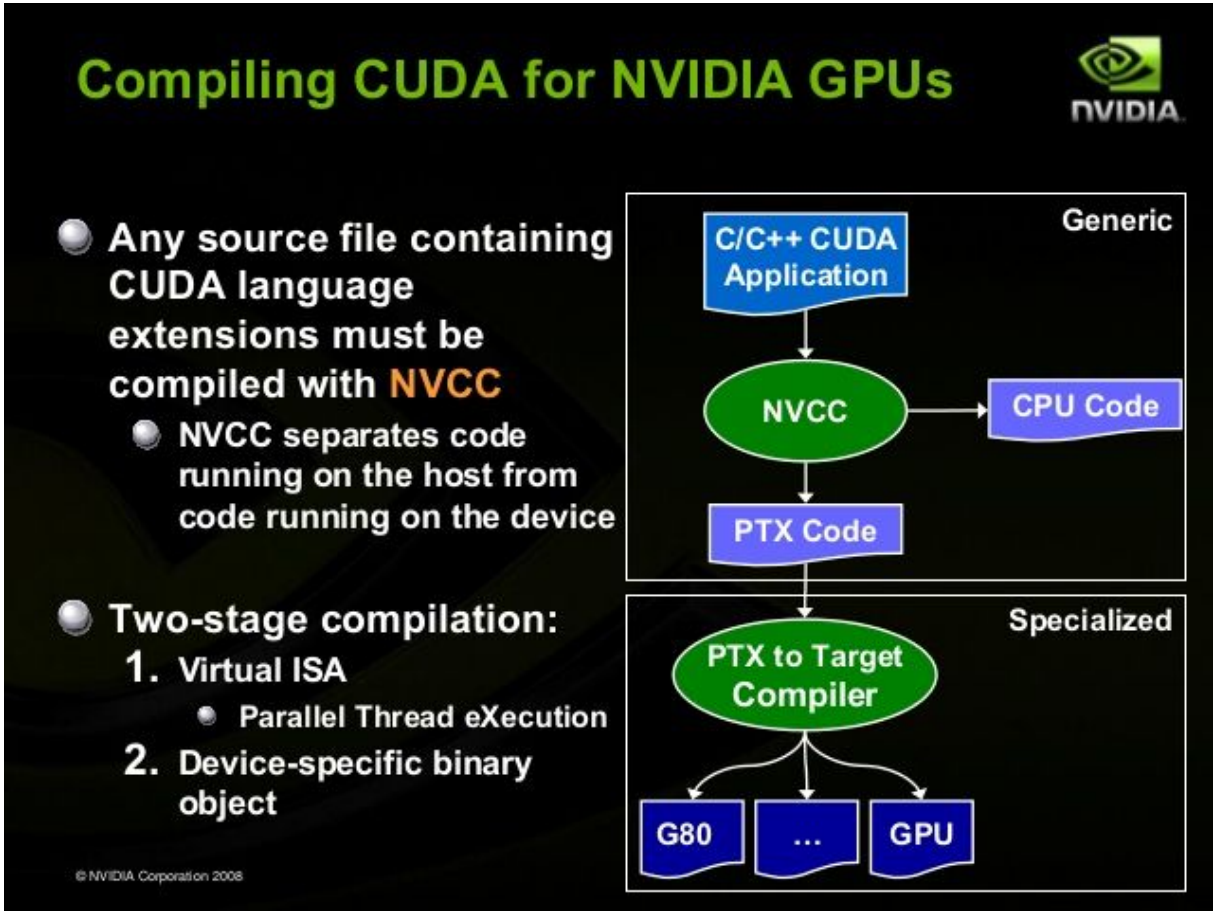


- A predicate register is used to track active/inactive threads
- Simple, but downside is reduced performance during divergence

Why SIMT and not "pure" SIMD?

1. Simplifies programming: thread-centric programming style without explicit parallelization
2. Intermediate representation independent of the back-end: provides portability to other devices.

Both approaches overcome limitations of the pure SIMD approach



Note: runtime-specialization means that no backwards compatibility needs to be provided by the actual GPU ISA!

What is PTX? How does it work?

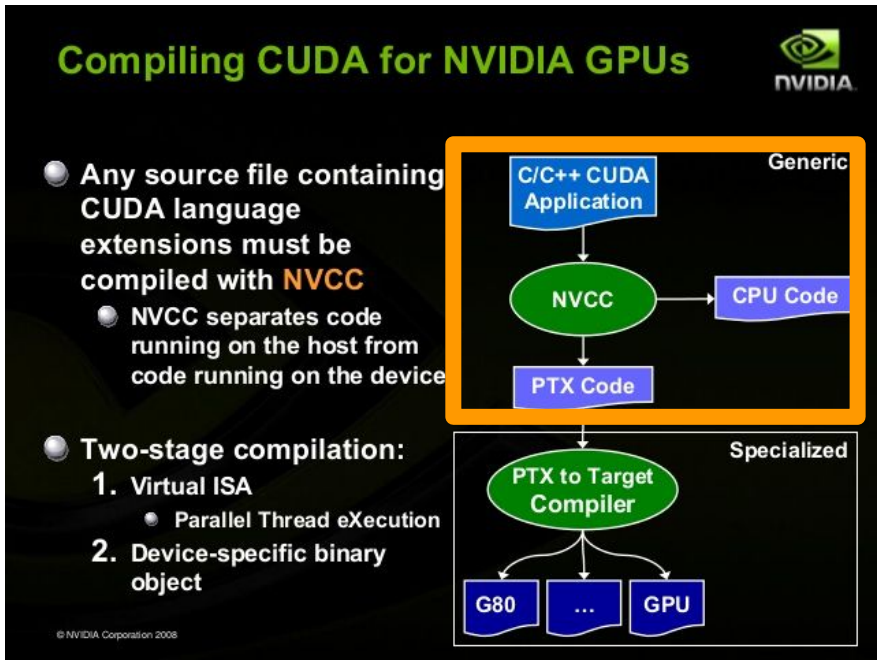
- High-level virtual instruction set architecture for GPU
  - PTX = Parallel Thread Execution ISA (by Nvidia)
- similar to a RISC ISA (eg ARM, MIPS, SPARC, or ALPHA)
- shares similarity to intermediate representations used within optimizing compilers, e.g. limitless set of virtual registers
- before running PTX code, need to compile PTX down to the actual GPU ISA (called SASS, “Streaming ASSembler”)
- PTX to SASS compilation accomplished either by GPU driver or `ptxas` assembler (part of NVIDIA’s CUDA Toolkit)
- AMD follows similar approach, the virtual ISA is HSAIL (Heterogeneous System Architecture Intermediate Language)

