Handout 6 CUDA, GPU, OpenCL Machine

Outline

- CUDA
- OpenCL
- SIMT GPU

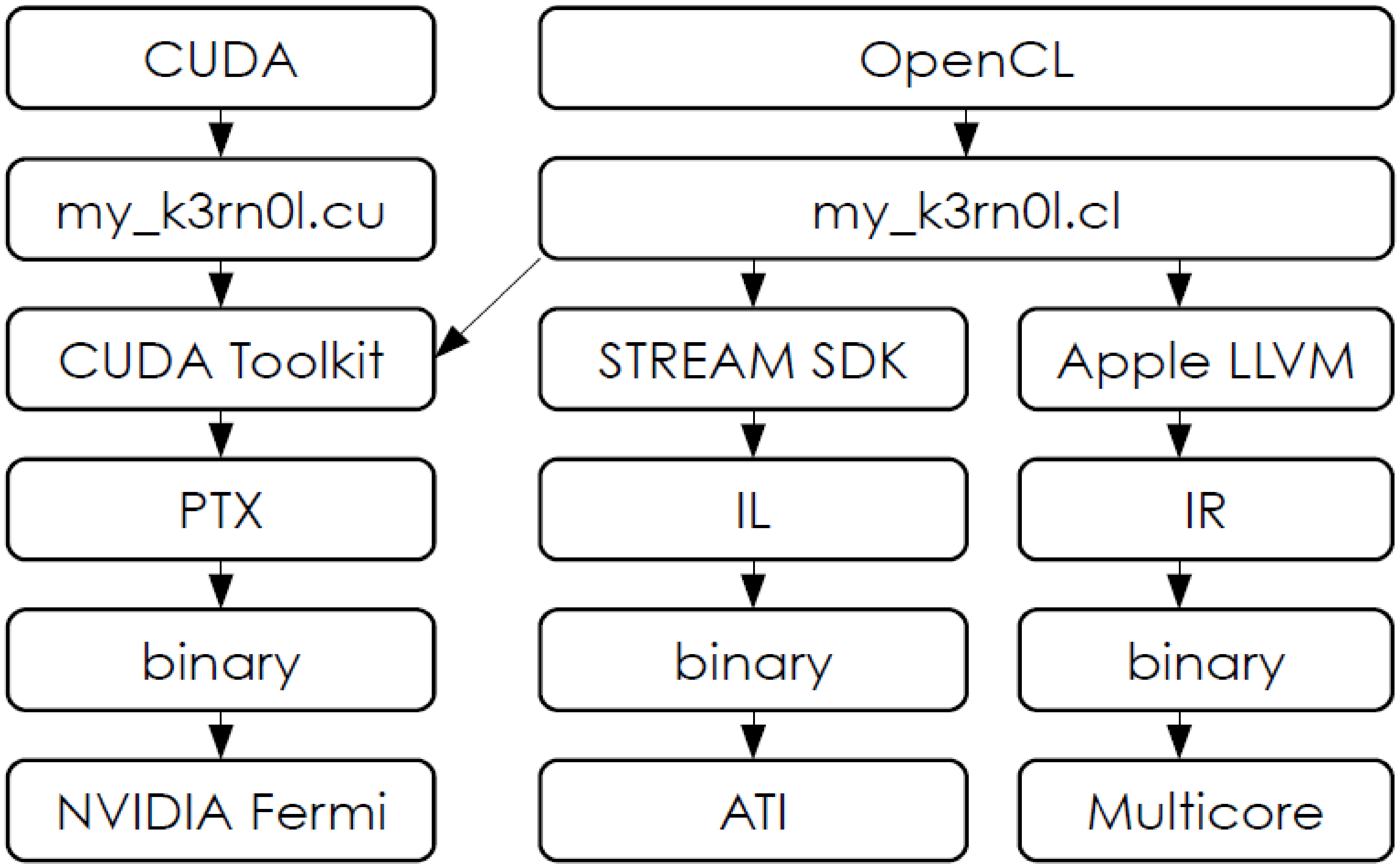
CUDA

- **CUDA**
  - **Compute Unified Device Architecture** (**CUDA**) is a parallel computing architecture developed by Nvidia.
  - **Heterogeneous execution model**
    - » **CPU is the *host*, GPU is the *device***
  - **Develop a C-like programming language for GPU**
  - **Unify all forms of GPU parallelism as *CUDA thread***
  - **Programming model is “Single Instruction Multiple Thread”**

PTX and LLVM

- **PTX**
  - **Parallel Thread Execution (PTX)** is a pseudo-assembly language used in Nvidia's CUDA programming environment. The nvcc compiler translates code written in CUDA, a C-like language, into PTX, and the graphics driver contains a translator which translates the PTX into a binary code which can be run on the processing cores.
- **LLVM**
  - **LLVM** (formerly Low Level Virtual Machine) is compiler infrastructure written in C++; it is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages.
  - **Compile the source code to the intermediate representation(LLVM-IR).**

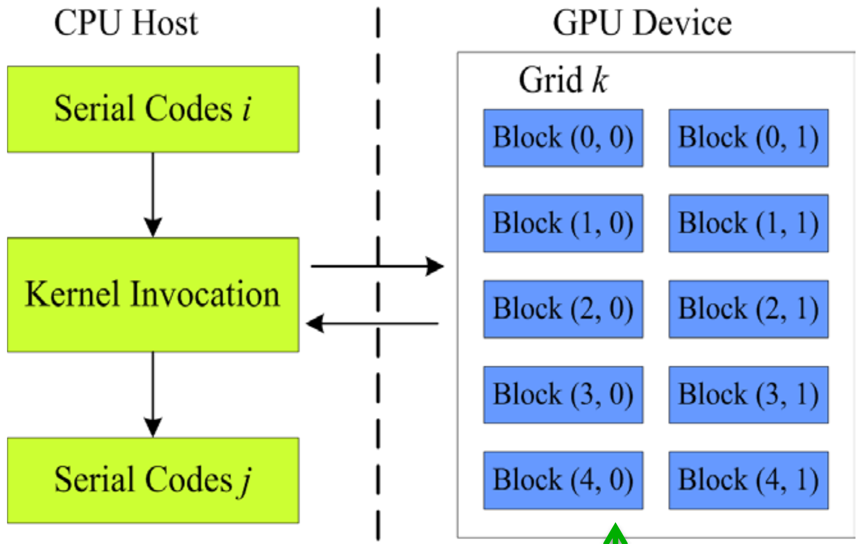
CUDA v.s. OpenCL Platform



GPU

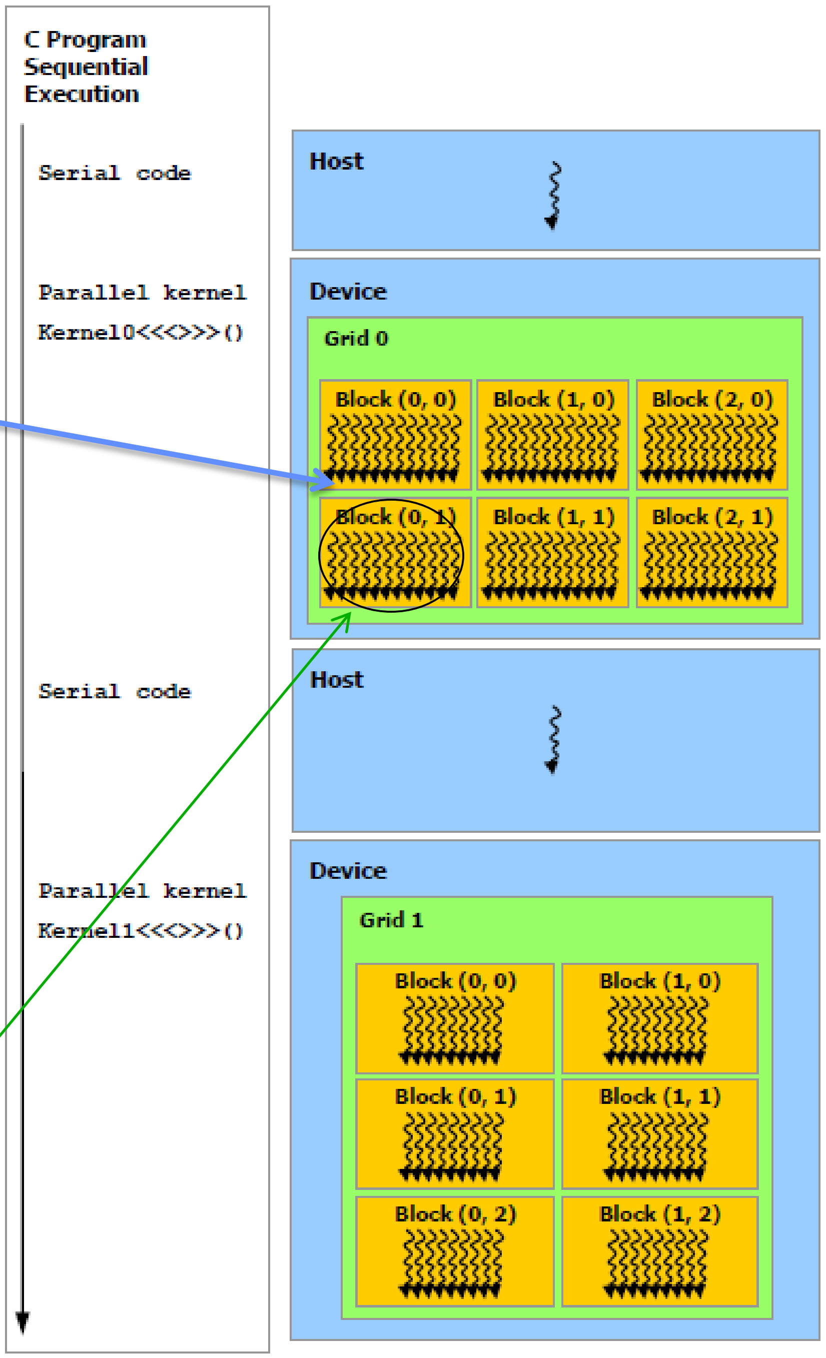
Parallel Kernel

Thread blocks can be executed independently and in any order



2 x 5 blocks

Programmer determines the parallelism by specifying the grid dimensions and the number of threads per SIMD processor



Execution model

Threads and Blocks

- A thread is associated with each data element
- Threads are organized into blocks
  - A thread block is assigned to a processor called multithreaded SIMD processor (or an SM, streaming multiprocessor)
- Blocks are organized into a grid
  - thread blocks can run independently and in any order
- A grid is the code that runs on a GPU that consists of a set of thread blocks.
- GPU hardware handles thread management, not applications or OS

A thread; user defined entity

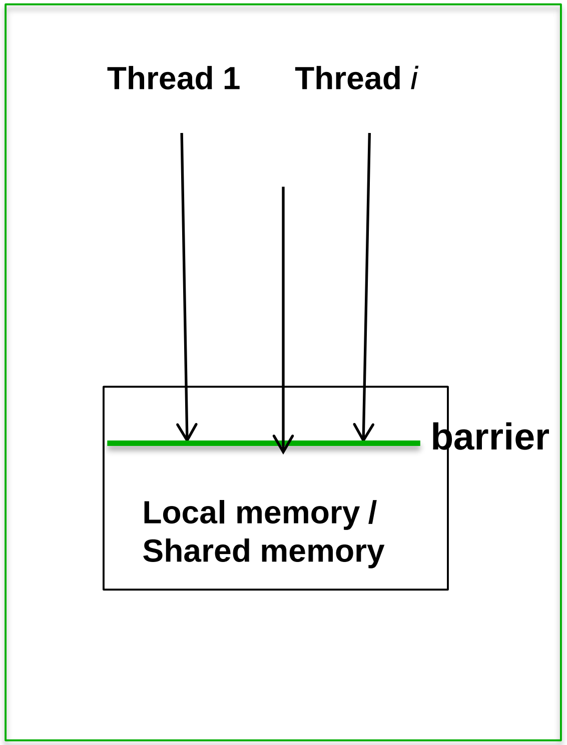
- A thread within a thread block (group) executes an instance of the kernel (code to execute)
  - Has a thread ID in the group
  - Has its program counter
  - Has its registers, per-thread private memory
    - For register spills, procedure call (stack)
  - Can have L1 and L2 cache to cache private memory
  - Map onto a SIMD lane
  - SIMD lanes do not share private memories

A thread is an instance of program code in execution!

