INF560 Algorithmique Parallèle et Distribuée

2021/2022

Patrick CARRIBAULT

CEA, DAM, DIF, F-91297 Arpajon

Lecture Outline – GPU (1/2)

- Overview of Dedicated Hardware
- NVIDIA GPGPU architecture
- CUDA Programming
  - Host-Side
  - Device-Side

Dedicated Hardware

- Overview

Motivations

- Homogeneous clusters
  - Building blocks: on-the-shelves CPUs
  - Good performance for majority of codes, but dedicate many transistors to micro-architecture
- Limitations of such clusters
  - Homogeneous large cores
  - Relate to limitation of general purpose processors
- One possible solution
  - →Towards dedicated hardware

Dedicated Hardware

- Definition

  Distinct resource dedicated to specific operations

- Impact
  - Aka *accelerators*
  - Processor dedicated to a specific class of applications
    - Different goals from regular CPU
    - Enable better performance for this class
  - Located on specific chip (or not)
    - Linked to CPU through some interconnect
    - Master/slave behavior

Interconnect Types

- Advantages
  - Low cost
  - Smooth integration
- Drawbacks
  - Moderate performance (generic bus)
  - Sub-optimal latency and bandwidth
- NVIDIA GPGPU, Intel Xe, AMD MI100

System Bus

- Advantages
  - Fast access
  - Possibility to share resources w/ CPU
- Drawbacks
  - Complex integration
  - Processor dependency
- IBM Cell , AMD/ATI fusion architecture

Processor Bus

Main Architecture

- Microarchitecture
  - Driven by target class of applications
  - Simplified compared to regular CPUs
    - No OoO execution
    - No branch predictor
    - …
- Exploited parallelism
  - Fine-grain parallelism (in general)
  - Mostly following the *Single Instruction Multiple Data (SIMD)* form
- But
  - Major work is let to software stack and code developer!

Main Architecture

- Memory hierarchy
  - Memory controlled by software/developer *(scratchpad memory)*
  - Cache handled by hardware (part of microarchitecture like on regular CPU)
  - Hardware connection between compute processors
  - Large bandwidth between device on its global memory

- Summary
  - Interesting characteristics for HPC
  - Many aspects to handle *by hand*
  - Direct influence on programming and execution model

Accelerator Programming

- Accelerator can be seen as slave device
- Main processor controls accesses to dedicated hardware
  - Main processor (CPU): *host*
  - Accelerator: *device*
- Main concepts
  - Data transfers
  - Remote code execution with dedicated language
  - Manual management of execution
  - Queries to accelerators

Existing Accelerators

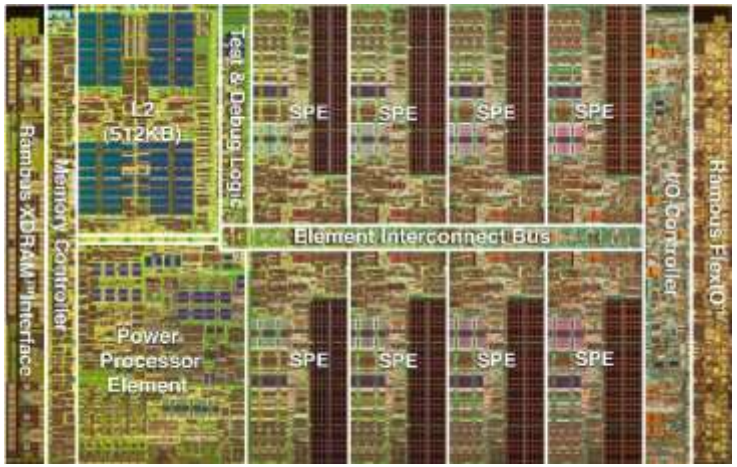
- Graphics-based processor
  - GPGPUs (General Purpose Graphics Processing Units)
  - NVIDIA
  - AMD
  - Intel Xe
- Dedicated accelerators
  - Cell (IBM)
  - Intel Xeon Phi (KNC/KNL)
- Reconfigurable architectures
  - FPGA