CUDA SAXPY and PTX

```
__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]+y[i];
}
```

SAXPY CUDA kernel




```
.visible .entry _Z5saxpyifPfS_(
.param .u32 _Z5saxpyifPfS__param_0,
.param .f32 _Z5saxpyifPfS__param_1,
.param .u64 _Z5saxpyifPfS__param_2,
.param .u64 _Z5saxpyifPfS__param_3
)
{
.reg .pred %p<2>;
.reg .f32 %f<5>;
.reg .b32 %r<6>;
.reg .b64 %rd<8>;


ld.param.u32 %r2, [_Z5saxpyifPfS__param_0];
ld.param.f32 %f1, [_Z5saxpyifPfS__param_1];
ld.param.u64 %rd1, [_Z5saxpyifPfS__param_2];
ld.param.u64 %rd2, [_Z5saxpyifPfS__param_3];
mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %ntid.x;
mov.u32 %r5, %tid.x;
mad.lo.s32 %r1, %r4, %r3, %r5;
setp.ge.s32 %p1, %r1, %r2;
@%p1 bra BB0_2;

cvta.to.global.u64 %rd3, %rd2;
cvta.to.global.u64 %rd4, %rd1;
mul.wide.s32 %rd5, %r1, 4;
add.s64 %rd6, %rd4, %rd5;
ld.global.f32 %f2, [%rd6];
add.s64 %rd7, %rd3, %rd5;
ld.global.f32 %f3, [%rd7];
fma.rn.f32 %f4, %f2, %f1, %f3;
st.global.f32 [%rd7], %f4;

BB0_2:
ret;
}
```

"virtual" predicate register

declare registers

load parameters into registers

`blockIdx.x*blockDim.x + threadIdx.x;`

check predicate (i<n)

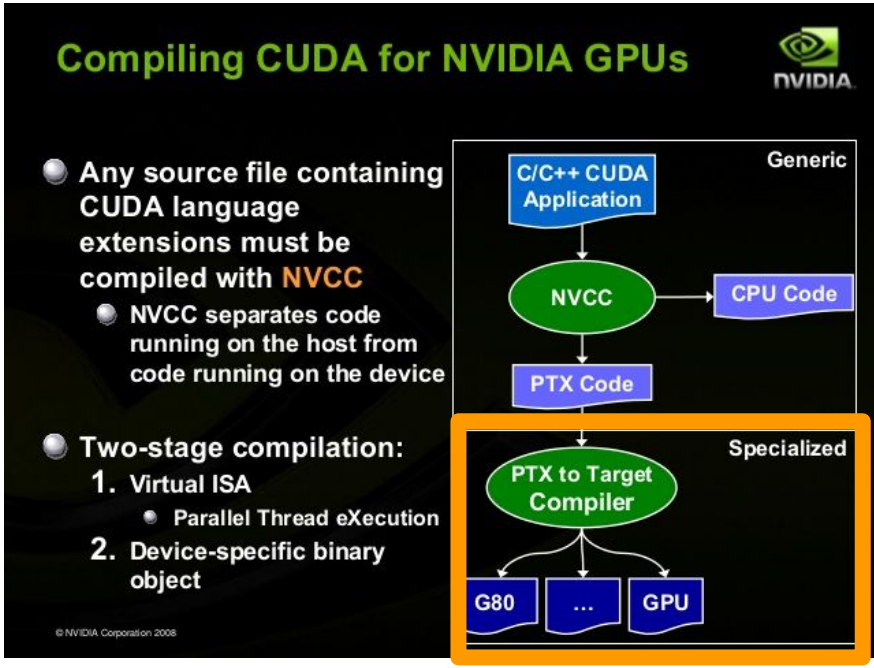
jump if (i >= n)