Thread is Work Item in OpenCL

A group of threads: thread block

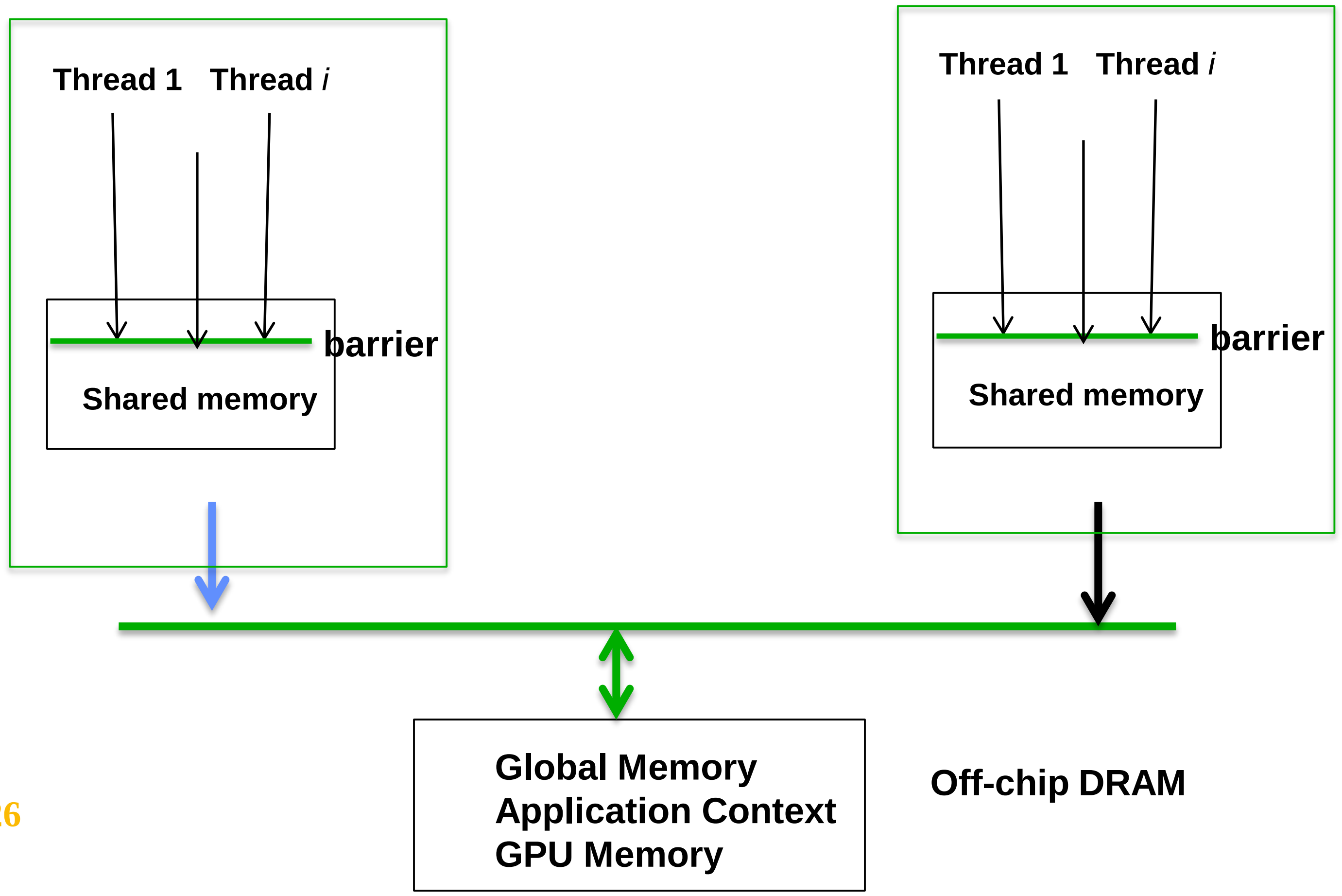
- A thread block: a group of concurrently executing threads within a thread block
  - Has a thread block ID in a grid
  - Synchronization through barrier
  - And communicate through a block level shared memory
    - Inter-thread communication, data sharing, result sharing
  - Map onto a multithread SIMD processor (a block of several SIMD lanes)
  - The SIMD processor dynamically allocates part of the LM to a thread block when it creates the thread block and frees the memory when all the threads in the thread block exit.
  - The local memory is shared by the SIMD lanes within the multithreaded SIMD processor



Thread Block
is Work Group in OpenCL

A group of thread blocks: grid

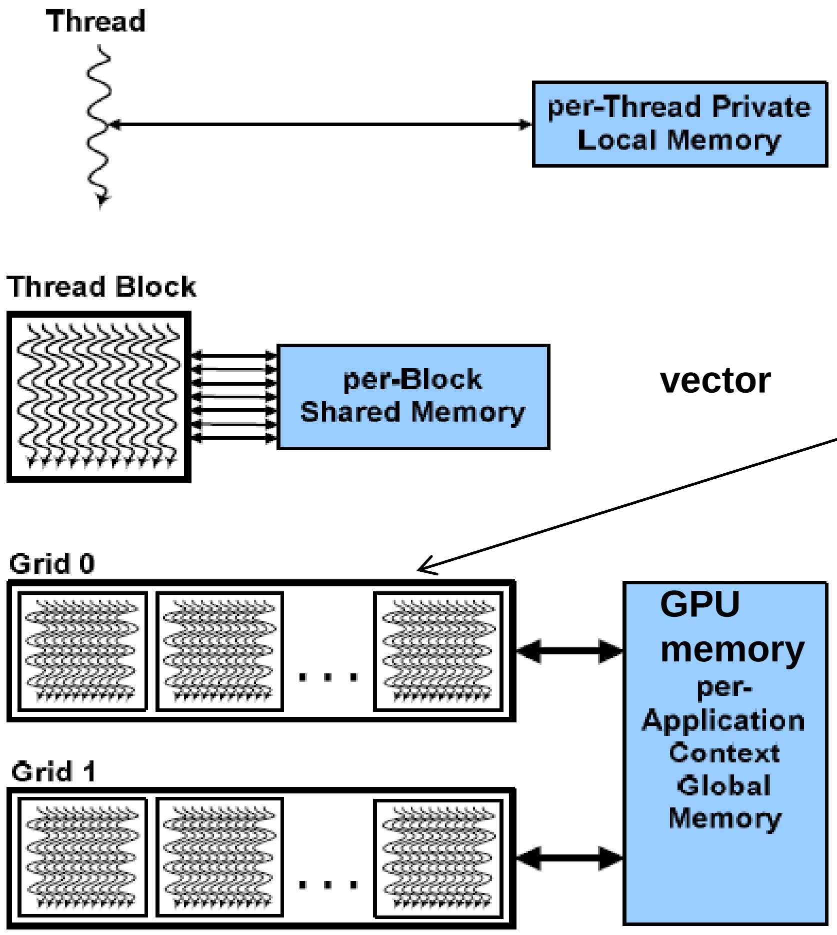
- A thread grid: a group of thread blocks that execute the same kernel, read/write inputs/results from/to global memory, synchronize dependent kernel calls through global memory,



Programmer’s job

- CUDA programmer explicitly specifies the parallelism
  - Set grid dimensions
  - Number of threads per SIMD processors
- One thread works on one element; no need to synchronize among threads when writing results to memory.

CUDA thread hierarchy


Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
Grid 1
. . .
GPU memory
per-Application Context Global Memory

Single element

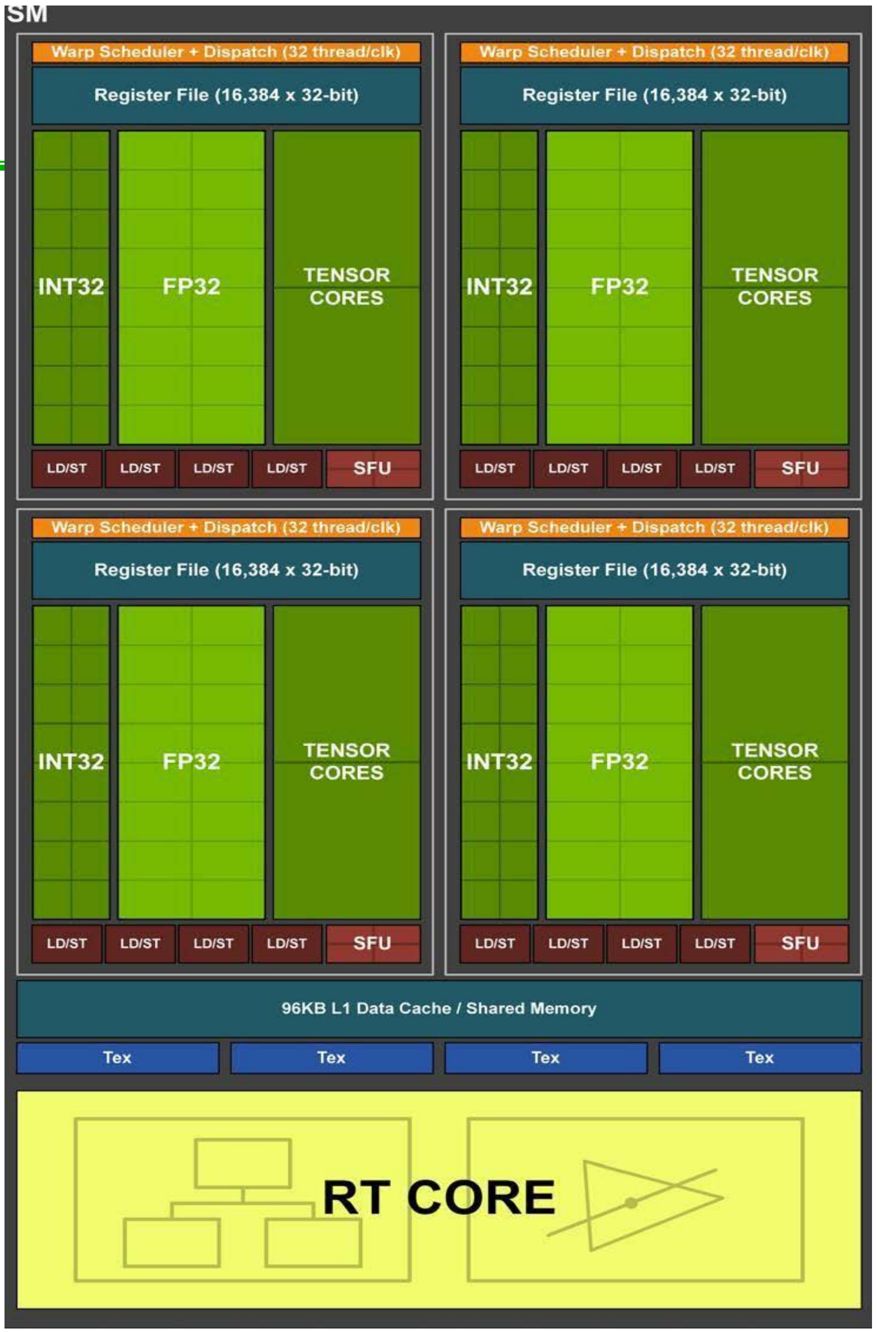
vector

All blocks run the same Kernel code

CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

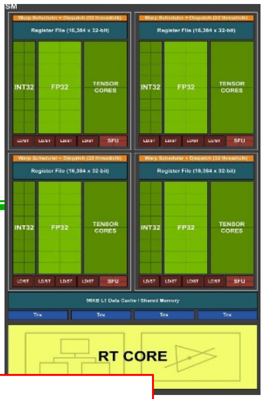
Nvidia Turing-2018

- **Ray Tracing Core**
- **One Turing SM** is partitioned into four processing blocks
  - each with 16 FP32 Cores, 16 INT32 Cores, two Tensor Cores, one warp scheduler, and one dispatch unit.
- **Each block** includes a new L0 instruction cache and a 64 KB register file. The four processing blocks share a combined 96 KB L1 data cache/shared memory.
- **Traditional graphics workloads** partition the 96 KB L1/shared memory as 64 KB of dedicated graphics shader RAM and 32 KB for texture cache and register file spill area.
- **Compute workloads** can divide the 96 KB into 32 KB shared memory and 64 KB L1 cache, or 64 KB shared memory and 32 KB L1 cache.


SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

Nvidia Turing-RTX

- **In a GPU, SM is just a small SM after all.**


SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

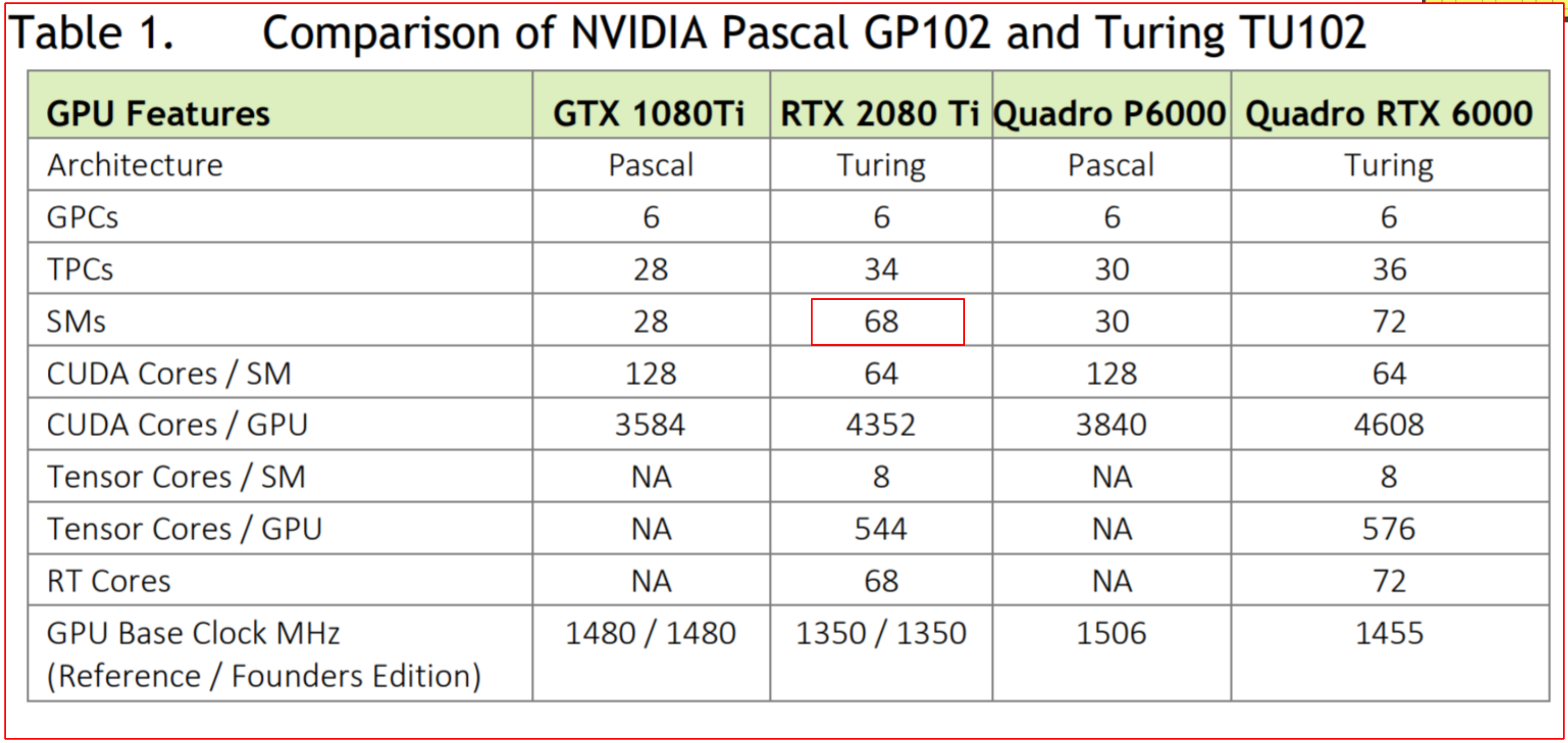


Table 1. Comparison of NVIDIA Pascal GP102 and Turing TU102

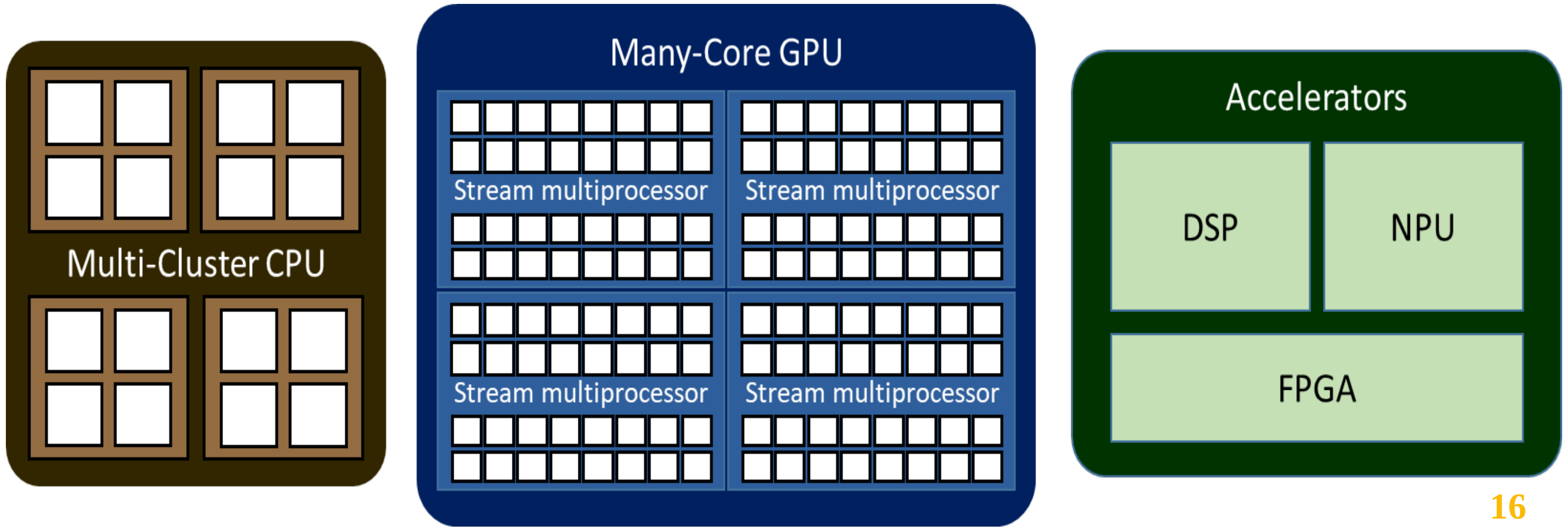
GPU Features	GTX 1080Ti	RTX 2080 Ti	Quadro P6000	Quadro RTX 6000
Architecture	Pascal	Turing	Pascal	Turing
GPCs	6	6	6	6
TPCs	28	34	30	36
SMs	28	68	30	72
CUDA Cores / SM	128	64	128	64
CUDA Cores / GPU	3584	4352	3840	4608
Tensor Cores / SM	NA	8	NA	8
Tensor Cores / GPU	NA	544	NA	576
RT Cores	NA	68	NA	72
GPU Base Clock MHz (Reference / Founders Edition)	1480 / 1480	1350 / 1350	1506	1455

OpenCL

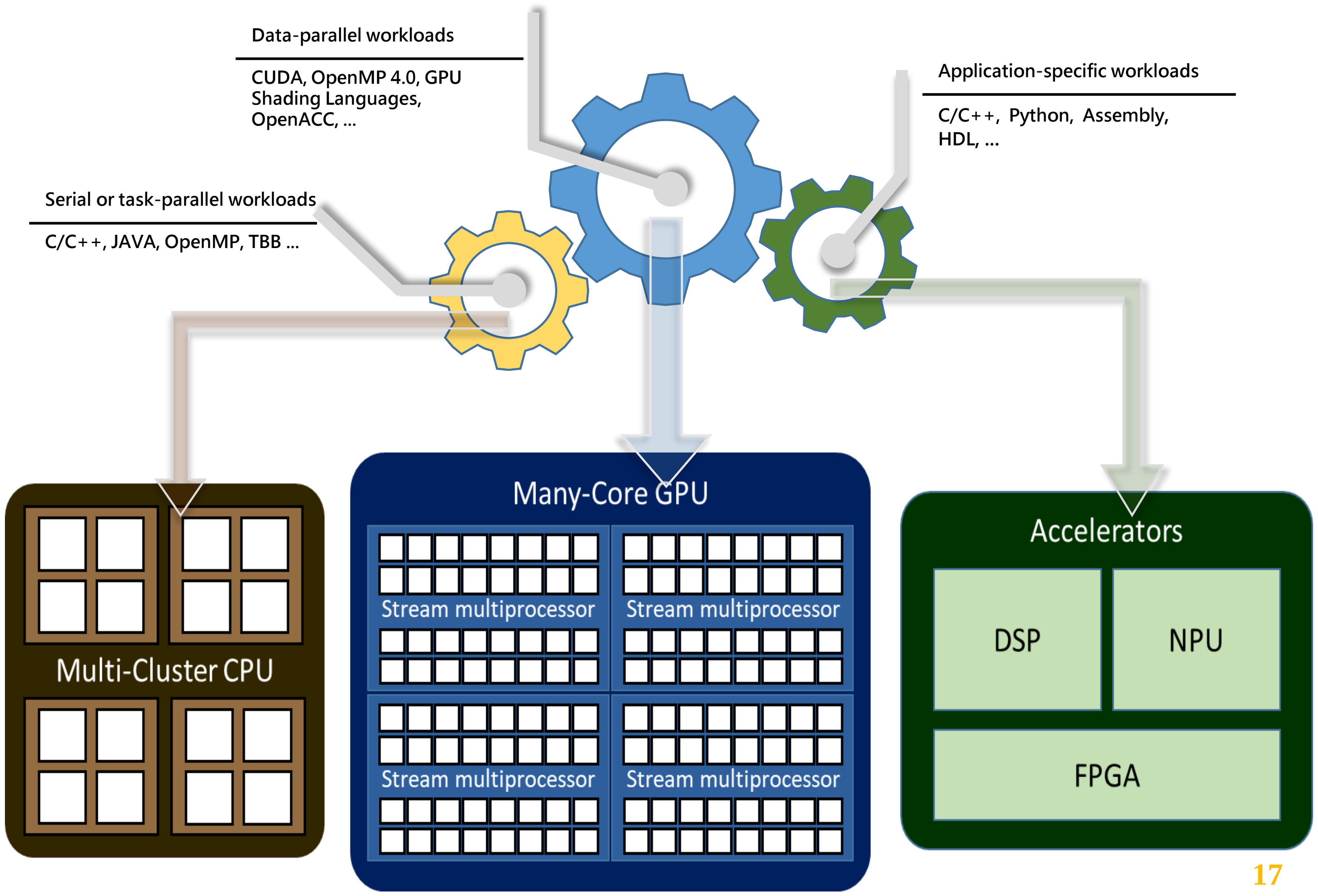
- **OpenCL**
  - **Open Computing Language** (**OpenCL**) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing unit (CPUs), graphics processing unit (GPUs), and other processors.
  - OpenCL includes a language for writing *kernels* (functions that execute on OpenCL devices), plus application programming interfaces (APIs) that are used to define and then control the platforms.
  - OpenCL provides parallel computing using task-based and data-based parallelism. OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group. It has been adopted by Intel, Advanced Micro Devices, Nvidia, and ARM Holdings.

OpenCL for Heterogeneous Computing

- Hardware heterogeneity
  - Multi-Core CPU
    - » MIMD machines, serial or task-parallel workloads
  - Many-Core GPGPU
    - » SIMT machines, data-parallel workloads
  - DSP, NPU, FPGA, ...
    - » Specific accelerators (VLIW, Data-reuse, Reconfiguration, or other domain-specific accelerations)

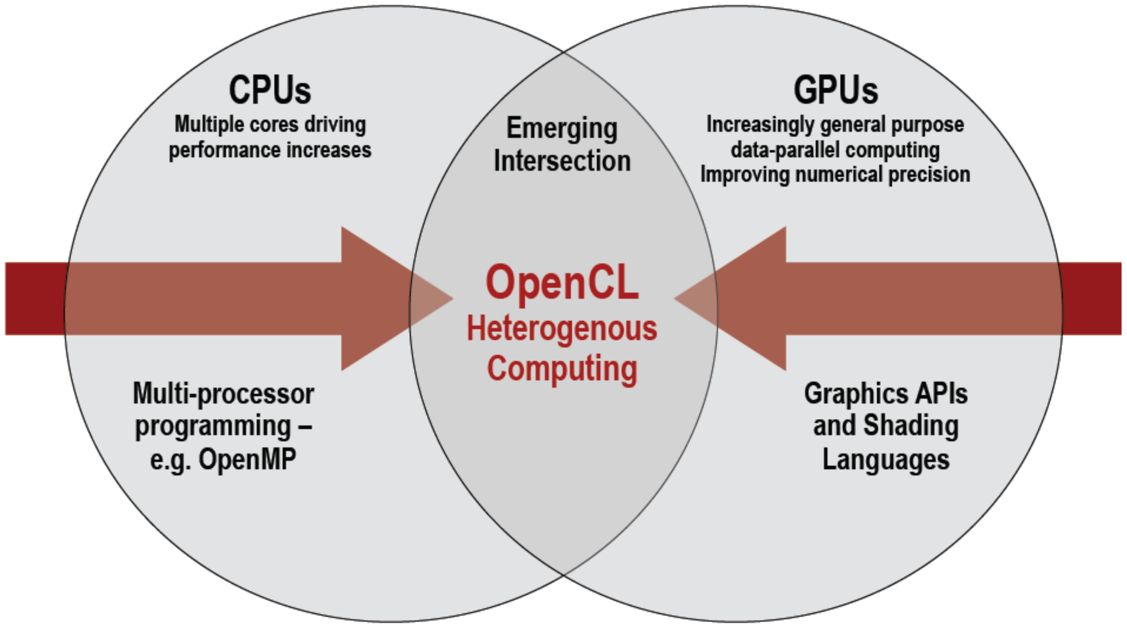


Software Diversity


Data-parallel workloads
CUDA, OpenMP 4.0, GPU Shading Languages, OpenACC, ...
Application-specific workloads
C/C++, Python, Assembly, HDL, ...
Serial or task-parallel workloads
C/C++, JAVA, OpenMP, TBB ...
Multi-Cluster CPU
Many-Core GPU
Stream multiprocessor
Stream multiprocessor
Stream multiprocessor
Stream multiprocessor
Accelerators
DSP
NPU
FPGA