Graphics Processor

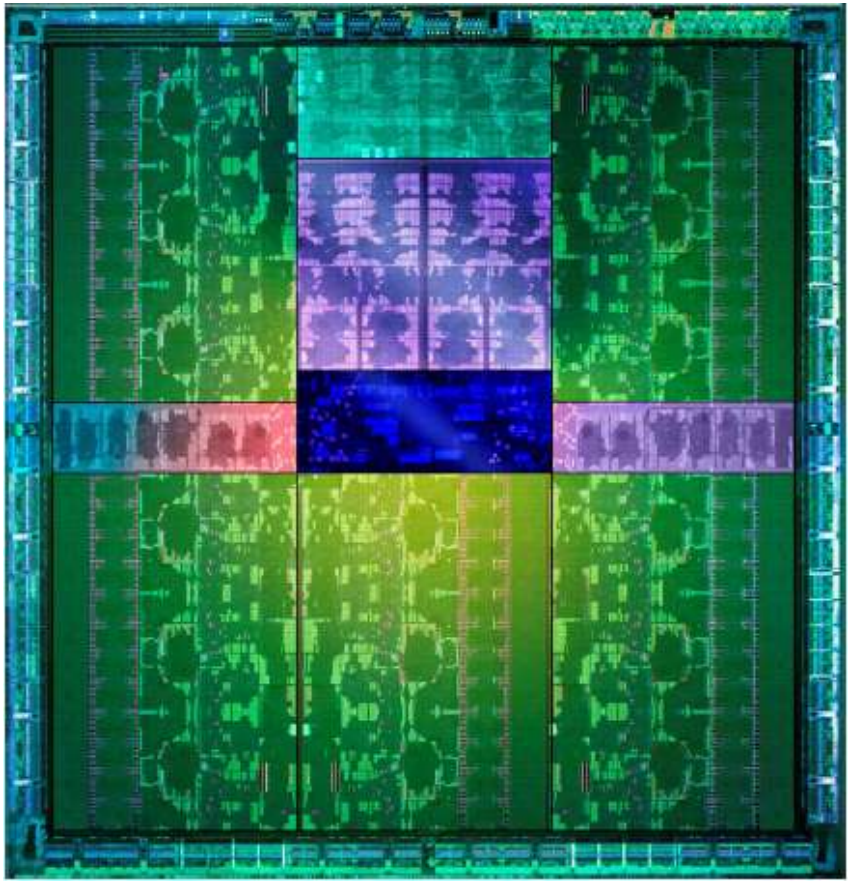
- Initially designed to print pixels on screen (for 2D or 3D graphics)
  - Dedicated processor for this purpose
- Large compute capabilities
  - Optimized to image rendering
  - Well adapted to regular data processing
- For scientific workloads
  - GPGPU (General Purpose Graphics Processing Unit)
  - Graphics cards on system bus
    - Connection to motherboard
    - Communication through PCI-E
  - Possibility to connect multiple graphics cards
    - Multi-GPU programming
    - Current configuration in heterogeneous supercomputers

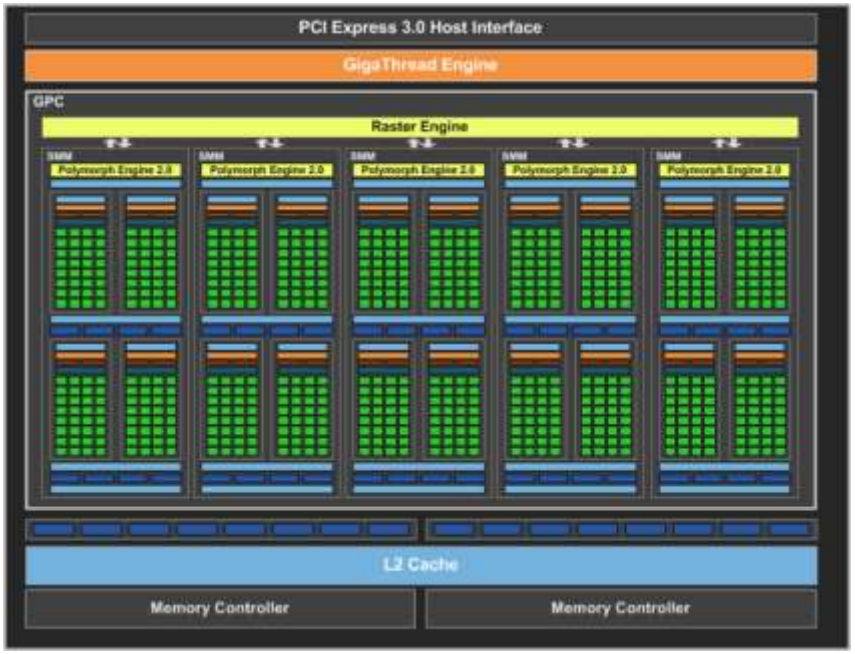
IBM Cell Processor

- Joint development between IBM, Sony and Toshiba
- 8 64-bit cores (w/ FP units)
  - *Synergistic Processor Elements* (SPEs).
  - SIMD: 128-bit register width (4 words of 32 bits)
- Master processor: 64-bit PowerPC
  - 2 logical threads
- Register file
  - 128 operands