`y[i]=a*x[i]+y[i];`

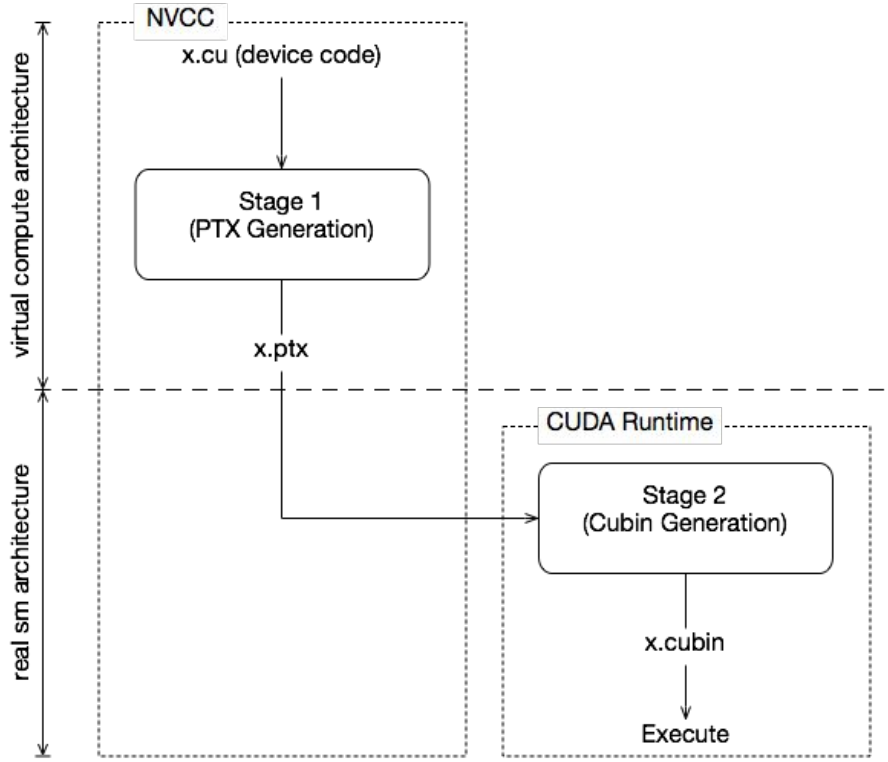
SAXPY PTX kernel

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

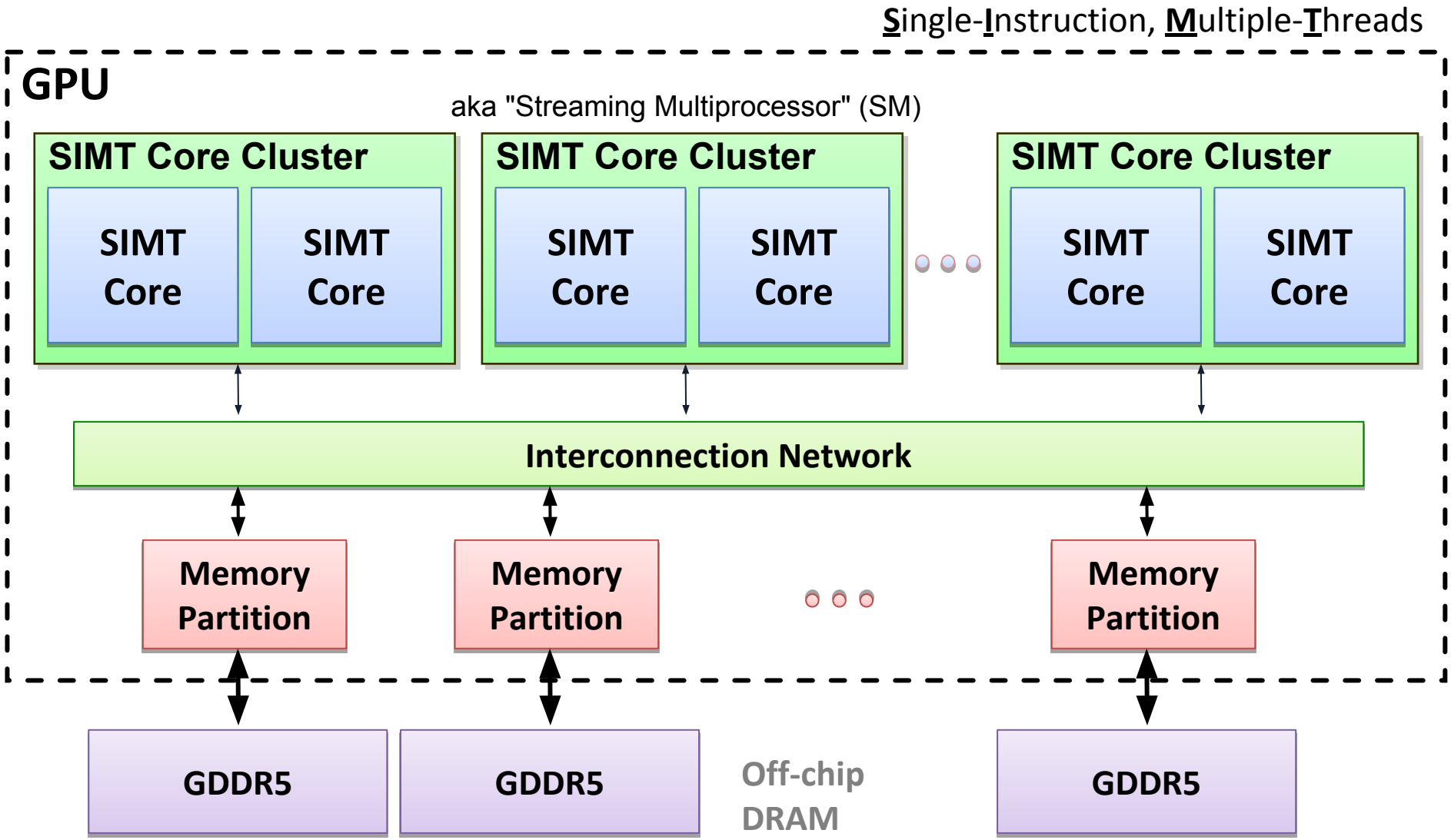
Just-in-Time Compilation



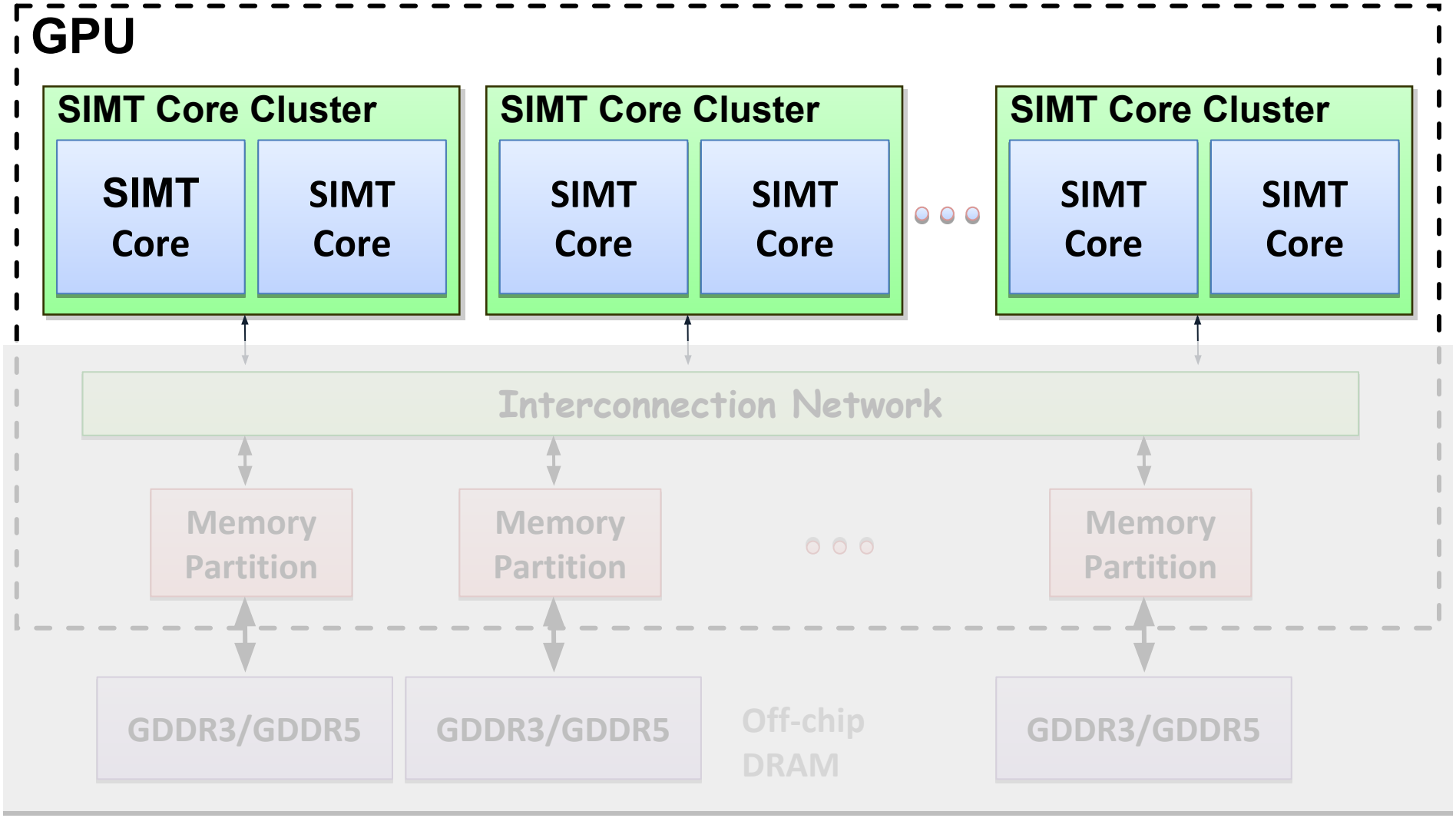
- The PTX code is compiled further to binary code by device driver ahead of execution:
  - "just-in-time compilation" (JIT)
- JIT increases application load time, but application can benefit from new compiler improvements.
- Also: application can run on devices that did not exist at the time the application was compiled!
  - remember: GPUs do not ensure backward compatibility
- When device driver JIT-compiles some PTX code, it automatically caches a copy of the generated binary to reduce future application load time.