Open Computing Language

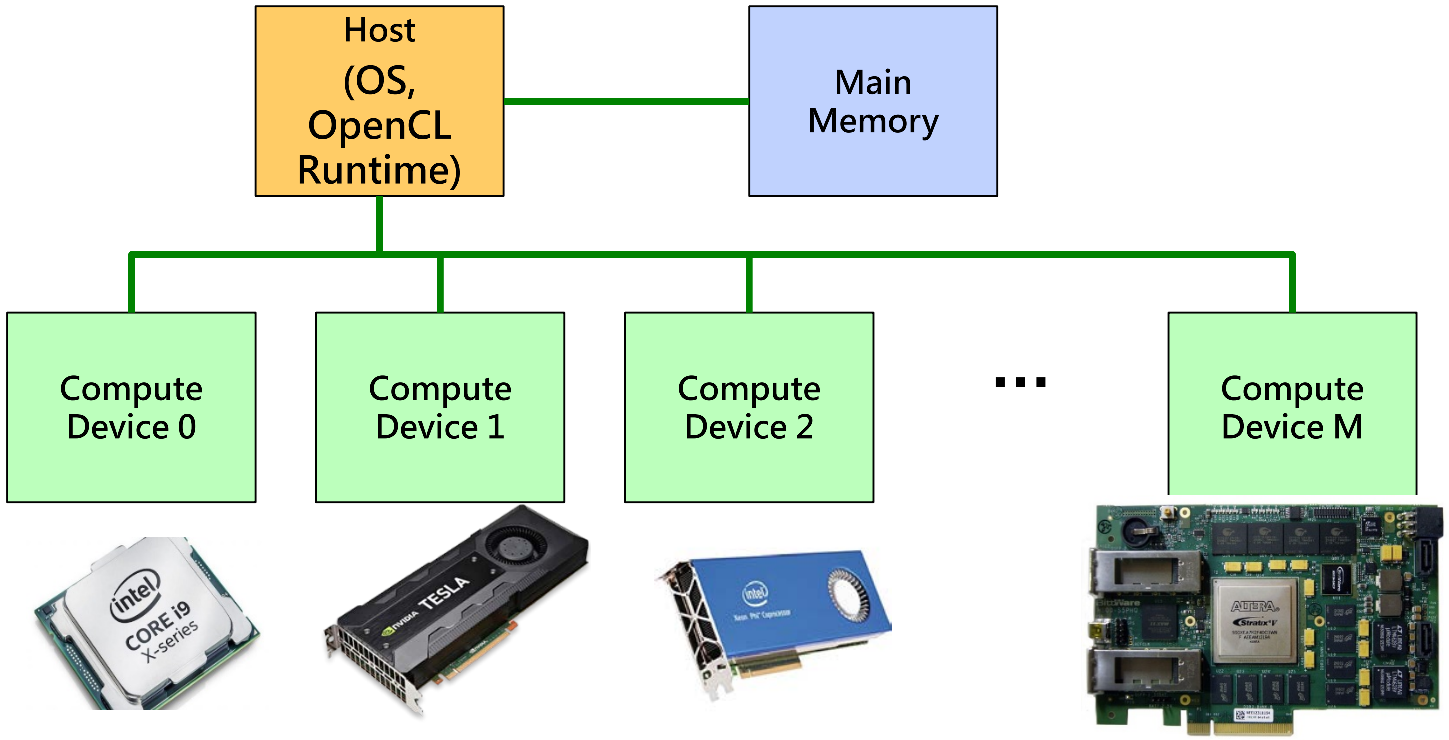
- Program Portability
  - OpenCL is a framework for building parallel applications that are portable across heterogeneous platforms.


CPUs
Multiple cores driving performance increases
Emerging Intersection
GPUs
Increasingly general purpose data-parallel computing
Improving numerical precision
OpenCL
Heterogenous Computing
Multi-processor programming – e.g. OpenMP
Graphics APIs and Shading Languages

OpenCL – Open Computing Language
Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

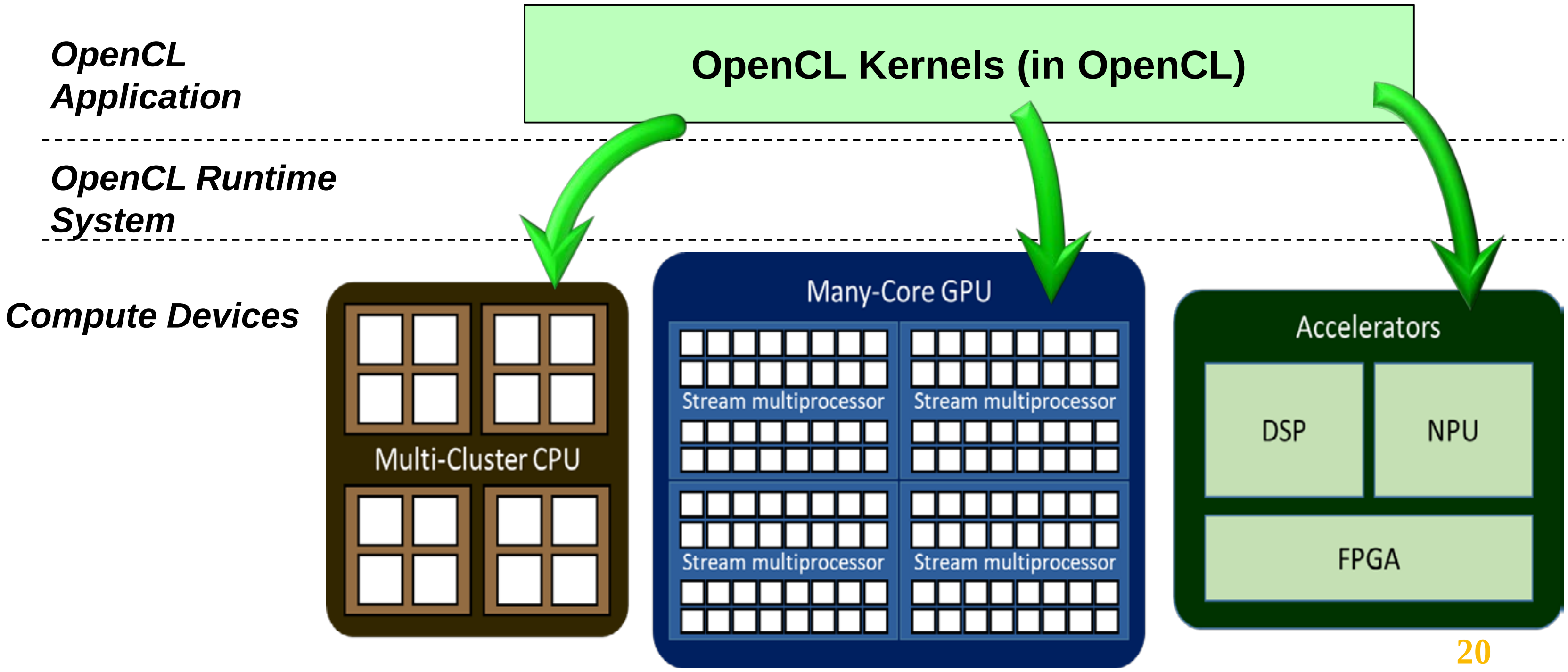
Methodology for Portability-1

- Unified Programming Interface
  - Abstracted OpenCL platform model
    - » A host connected to multiple compute devices with open CL device model
  - Runtime kernel source compilation



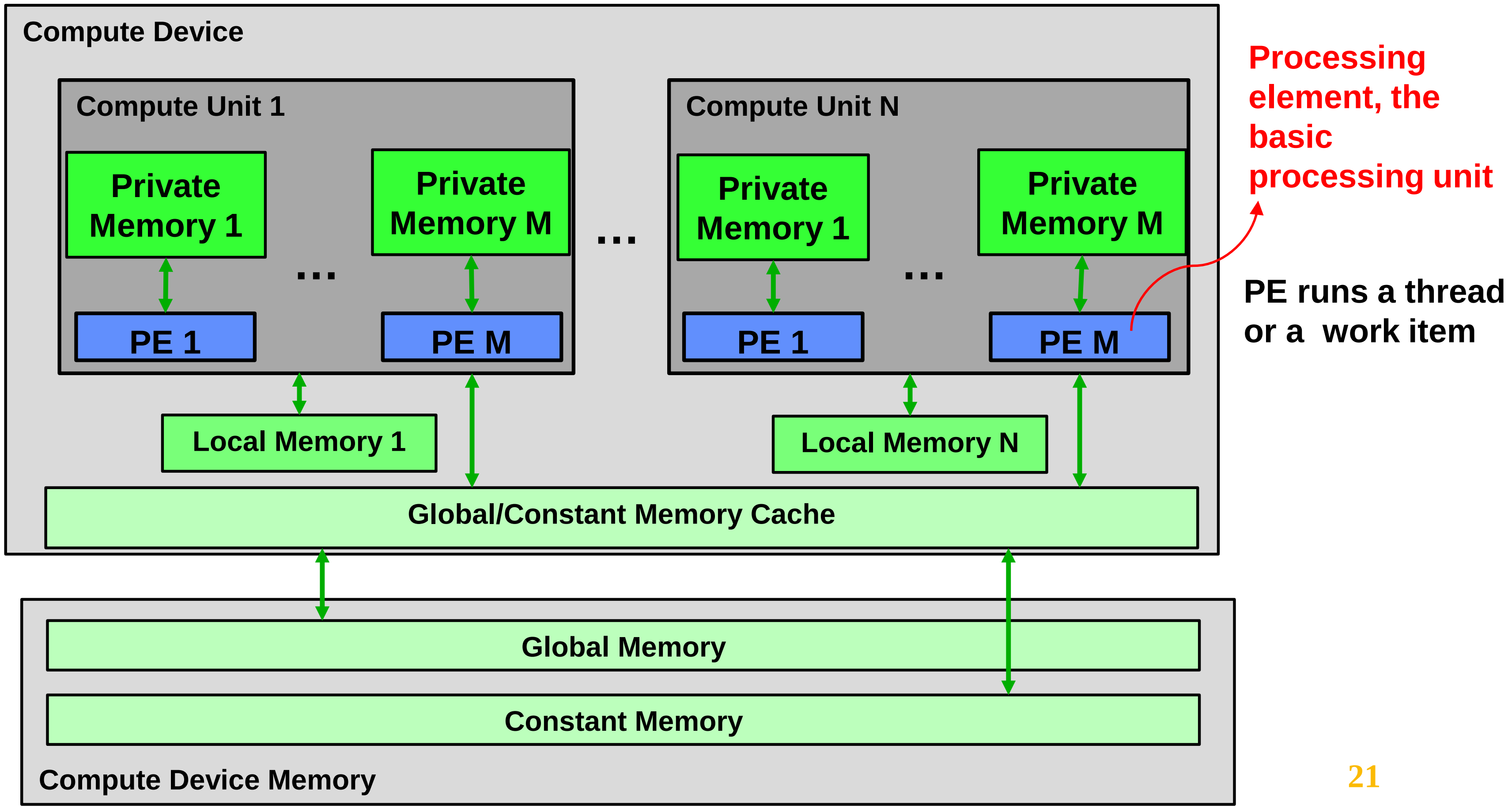
Methodology for Portability-2

- **Unified Programming Interface**
  - **Abstracted OpenCL platform model**
    - » **A host connected to multiple compute devices**
  - **Runtime kernel source compilation**



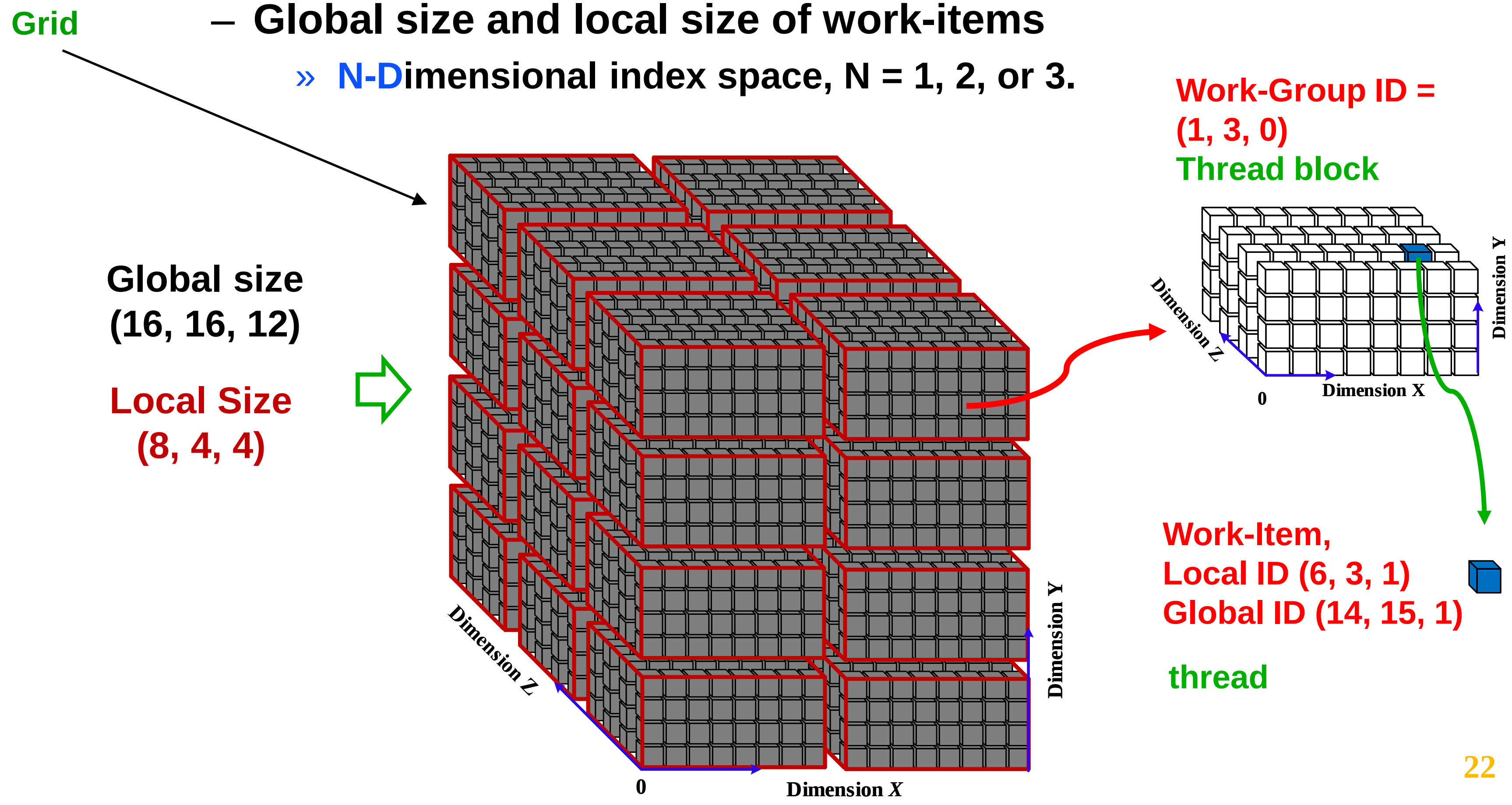
OpenCL Compute Device

- **Abstracted Hierarchical System**
  - Both the compute hierarchy and memory hierarchy