NVIDIA Maxwell Architecture

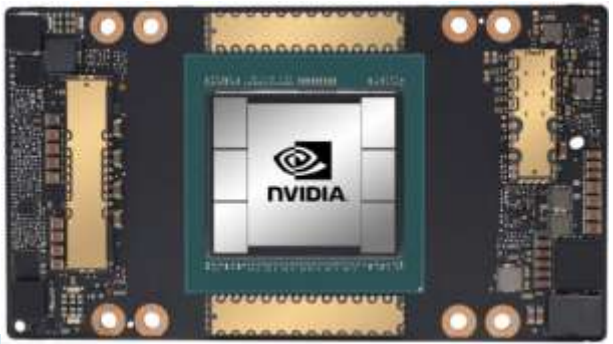




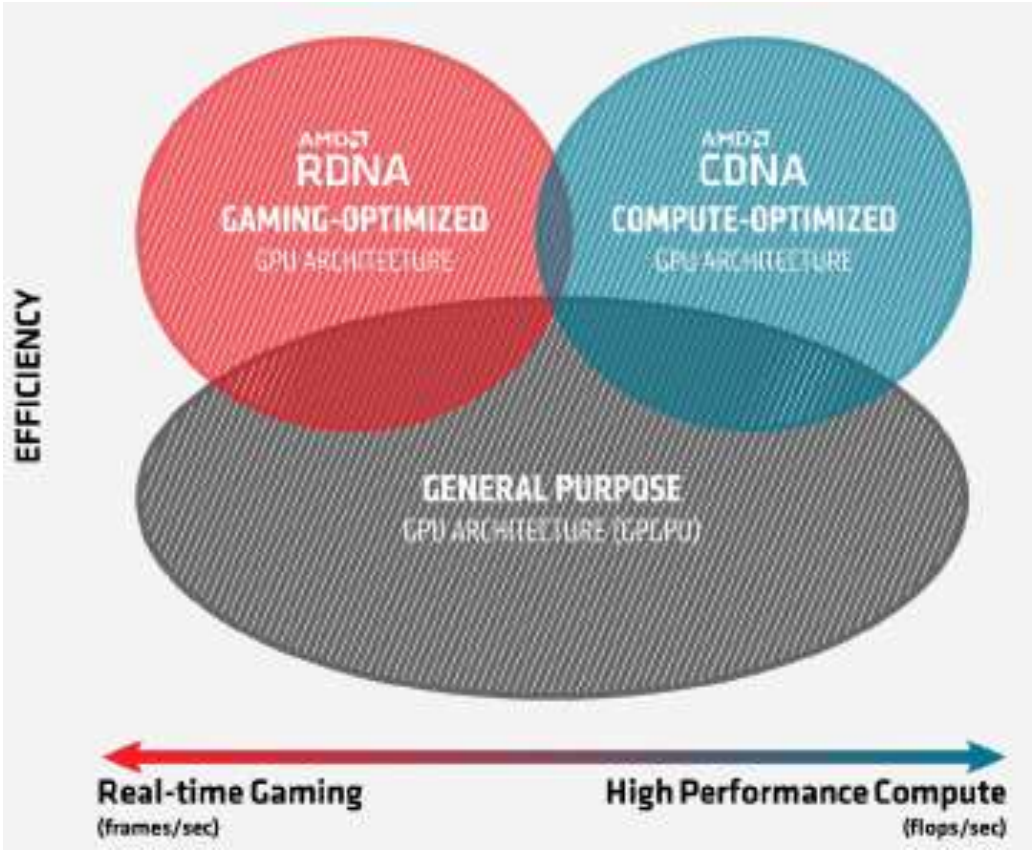
NVIDIA Pascal Architecture

NVIDIA Volta Architecture

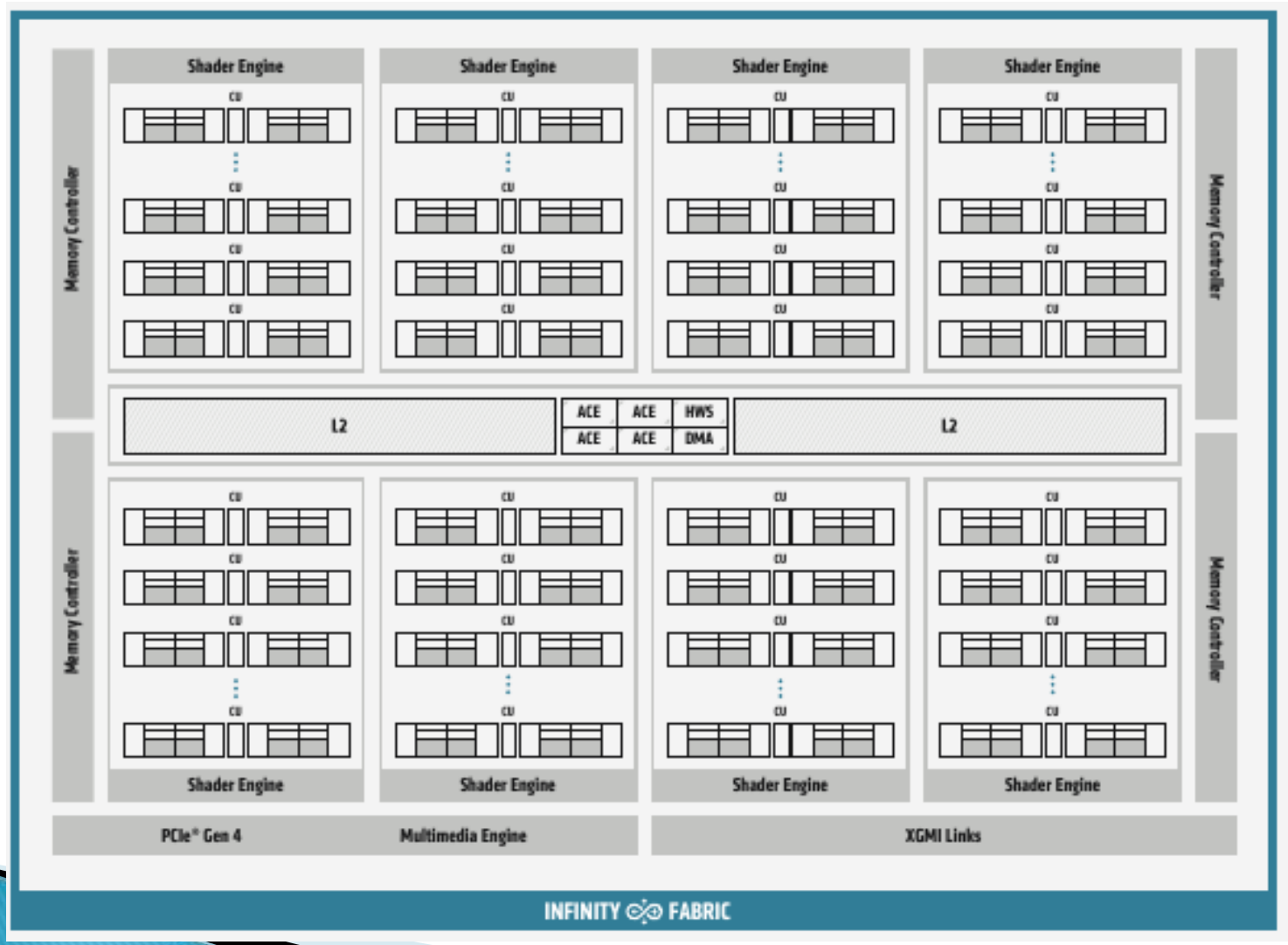
NVIDIA Ampere Architecture



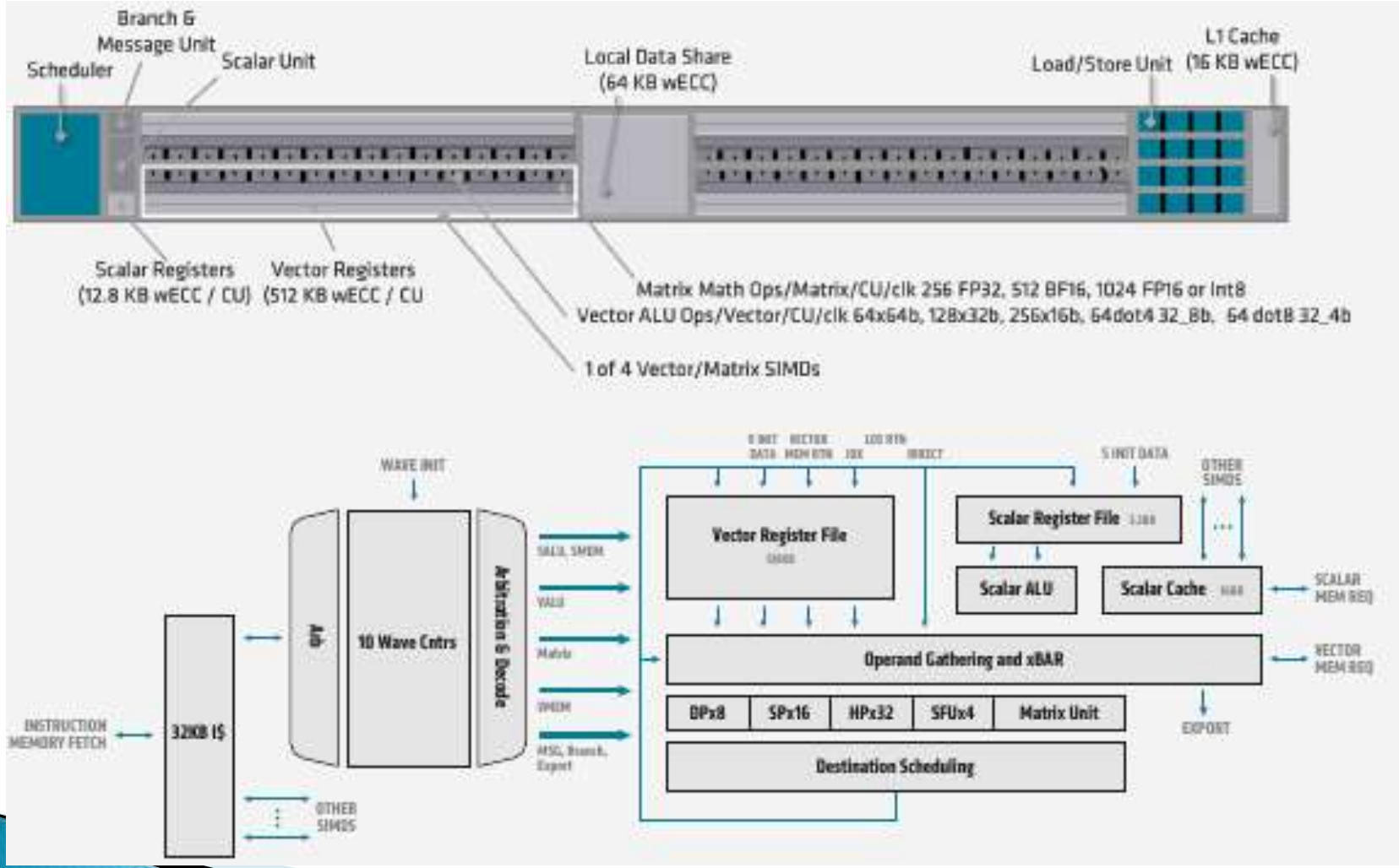
AMD CDNA Architecture



AMD CDNA Architecture

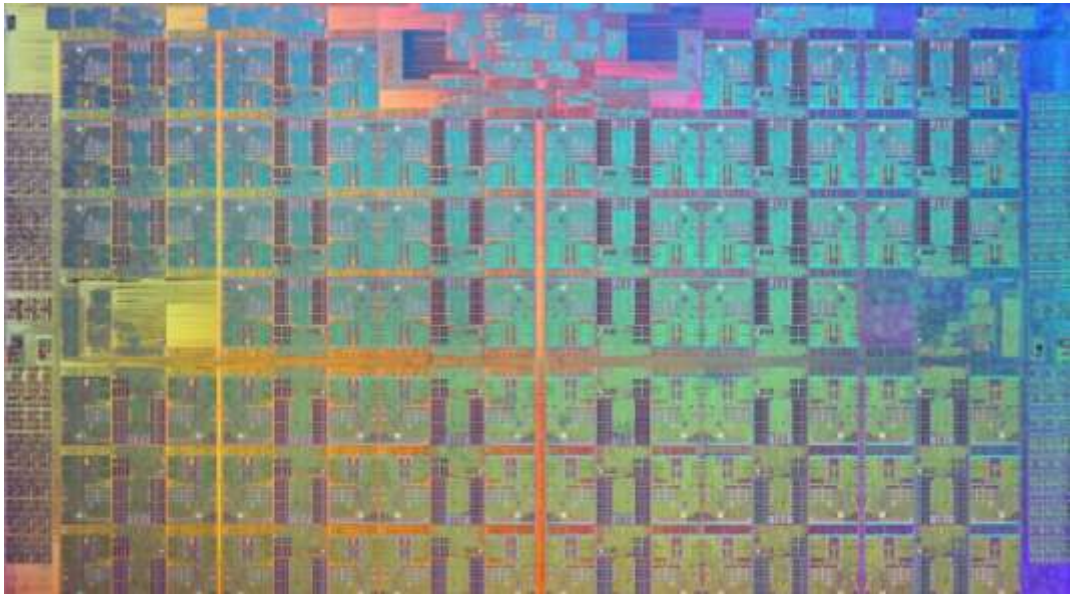


AMD CDNA Architecture



Intel Xeon Phi

- Accelerator designed by Intel
  - Many-Integrated Core (MIC)
- Introduced in 2008
  - @SigGraph 2008 conference
  - Codename at that time: *Larabee*
- Architecture
  - X86 cores
  - Support of *hyperthreading*
  - Cache coherency
- Current version: KNL
  - Exist in non-accelerator mode



Future Exascale Systems

- Next year: first exascale machine!
- Funded by DoE, located in National Labs
- Argonne National Lab: Aurora
- Oakridge National Lab: Frontier
- Based on GPUs!

ANL - Aurora

- Sustained performance > 1 Exaflop
- Aggregate system memory > 10 PB
- Storage > 230 PB, 25 TB/s (DAOS)
- Compute node
  - 2 Intel Xeon processors
  - 6 Xe arch-based GP-GPUs
- Interconnect
  - CPU-GPU: PCIe
  - System : Cray Slingshot
- Software stack
  - Cray stack (+ Intel improvement)
  - Programming models : Intel OneAPI, OpenMP, SYCL/DPC++