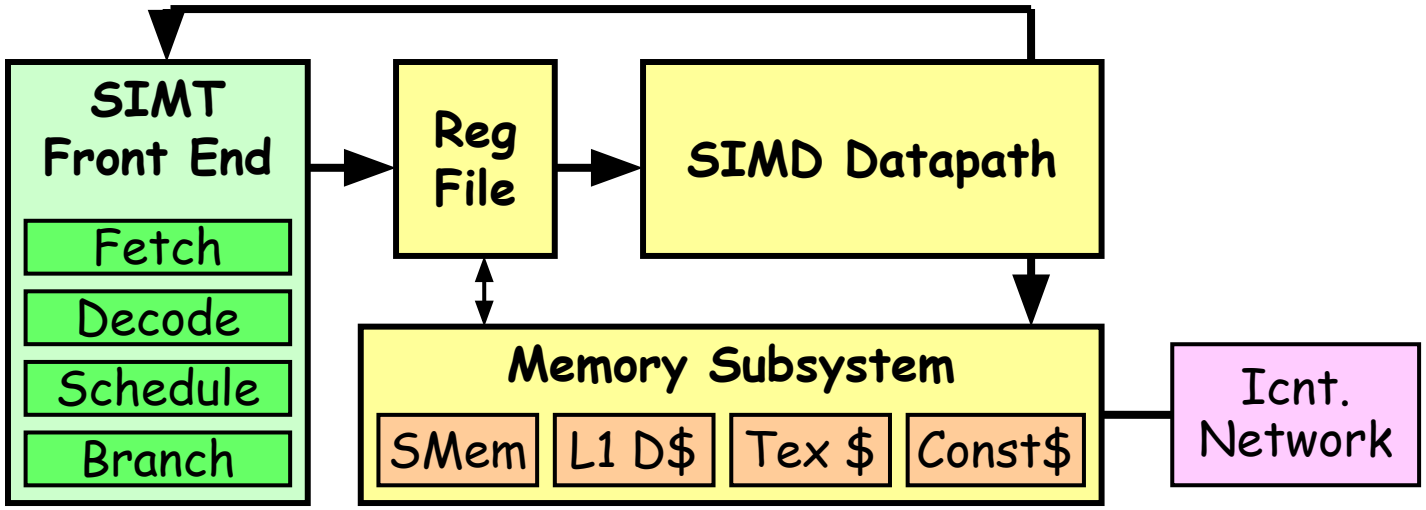
GPU Microarchitecture Overview



GPU Microarchitecture Overview

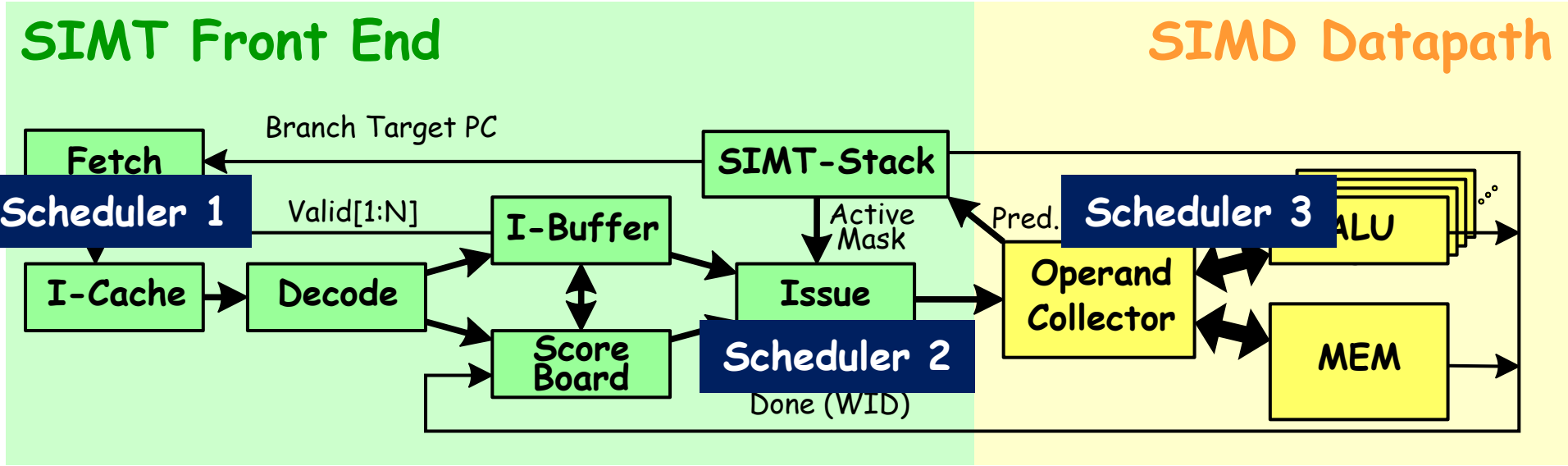


Inside a SIMT Core



- SIMT front end / SIMD backend
- Fine-grained multithreading
  - Interleave warp execution to hide latency
  - Register values of all threads stays in core
  - Caches are small, but still effective. Why?