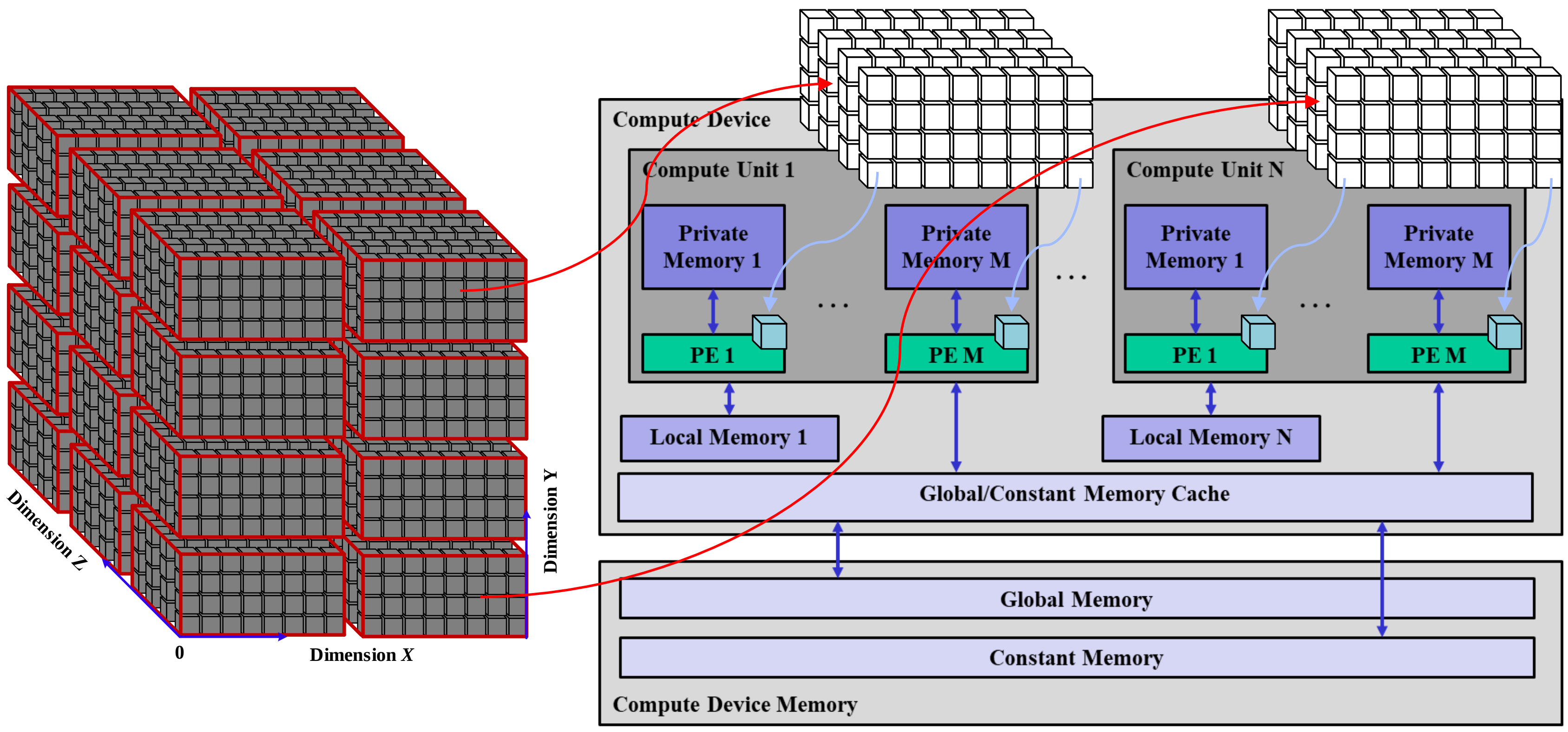
Programming Model: NDRange Index Space

- Work-item
  - A thread
- Workload Hierarchy
  - Global size and local size of work-items
    - » N-Dimensional index space, N = 1, 2, or 3.



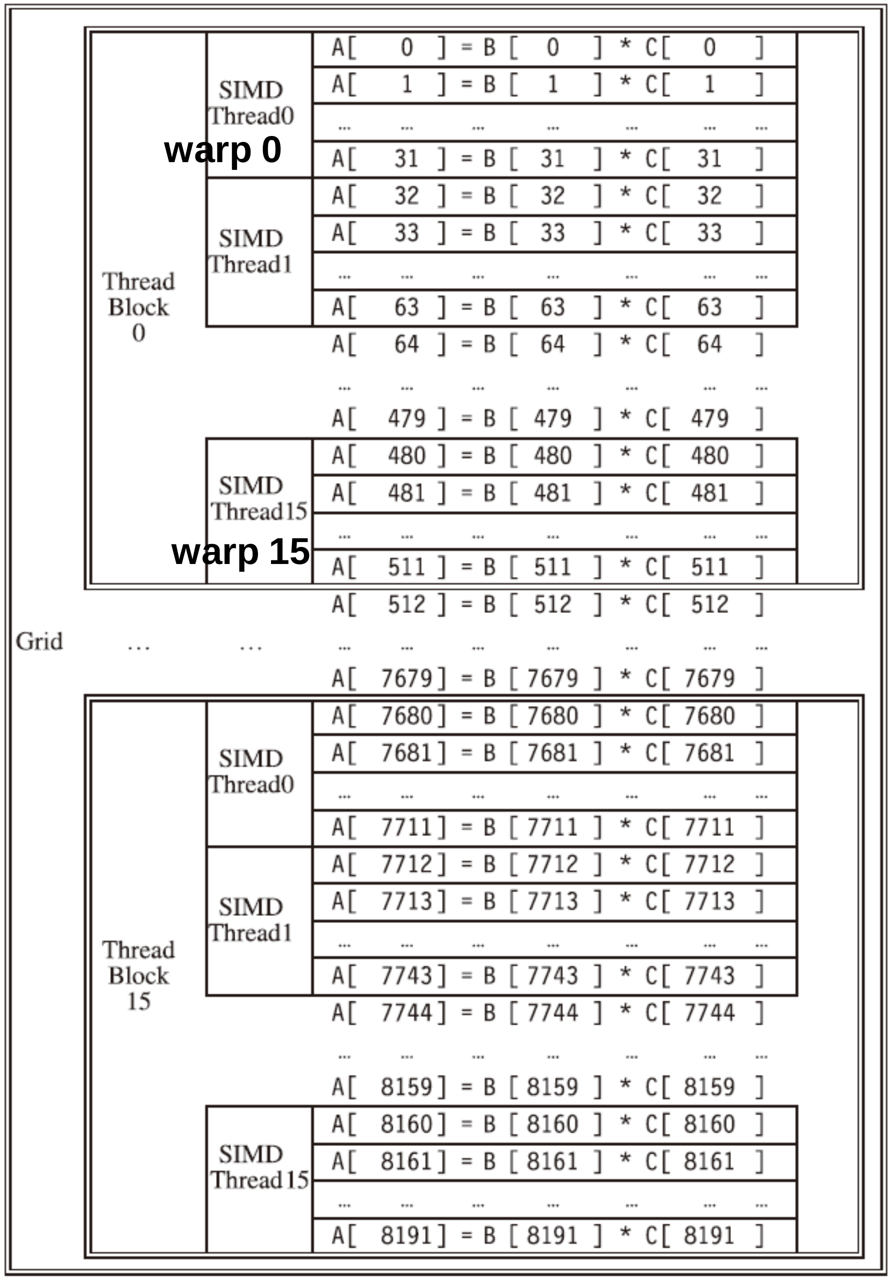
NDRange Workload on Compute Device

- **NDRange workload to a compute device**
  - **A work-group to a compute unit (Synchronization unit)**
  - **A work-item to a processing element**



Example of vector multiply

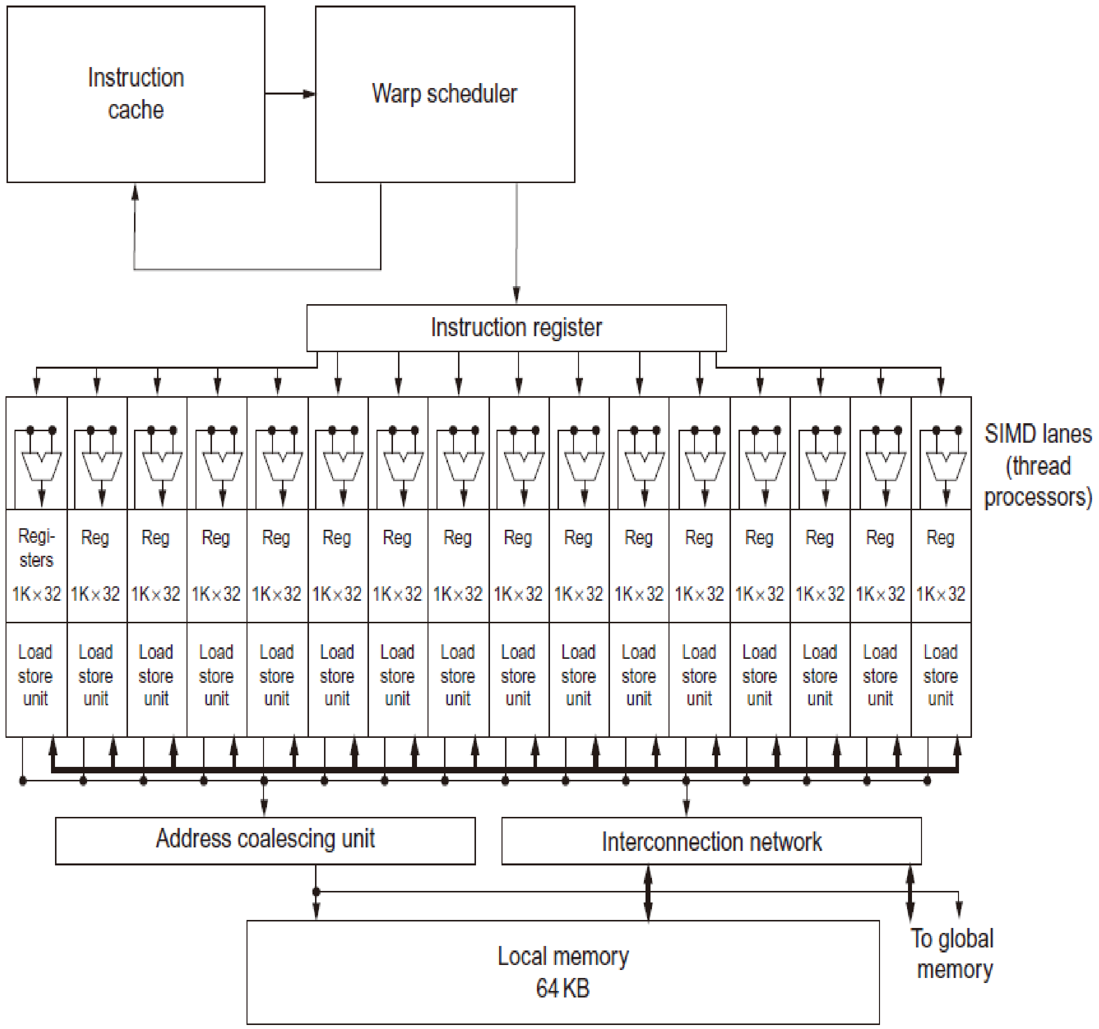
- Vectorizable loop
- Multiply two vectors: A = B x C, each 8192 elements
- Grid is the GPU code that works on all 8192 element multiply.
- A thread block, codes that do 512 element multiply, runs on a multithreaded SIMD processor or an SM
  - Hence 8192/512 = 16 thread blocks
- Instructions in a warp (a SIMD thread here) execute 32 elements at a time.
- Warp size is determined by hardware implementation



Grid
Thread Block 0
SIMD Thread0
warp 0
SIMD Thread1
SIMD Thread15
warp 15
Thread Block 15
A[0] = B [0] * C[0]
A[1] = B [1] * C[1]
A[31] = B [31] * C[31]
A[32] = B [32] * C[32]
A[33] = B [33] * C[33]
A[63] = B [63] * C[63]
A[64] = B [64] * C[64]
A[479] = B [479] * C[479]
A[480] = B [480] * C[480]
A[481] = B [481] * C[481]
A[511] = B [511] * C[511]
A[512] = B [512] * C[512]
A[7679] = B [7679] * C[7679]
A[7680] = B [7680] * C[7680]
A[7681] = B [7681] * C[7681]
A[7711] = B [7711] * C[7711]
A[7712] = B [7712] * C[7712]
A[7713] = B [7713] * C[7713]
A[7743] = B [7743] * C[7743]
A[7744] = B [7744] * C[7744]
A[8159] = B [8159] * C[8159]
A[8160] = B [8160] * C[8160]
A[8161] = B [8161] * C[8161]
A[8191] = B [8191] * C[8191]