ANL - Aurora

Building the Foundation for Exascale Computing

2 INTEL XEON SCALABLE PROCESSORS
"Sapphire Rapids"

6 Xᴱ ARCHITECTURE BASED GPU'S
"Ponte Vecchio"

ONEAPI
Unified programming model

LEADERSHIP PERFORMANCE
For HPC, data analytics, AI

UNIFIED MEMORY ARCHITECTURE
Across CPU & GPU

ALL-TO-ALL CONNECTIVITY WITHIN NODE
Low latency, high bandwidth

UNPARALLELED I/O SCALABILITY ACROSS NODES
8 fabric endpoints per node, DAOS

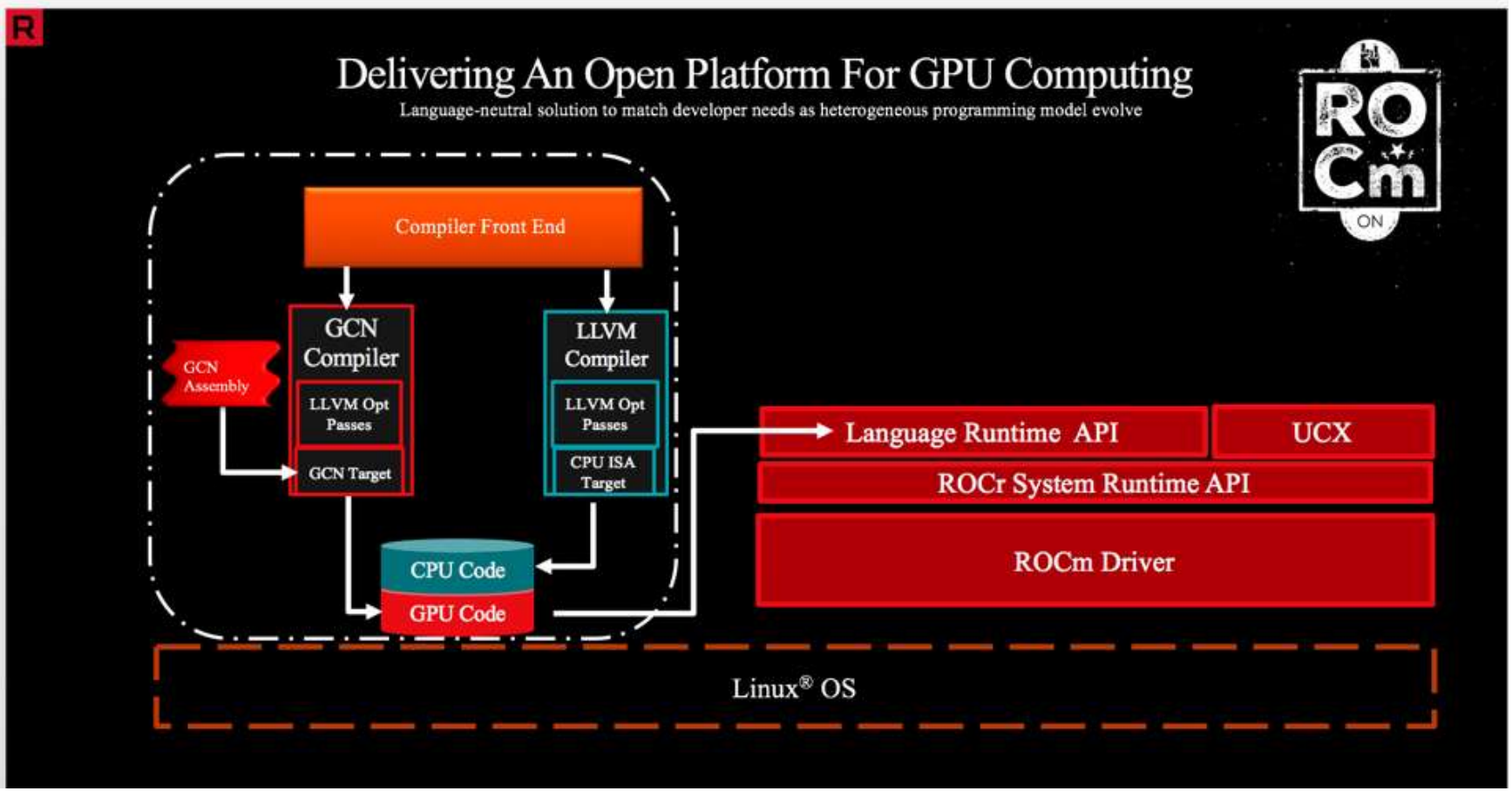
DELIVERED IN 2021

U.S. DEPARTMENT OF ENERGY | Argonne NATIONAL LABORATORY | intel | CRAY

ONL - Frontier

- Peak Performance > 1.5 Exaflop
- More than 100 cabinets
- Node
  - 1 HPC and AI optimized AMD EPYC CPU
  - 4 AMD Radeon Instinct GPU
- Interconnect
  - CPU-GPUI: AMD infinity Fabrix (coherent memory across node)
  - System: Cray Slingshot
- Software stack
  - Cray + AMD ROCm
  - Programming models : MPI, OpenMP 5.X, HIP, …