Inside an "NVIDIA-style" SIMT Core

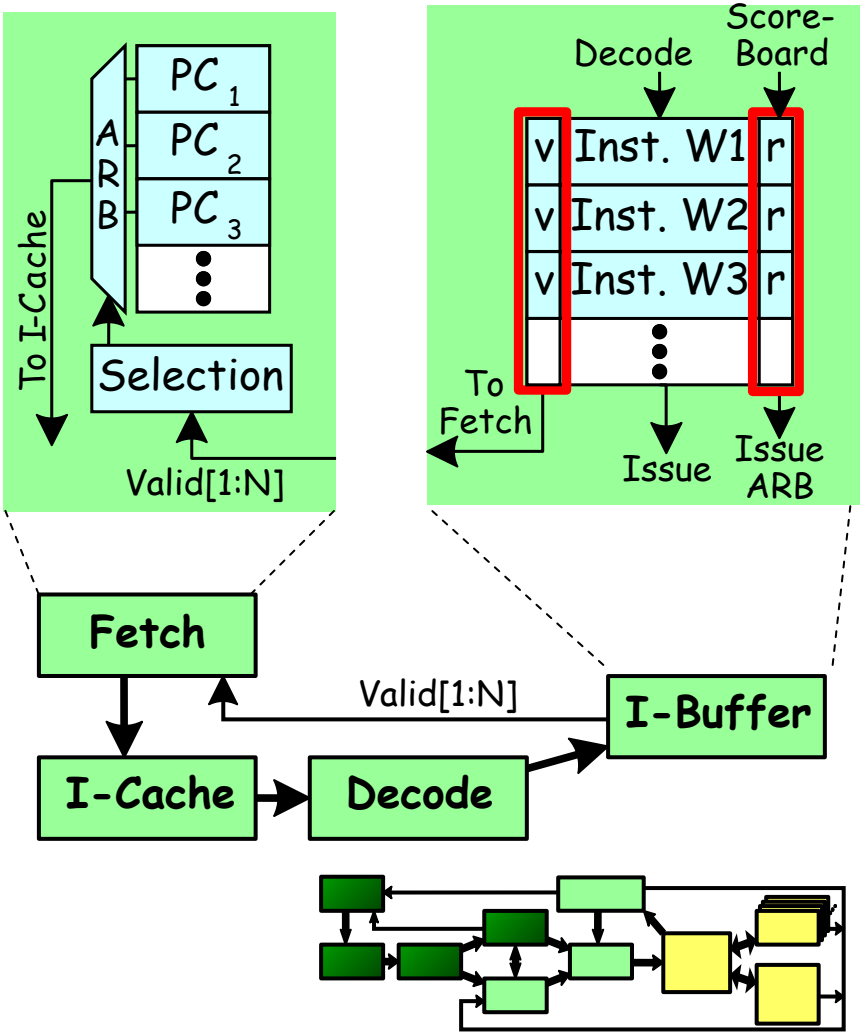


- Goal: maximise throughput
- Three decoupled warp schedulers
- Scoreboard
- Large register file
- Multiple (heterogeneous) SIMD functional units:
  - Special Function Unit (SFU), load/store unit, floating-point function unit, integer function unit, or Tensor Cores (since *Volta*)

25

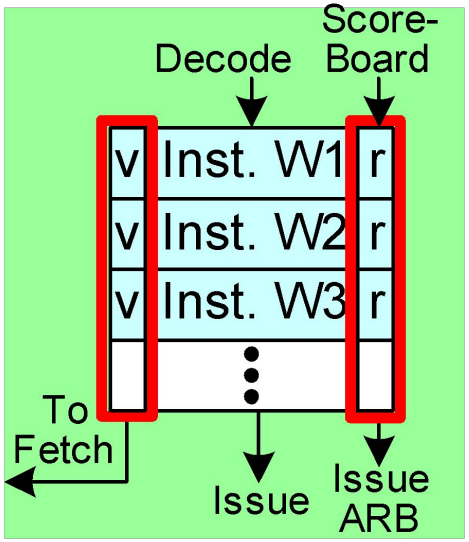
Fetch + Decode

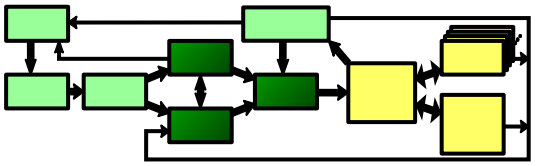
- Arbitrate the I-cache among warps
  - Cache miss handled by fetching again later
- Fetched instruction is decoded and then stored in the I-Buffer
  - 1 or more entries / warp
  - Only warp with vacant entries are considered in fetch