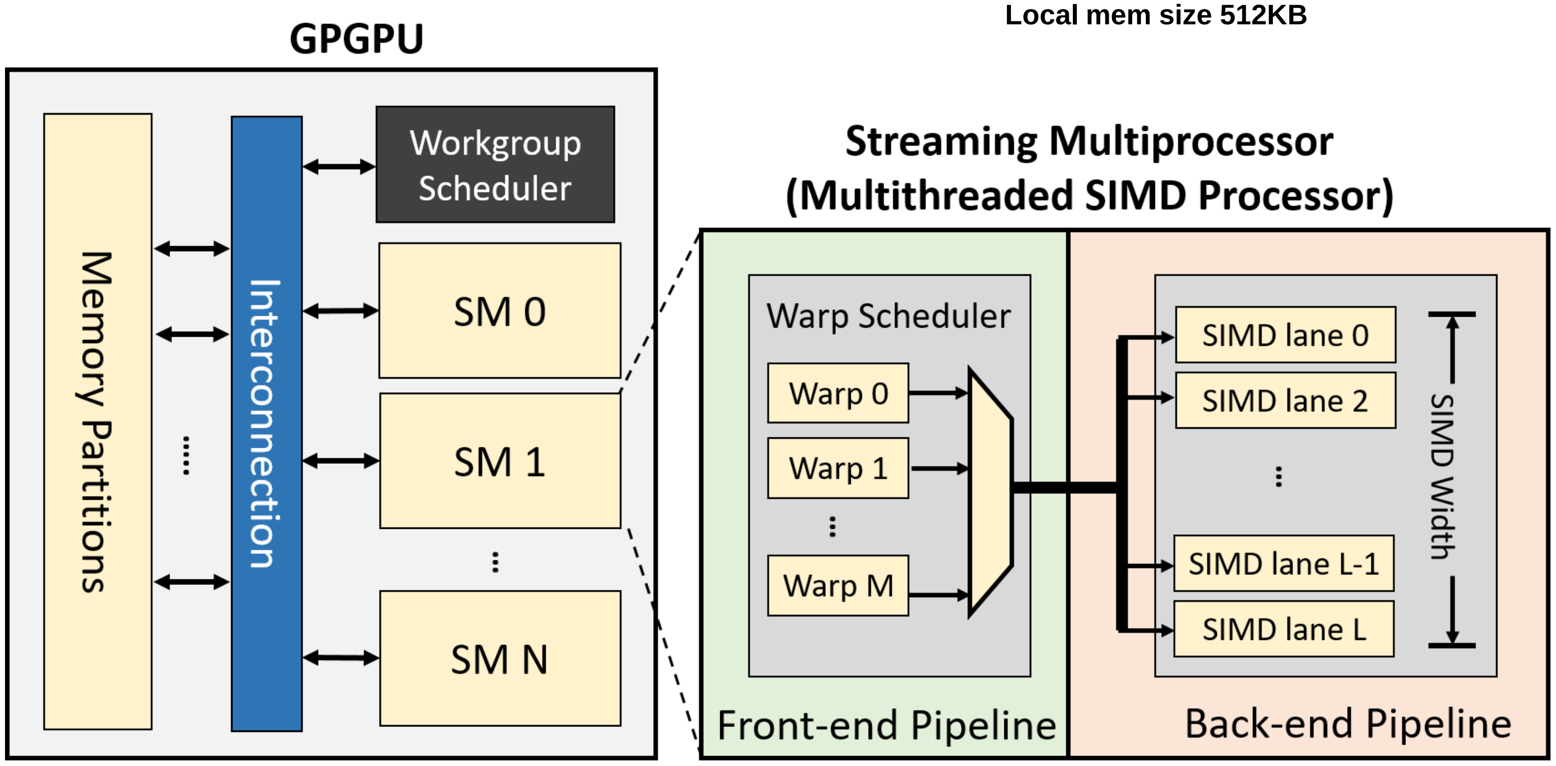
A multithreaded SIMD processor (SM)

- This is a multithreaded SIMD processor which runs a thread block
- Thread block scheduler
  - Determine the # of thread blocks required for the task
  - and keep allocating them to different SIMD processors until the Grid is completed.
- Warp scheduler, i.e., thread scheduler
  - Inside the SIMD processor, schedules instructions from ready-to-run warp



Example: Architecture of CASLab GPU

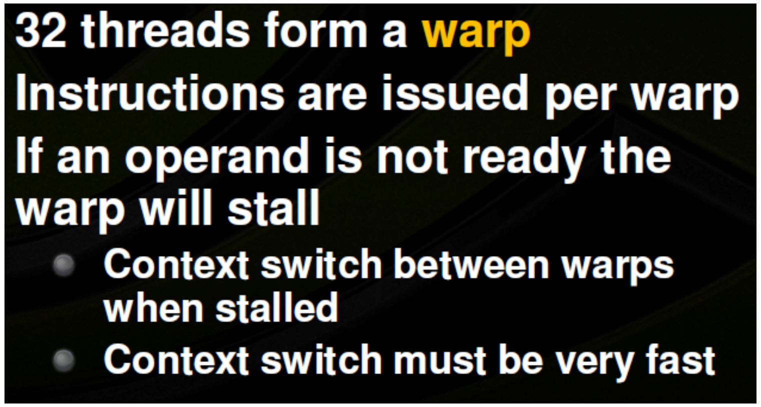
# of SM = 4
# of Warps/SM = 48
# of Threads/Warp = 32
Icache size = 16KB
Dcache size = 16KB
Local mem size 512KB



Warp scheduler in Fermi

A warp is simply an instruction stream of SIMD instructions.

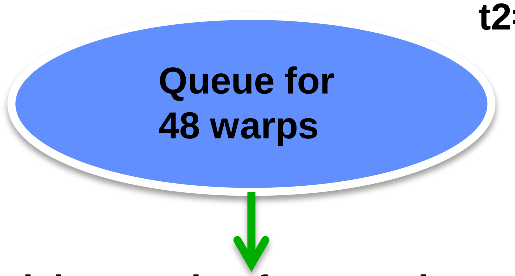
32 threads (on 32 SIMD lanes) have only one instruction stream due to SIMT!



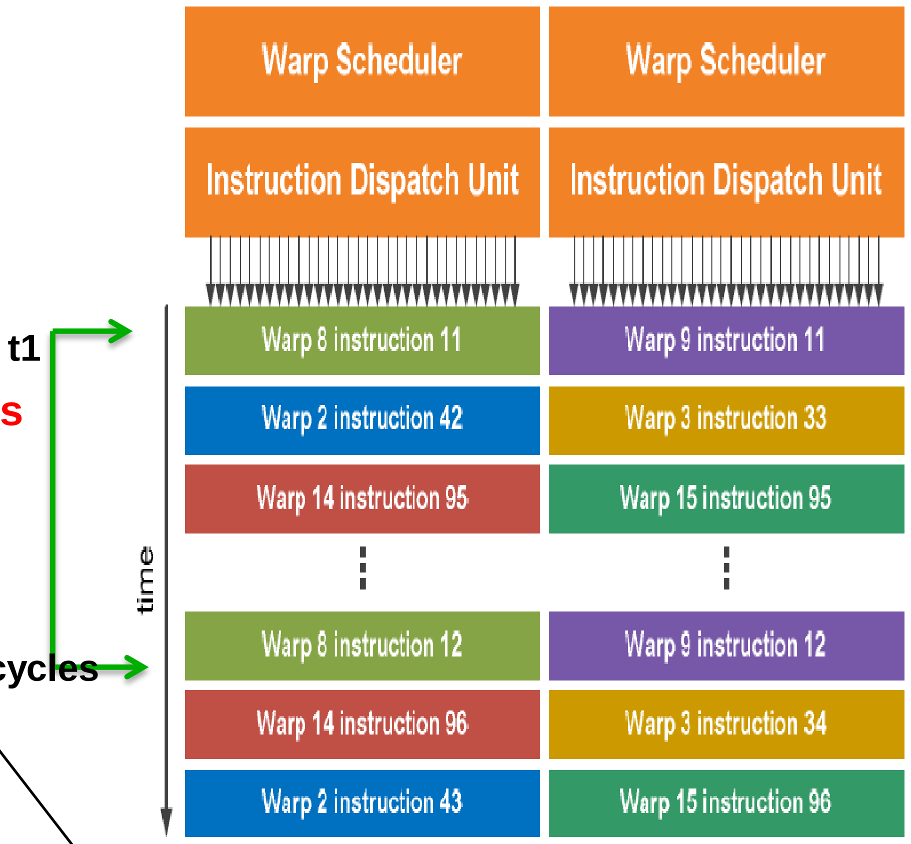
32 threads form a warp
Instructions are issued per warp
If an operand is not ready the warp will stall
- Context switch between warps when stalled
- Context switch must be very fast

- Fermi can have up to 48 active warps per SM => meaning 48 instruction streams.