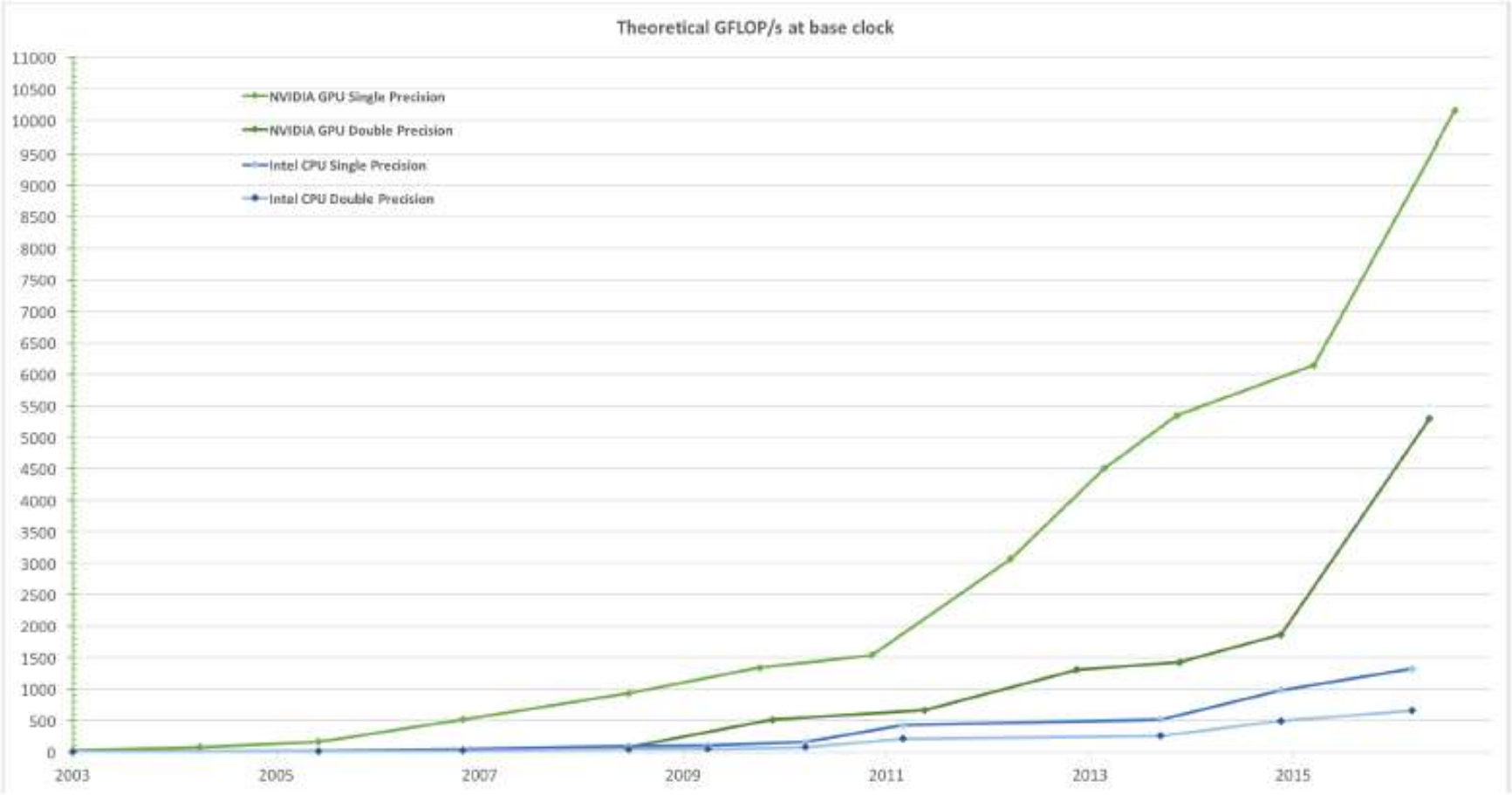
ONL - Frontier



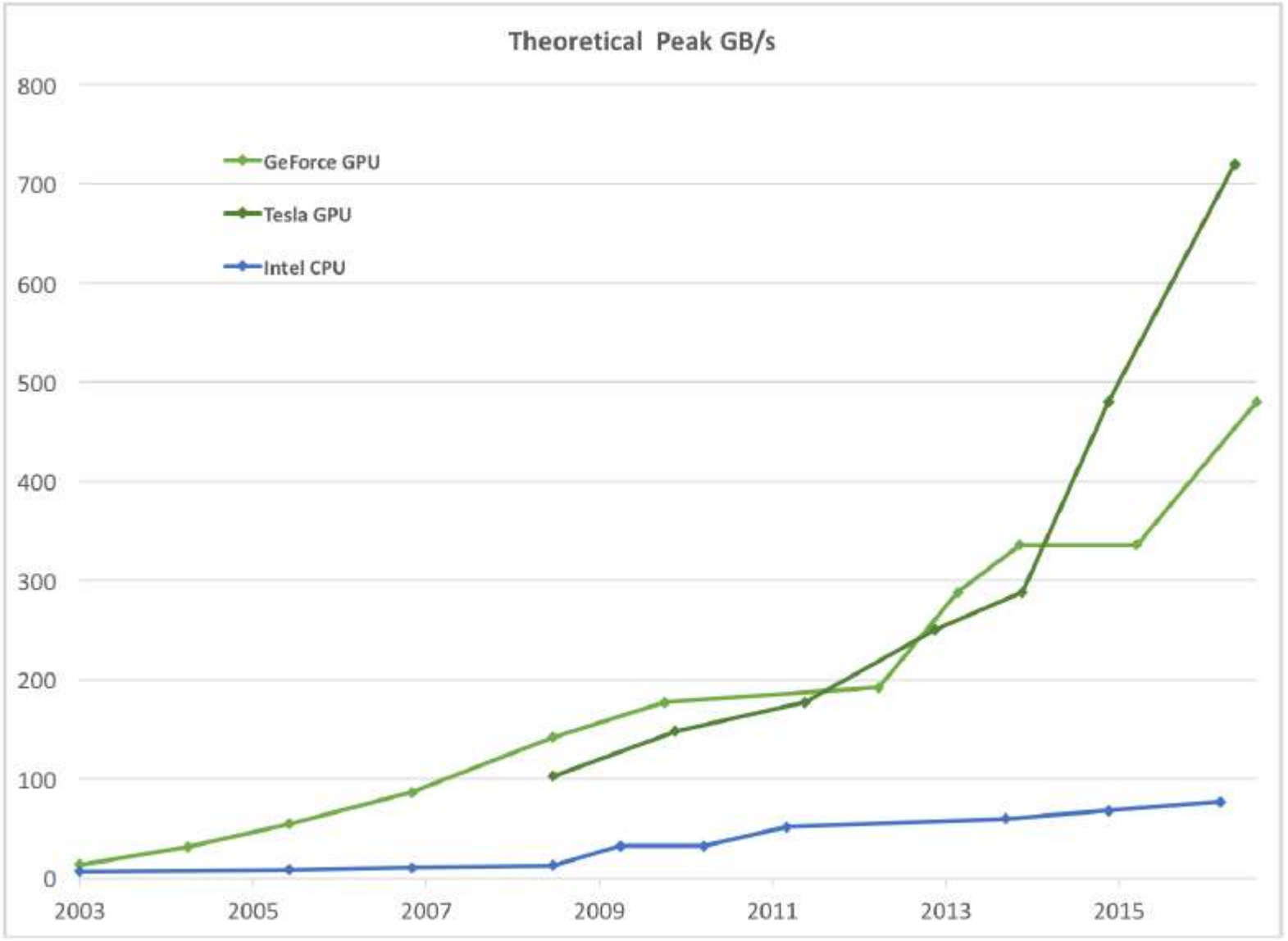
Dedicated Hardware

- NVIDIA GPGPU Architecture

Floating-Point Performance



Bandwidth Evolution

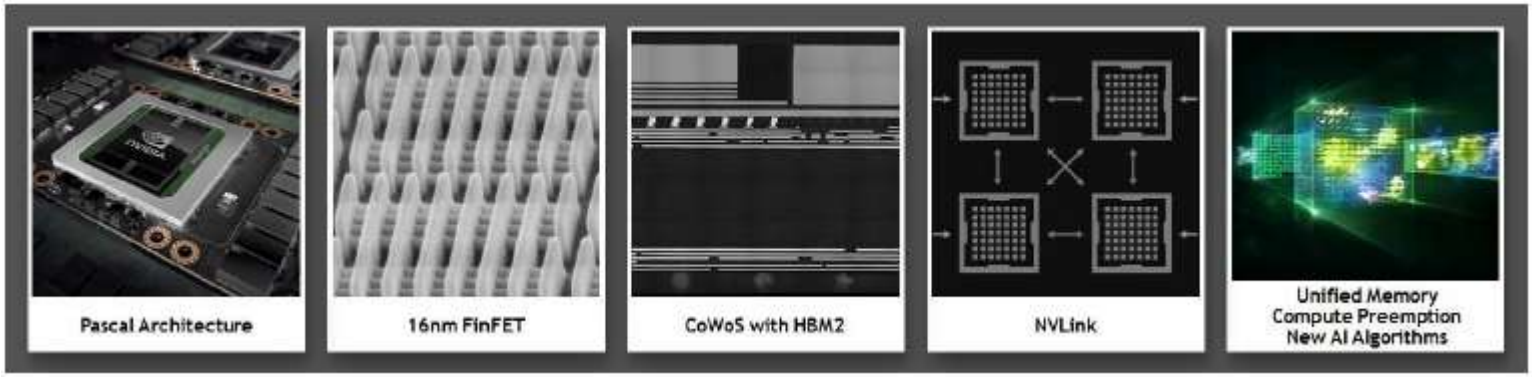


NVIDIA Architecture Overview

- Building block
  - Very simple compute core
  - Set of cores grouped inside a *stream multiprocessor*
- Why *stream ?*
  - Made to handle data streams (contiguous flow of data)
  - Synchronous language on each multicore processor
- Designed for fine-grain parallelism

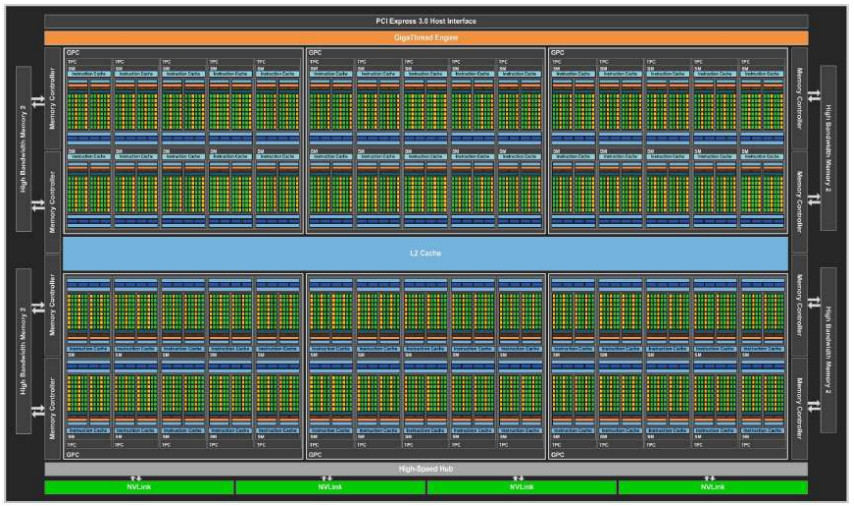
NVIDIA Pascal Architecture

- Recent generation of NVIDIA architecture (w/ double-precision performance)