Instruction Issue

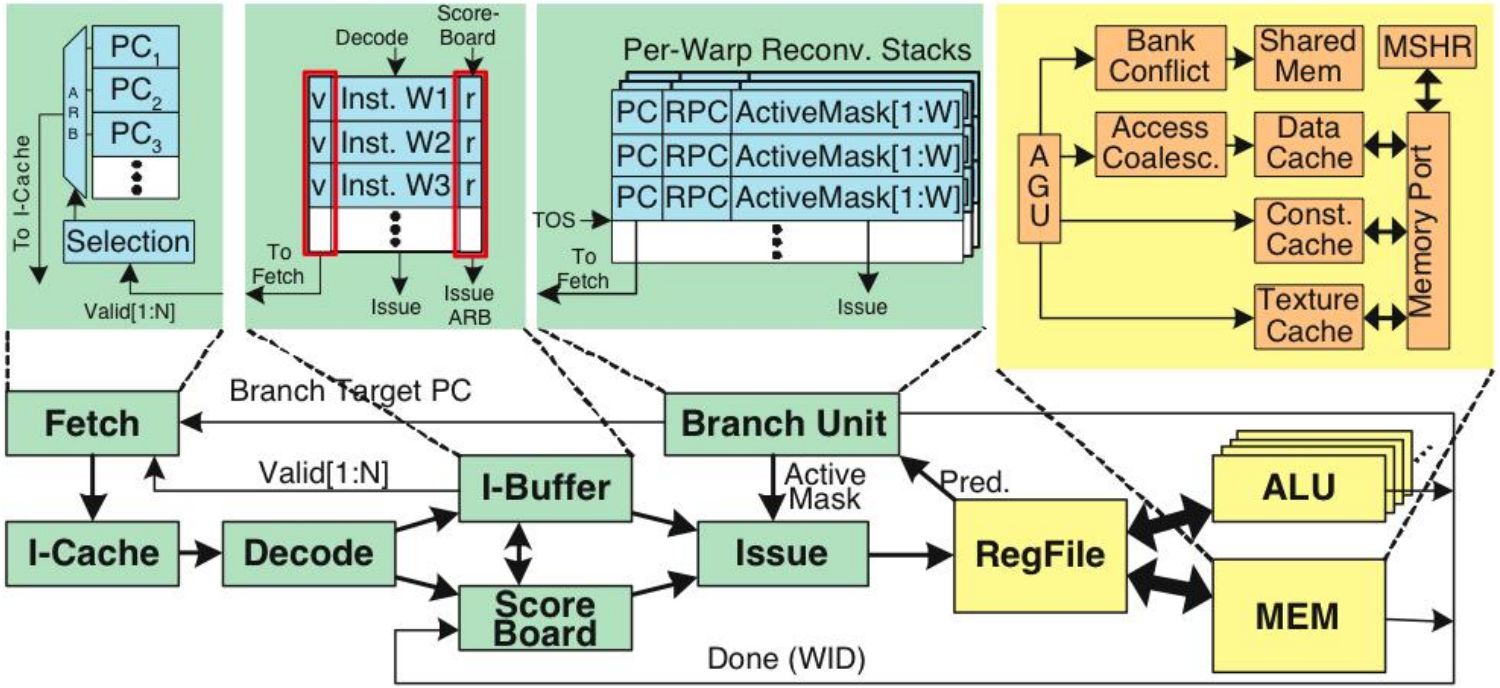
- Select a warp and issue an instruction from its I-Buffer for execution
  - Scheduling: Greedy-Then-Oldest (GTO)
    - Schedule from a single warp until it stalls
    - Then pick the oldest warp
      - Reduces number of warps that core must support to hide long exec latencies by overlapping instructions within same thread.
    - In-order Scoreboard to avoid RAW/WAW/WAR
  - GT200/later Fermi/Kepler: Allow dual issue (superscalar)
  - To avoid stalling pipeline might keep instruction in I-buffer until know it can complete (replay)



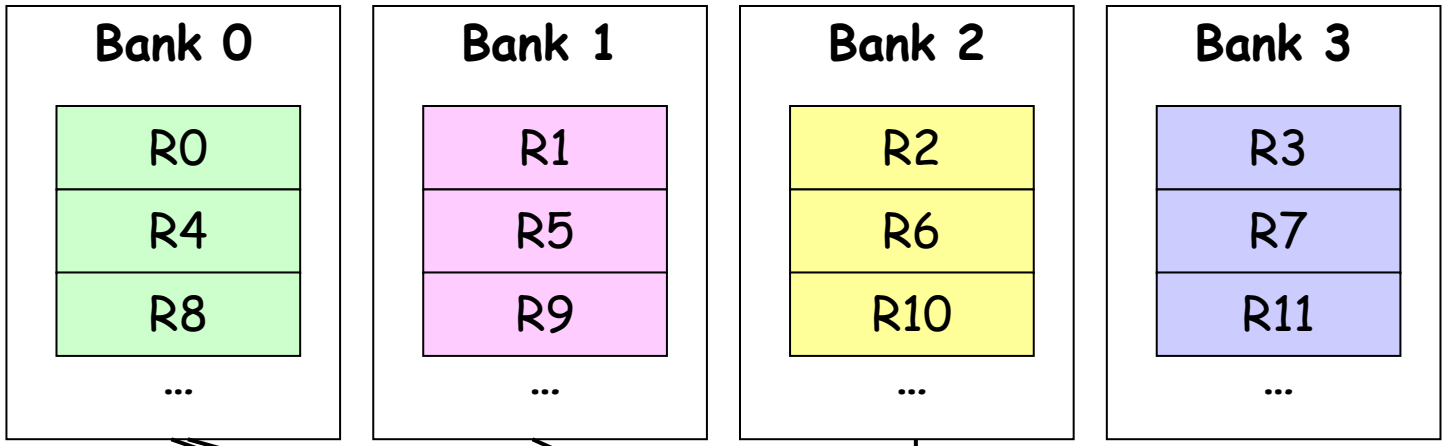


Fetching operands: Register File

- Large number of registers = 256KB on recent GPUs!
- Need “4 ports” (e.g., FMA 3-input, 1-output)
  - Greatly increases area! (area ∝ number of ports)
- Alternative: banked single ported register file. How to avoid bank conflicts?