48 x 32 = 1536 threads/SM



Pick instruction from ready warps

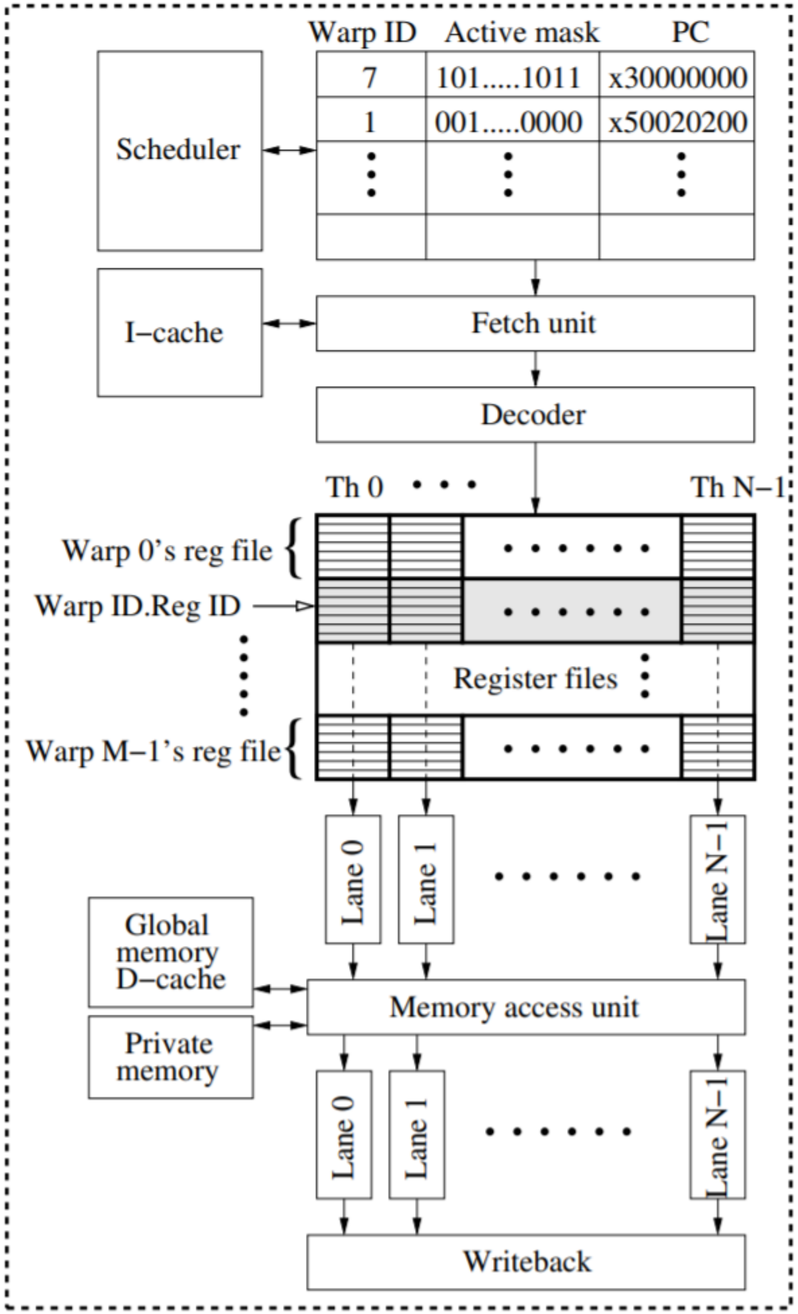


t2= t1 + xx cycles

t2 is the cycle for issuing the next instru. from Warp 8

Warp Scheduler

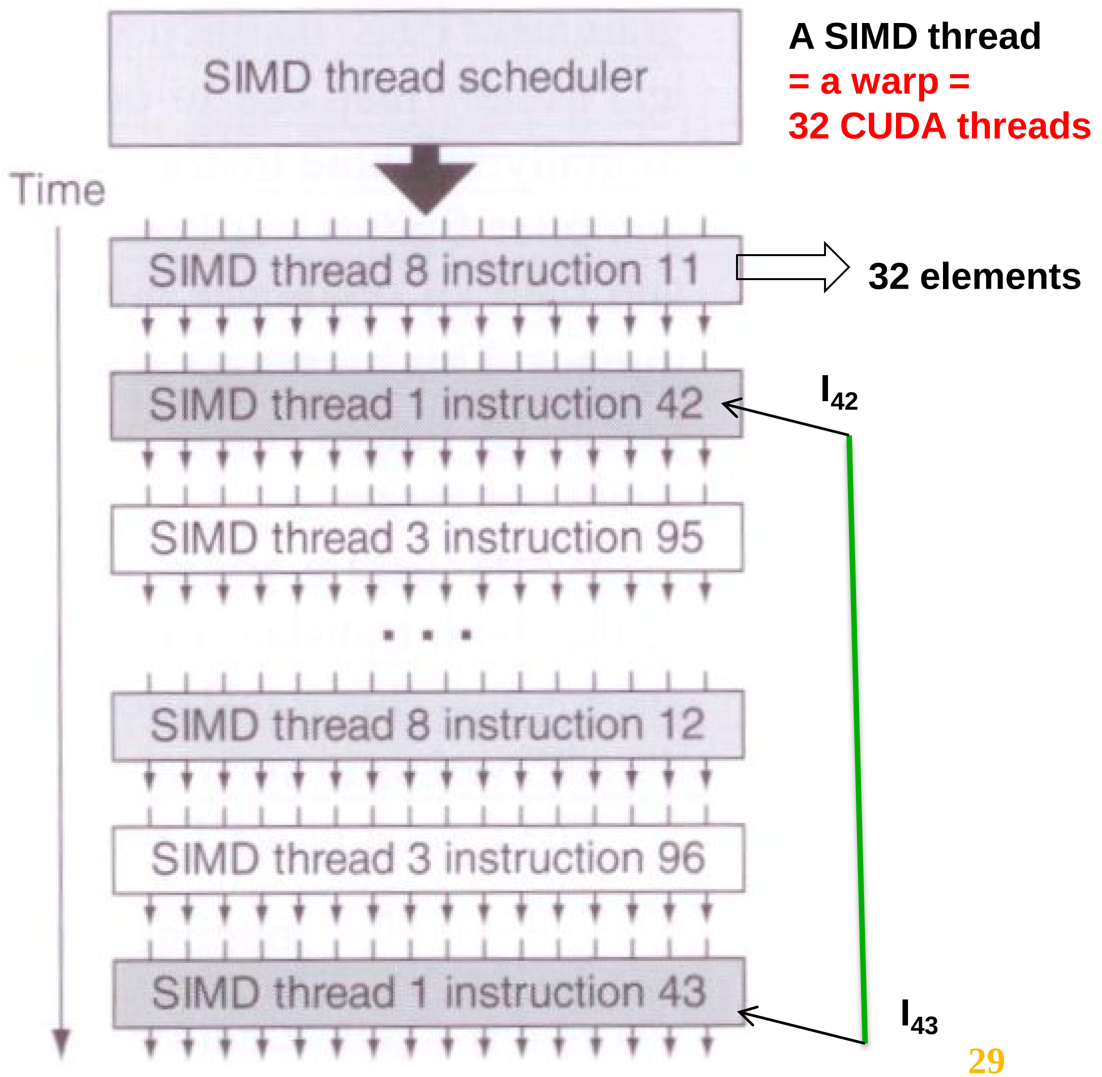
- Inside a SIMD processor, a warp scheduler selects a warp instruction for dispatching to EXE lanes:
  - Scheduling policy, instruction from which warp to dispatch
  - Round robin
  - Greedy ….
- Scoreboard: keep track of which instruction is ready for execution
- A PTX instruction = A SIMD instruction which is executed across SIMD lanes
- Associated with each warp is a warp ID, a bit vector called the active mask, and a single Program Counter (PC). Each bit in the active mask indicates whether the corresponding thread is active. When a warp is first created, all threads are active.
- If the total number of threads is not a multiple of the warp size, one warp may be created without all threads active.



V. Narasiman, et. al. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling, Micro 2011

SIMD Instruction scheduling

- Select a ready thread and issues an instruction synchronously to all the SIMD lanes executing the SIMD thread.
- Fine-grained multi-threading
  - Hide memory latency
  - Wait for pipeline stalls
  - Wait for execution latency

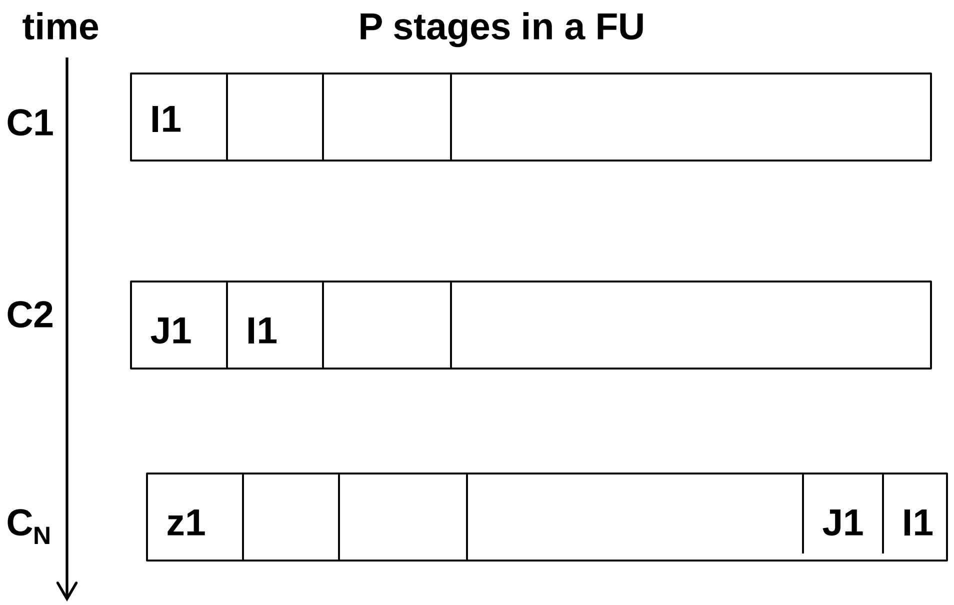


A SIMD thread
= a warp =
32 CUDA threads
SIMD thread scheduler
Time
SIMD thread 8 instruction 11
32 elements
I_{42}
SIMD thread 1 instruction 42
SIMD thread 3 instruction 95
SIMD thread 8 instruction 12
SIMD thread 3 instruction 96
SIMD thread 1 instruction 43
I_{43}

Instruction Stream Scheduling and Pipeline

- An SM executes one or more thread blocks
- A group of X-threads called a warp
- A warp scheduler issues (broadcasts) one instruction to either X cores (thus SIMD) or Y Load/Store Units, or to Z SFUs.
- However, this is a pipeline functional unit!
- Assuming independent N instruction streams or N warps
  - Example: N=48, a warp scheduler picks from 48 warps for instruction dispatching!

See a core in the system



Warp1: I1 I2 I3 I4 I5…
Warp2: J1 J2 J3 J4 J5..

WarpN: z1 z2 z3….

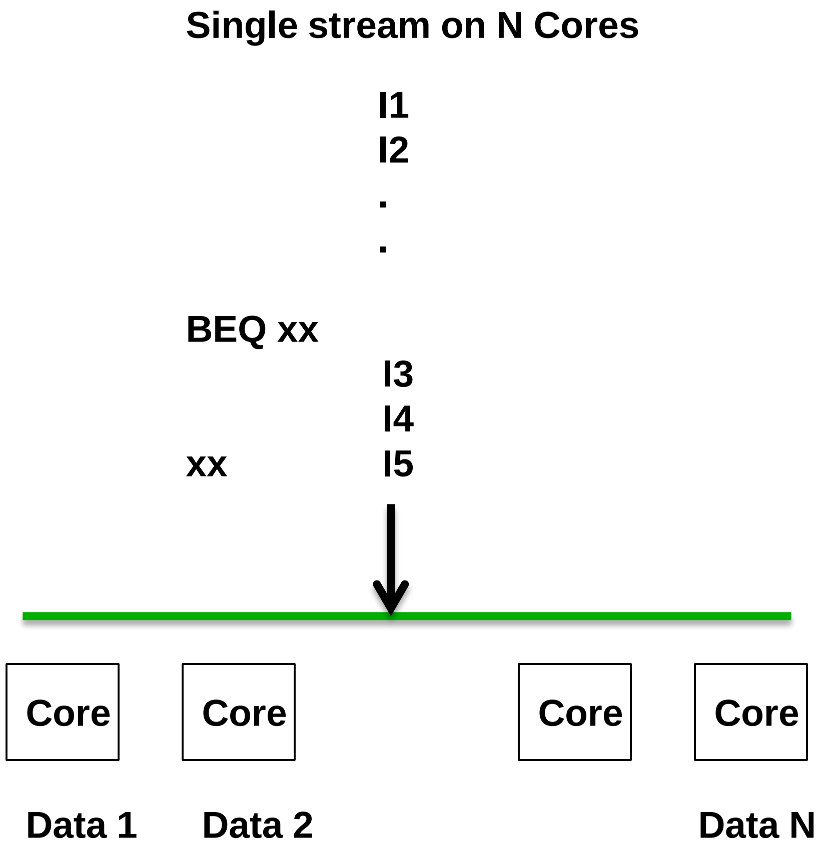
After N cycles,
I1 completes
Warp back to
Issue I2 of Warp1, and etc.
So, if I2 depends on I1,
It has a room of N cycles
for execution latency.

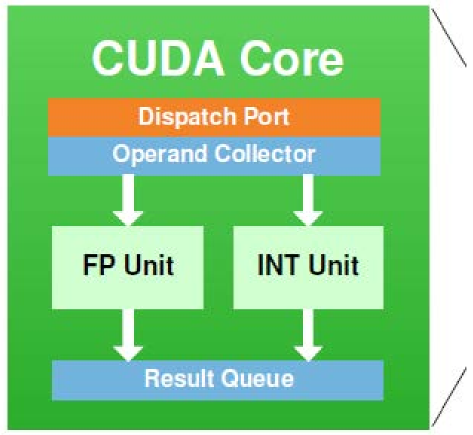
Address Coalescing Hardware

- **For data transfer from/to memory, a burst transfer of, say 32 sequential words is performed by the runtime hardware.**
- **To do this, the programmer must ensure that adjacent threads access nearby addresses at the same time so that they can be coalesced into one or a few memory blocks.**

ISA issues for SIMT

- **Branch problem in SIMT**
  - Can not use "regular branches" in SIMT because
  - If some gets I3 etc and some get I5,
  - then there is no single instruction stream anymore.





If-Conversion for SIMT

- If-conversion uses predicates to transform a conditional branch into a single control stream code.

```
if(r1 == 0)
   add r2, r3,r4
else
   sub r2, r7,r4
mov  r5, r2
```

code using br

```
f0: cmp r4, #0
f4: beq 0x100
f8: sub r2, r7,r4
fc: bne 0x104
100: add r2, r3, r4
104: mov r5, r2
```

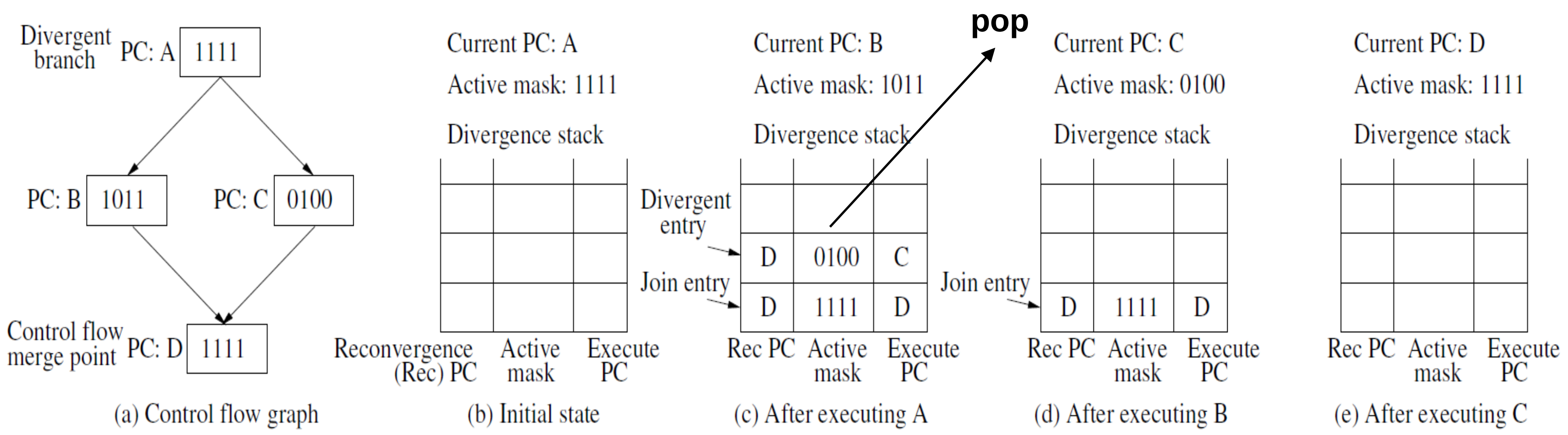
Control merge point

If-converted code

```
cmp         r1 , #0
addeq       r2, r3,r4
subne       r2, r7,r4
mov         r5, r2
```

Conditional Branch

- **Like vector architectures, GPU branch hardware uses internal masks**
- Branch divergence at the end of basic block A,
  - (1) Push control flow merge point first (Rec PC, Active mask, Execute PC) at control flow merge point D
  - (2) Push the other path to be executed later. Execute basic block B first, push the other path. (Rec PC, Active mask {C}, Execute PC{C}) for divergence point.
  - After executing the first path, in basic block B, when PC + 8 = Rec PC (TOS), set the second path: PC = C (TOS), Active Mask = Active Mask (TOS), pop TOS.