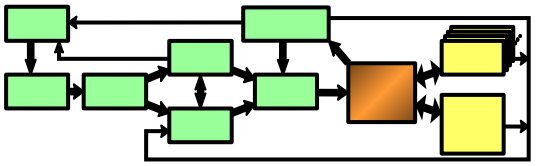


Banked Register File: Conflicts

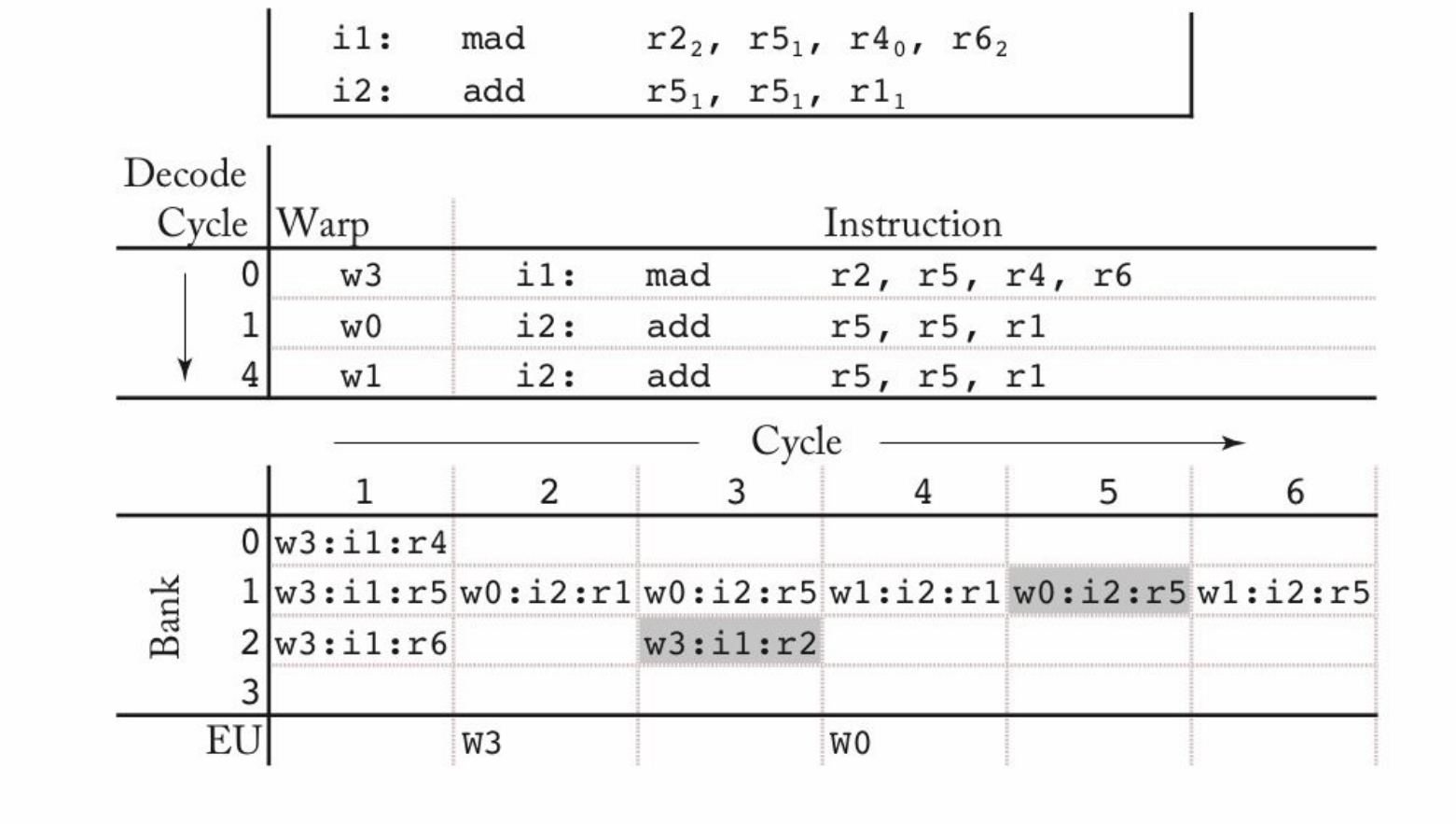



```
add.s32   R3, R1, R2;
```
No Conflict

```
mul.s32   R3, R0, R4;
```
Conflict at bank 0

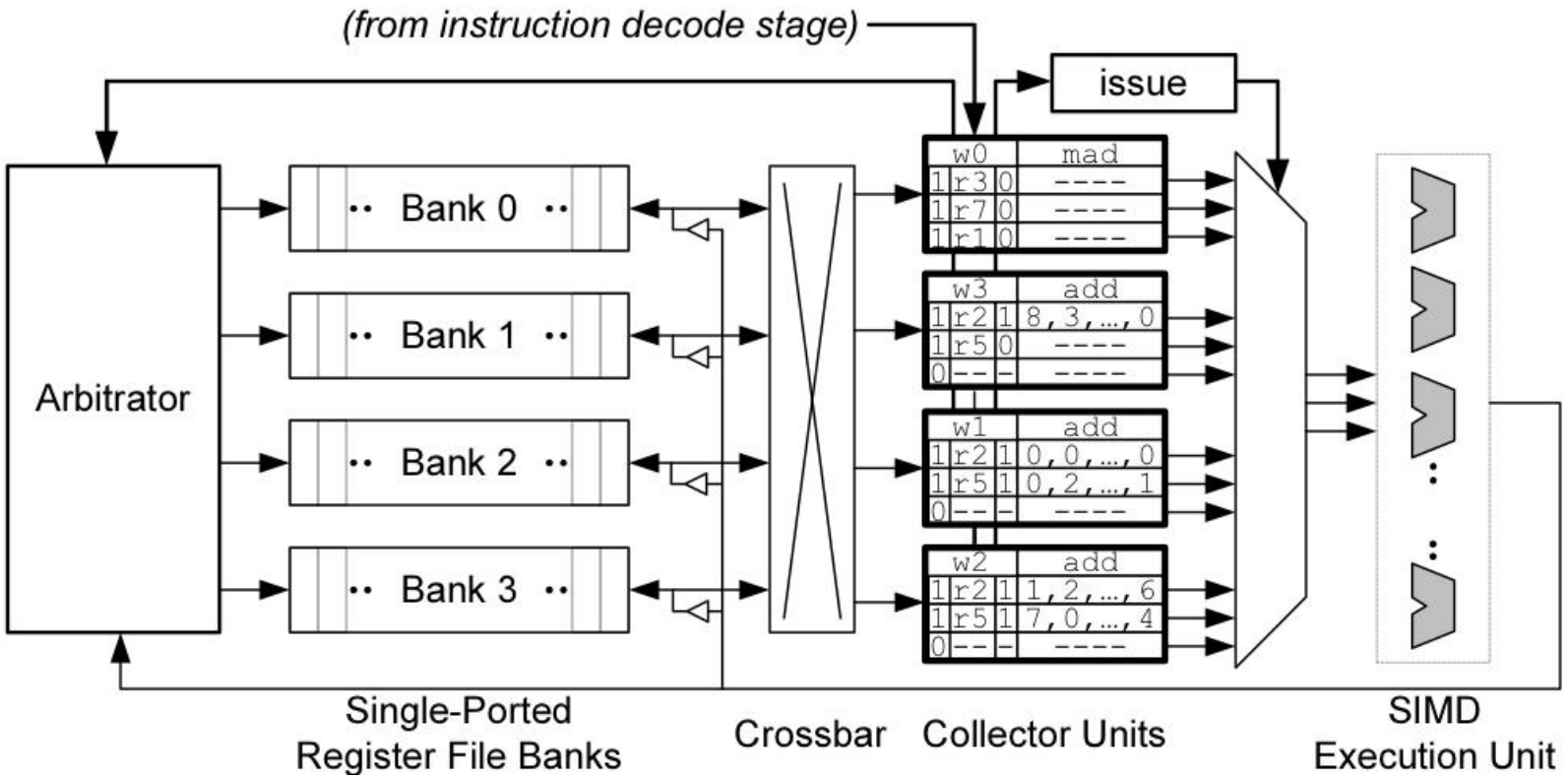


Naive Banked Register File (Example)



- It takes six cycles for three instructions to finish reading their source registers.
- During this time many of the banks are not accessed.

Operand Collector



- Issue instruction to collector unit.
- Collector unit similar to reservation station in Tomasulo's algorithm.
- Stores source register identifiers.
- Arbiter selects operand accesses that do not conflict on a given cycle.
- Arbiter needs to also consider writeback (or need read+write port)

31

Operand Collector (Example)

```
i1:  add   r1, r2, r5
i2:  mad   r4, r3, r7, r1
```

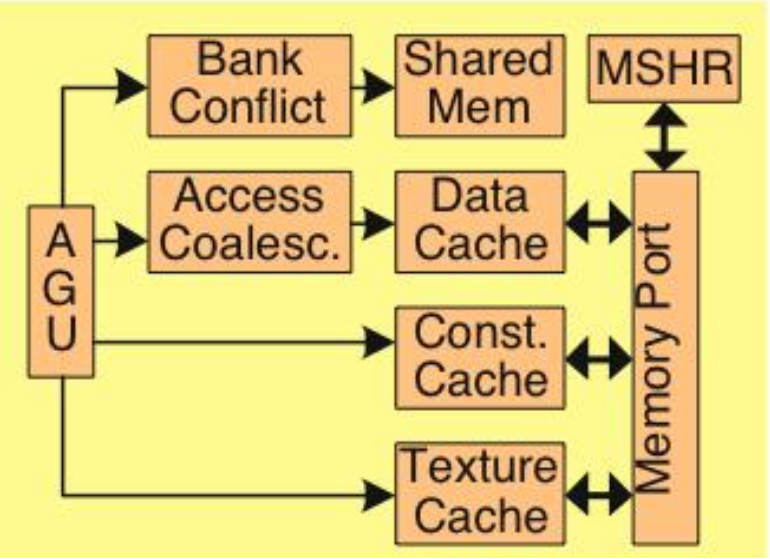
Cycle	Warp	Instruction
0	w1	i1: add $r1_2$, $r2_3$, $r5_2$
1	w2	i1: add $r1_3$, $r2_0$, $r5_3$
2	w3	i1: add $r1_0$, $r2_1$, $r5_0$
3	w0	i2: mad $r4_0$, $r3_3$, $r7_3$, $r1_1$

Cycle →

Bank	1	2	3	4	5	6
0		w2:r2		w3:r5		w3:r1
1			w3:r2			
2		w1:r5		w1:r1		
3	w1:r2		w2:r5	w0:r3	w2:r1	w0:r7
EU			w1	w2	w3	

- Multiple collector units: multiple instructions overlap reading of source operands -> helps improve throughput when there are bank conflicts

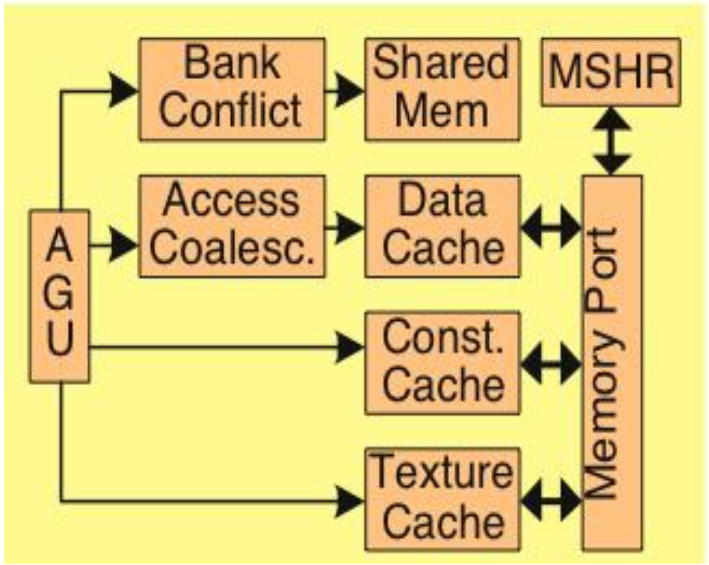
GPU On-chip Memory



Each multiprocessor (SM) has four types of on-chip memory:

1. One set of local 32-bit registers per processor (= **Register File**)
2. A parallel **data cache** or **shared memory** that is shared by all scalar processor cores.
   a. where shared memory space resides (`__shared__`)
3. A **read-only constant cache** that is shared by all scalar processor cores
   a. speeds up reads from constant memory (read-only region of device memory)
4. A **read-only texture cache** that is shared by all scalar processor cores
   a. speeds up reads from texture memory (read-only region of device memory)
   b. SMs access texture cache via the texture unit

GPU External Memory



- Throughput-oriented architecture: requires high BW
- Use high-BW DRAM (such as GDDR or HBM)
- HW combines as many memory address as possible to generate fewer memory transactions (coalescing)

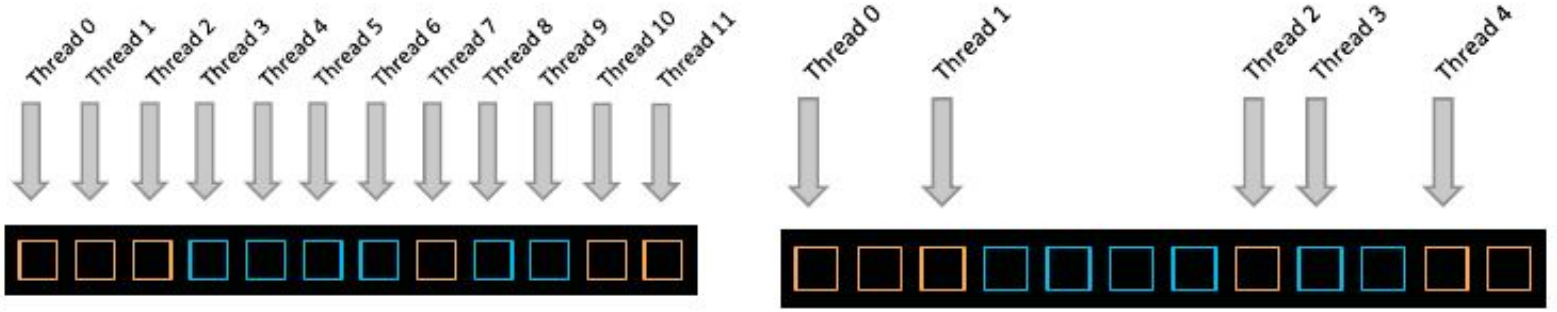


Figure 1.18: Coalesced uncoalesced memory requests. Left: all threads are accessing sequential memory addresses (coalesced); right: threads are accessing non-sequential memory addresses (uncoalesced): courtesy of [72].

Summary

- GPGPU: Motivation
- SIMT Programming Model
- Two-stage compilation
- GPU Microarchitecture
- GPU Memory System