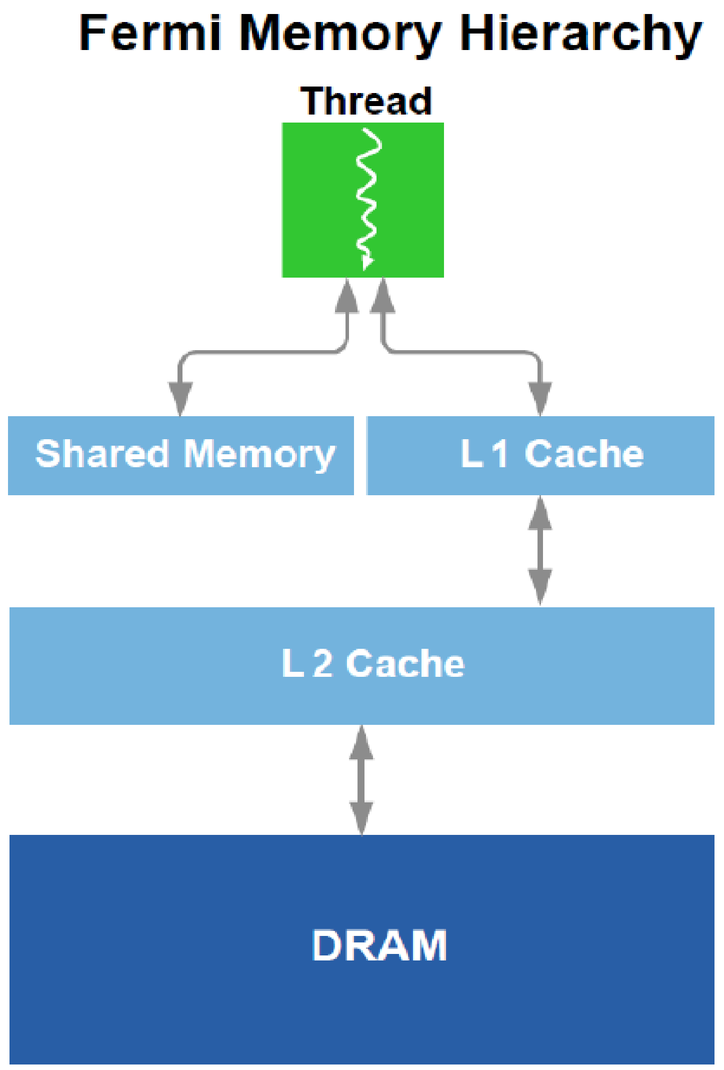


Illusion of MIMD branch-based program behavior on SIMD instructions

- **Illusion of some threads go one way, the rest go another.**
- **Illusion of a thread works independently on one element in a thread of SIMD instructions.**
- **In fact, each thread (each SIMD lane) is executing the same instruction either “committing their results” or “idle, i.e. no operation.”**

Memory Hierarchy

- Similar to general purpose CPU
- Add a scratch-pad mem for group of threads that can locally share through load/store in the instruction stream-- a common DSP technique



NVIDIA GPU Memory Structures

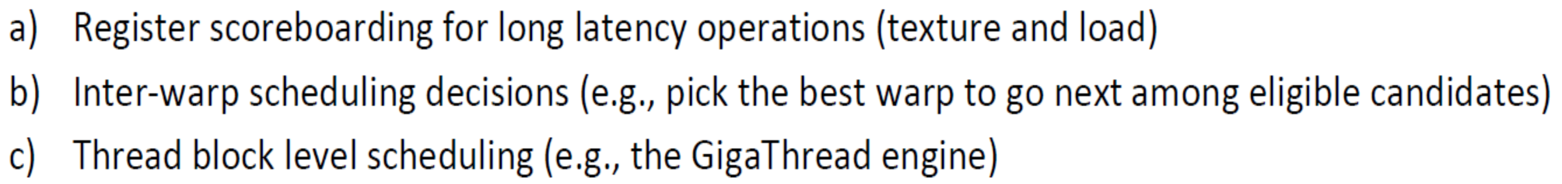
- **Each SIMD Lane (a CUDA thread) has private section of off-chip DRAM**
  - **"Private memory"**
  - **Contains stack frame, spilling registers, and private variables**
- **Each multithreaded SIMD processor also has local memory**
  - **Shared by SIMD lanes / threads within a block**
- **Memory shared by SIMD processors is GPU Memory**
  - **Host can read and write GPU memory**

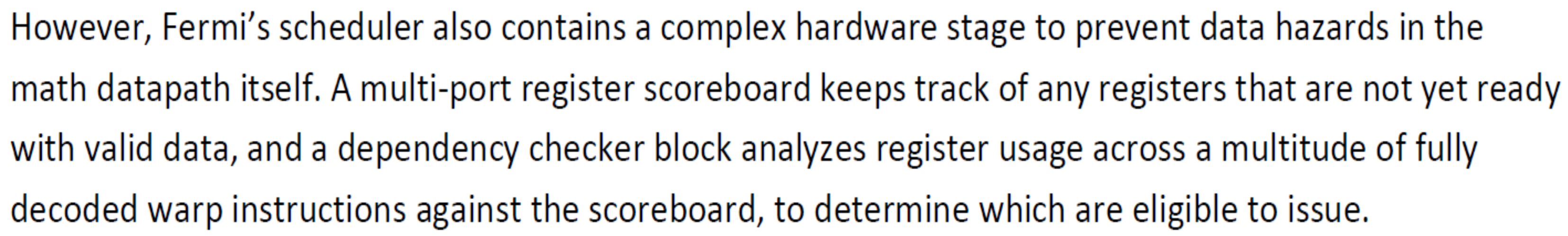
NVIDIA GPU Architecture

- **Similarities to vector machines:**
  - **Works well with data-level parallel problems**
  - **Scatter-gather transfers**
  - **Mask registers**
  - **Large register files**
- **Differences:**
  - **No scalar processor**
  - **Uses multithreading to hide memory latency**
  - **Has many functional units, as opposed to a few deeply pipelined units like a vector processor**

Inside warp scheduler

- **Scheduling optimization: ILP & Hyper threading**
  - **Limited version of OOO**
  - **Register scoreboard: Allow OOO but stall on WAW and WAR hazards. Per stream view!**
  - **For RAW hazard, similar toTomasulo's basic. Per stream view.**
  - **Many instruction streams to dispatch through multiple warp schedulers. Simultaneous Multi-Threading !**

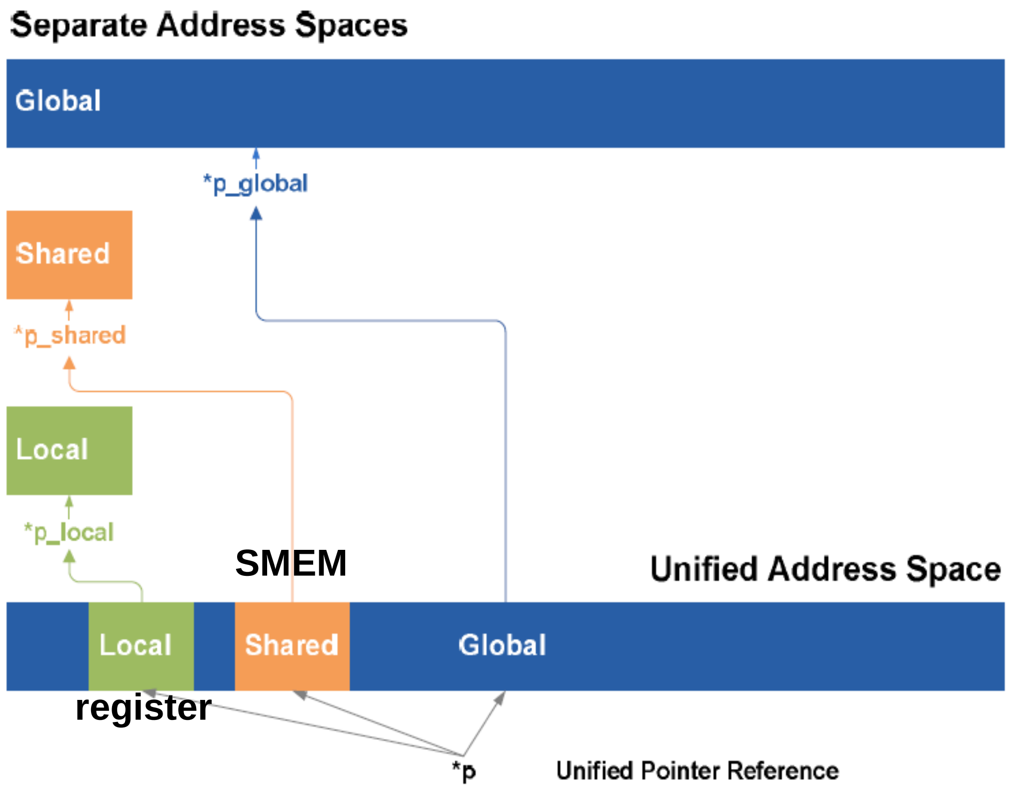

a) Register scoreboarding for long latency operations (texture and load)
b) Inter-warp scheduling decisions (e.g., pick the best warp to go next among eligible candidates)
c) Thread block level scheduling (e.g., the GigaThread engine)


However, Fermi's scheduler also contains a complex hardware stage to prevent data hazards in the math datapath itself. A multi-port register scoreboard keeps track of any registers that are not yet ready with valid data, and a dependency checker block analyzes register usage across a multitude of fully decoded warp instructions against the scoreboard, to determine which are eligible to issue.

Unified Address Space in Program View

- **A load/store directly accesses any type of the memory.**
- **A hardware translation unit maps load/store address to the correct memory location.**

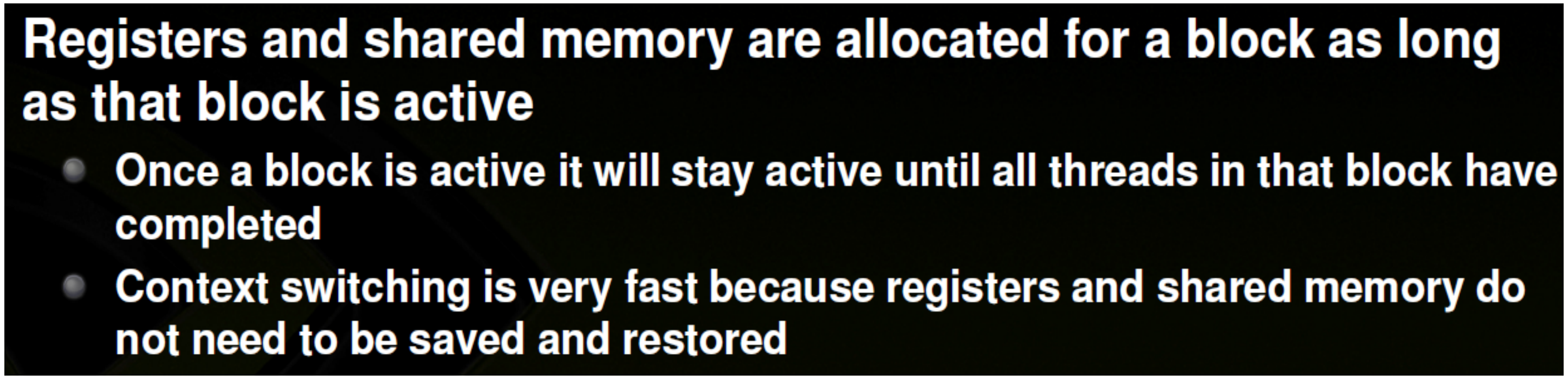
With PTX 2.0, a unified address space unifies all three address spaces into a single, continuous address space. A single set of unified load/store instructions operate on this address space, augmenting the three separate sets of load/store instructions for local, shared, and global memory. The 40-bit unified address space supports a Terabyte of addressable memory, and the load/store ISA supports 64-bit addressing for future growth.



Unified address memory access by:

- Hardware assisted page mapping that determines
  - which regions of virtual memory get mapped into a thread's private memory
  - which are shared across a block of threads
  - which are shared globally
  - which are mapped onto DRAM
  - which are mapped onto system memory
- As each thread executes, Fermi automatically maps its memory references and routes them to the correct physical memory segment.

Resource Allocation in an SM



Registers and shared memory are allocated for a block as long as that block is active

- Once a block is active it will stay active until all threads in that block have completed
- Context switching is very fast because registers and shared memory do not need to be saved and restored

- **How many active threads to run depends on**
  - **How many registers to use for a thread**
    - » **since total has 32K registers**
  - How much SMEM to use for a thread

As usual, Compiler determines these allocations!

Resource Utilization in an SM

- **Utilization determined by:**
  - How many registers are allocated to each active thread or to each instruction stream? (compiler)
  - How many SMEM are allocated to each thread? (compiler)
  - Each SM support s 8 active blocks and how big is the block size of each of the active blocks? Cannot be too small! (programmer??)

✓ **Example**

a thread uses 21 registers, 32K/21 = 1560 threads

1560 > 1536 threads (spec)

Good utilization depends on the above 3 settings!

Need to see: FU utilization, throughput achieved, and bandwidth used

And in Conclusion

- ISA Architecture for GPU
  - ISA design, branch, predication, indexed Jump, etc
- SIMT Architecture
  - Multi-threaded SIMD processor
  - Whole GPU
  - Memory support
- Software
  - Compiler
  - PTX assembler and optimizer
  - Run time