  - Codename: Pascal
- Tesla version
  - 15.3 billion transistors
  - 5.3 Tflops (double precision)



NVIDIA Pascal Architecture

- GP100 → array of 6 GPCs (Graphics Processing Clusters)
  - 1 GPC → 10 SMs (Streaming Multiprocessors)
- Placement according to L2 cache
- Figure
  - Green → compute core
    - Double-precision unit in orange
  - Orange → scheduler and dispatcher
  - Light blue → register file and L1$



NVIDIA Pascal Architecture

Pascal: *Streaming Multiprocessor*

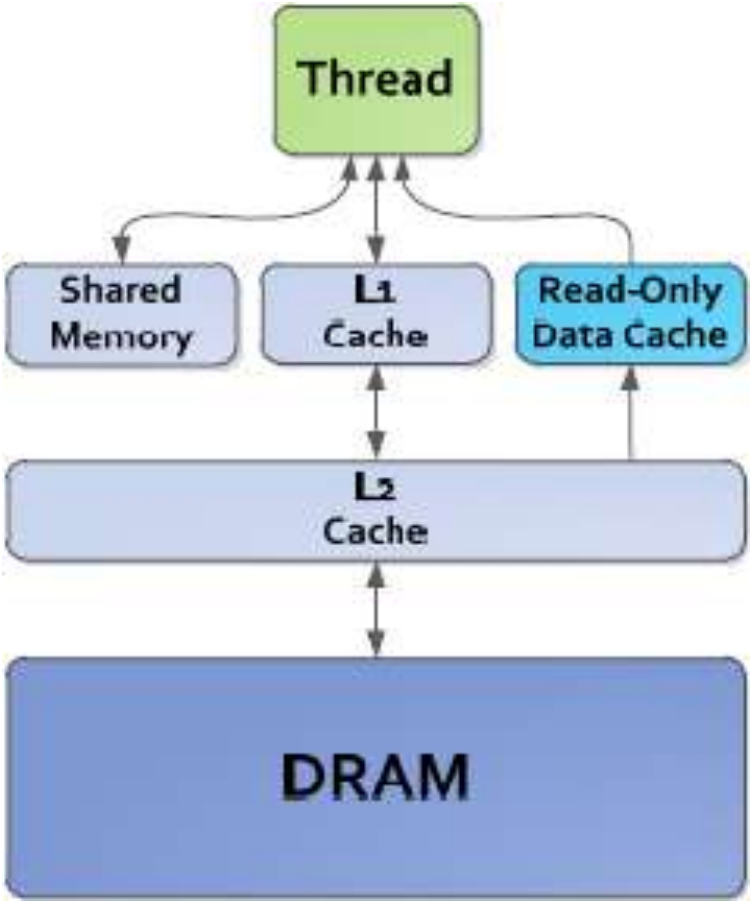
- 6th generation
- 64 CUDA-compatible cores
  - ALU unit
  - Simple-precision compute unit
- 32 double-precision units
- 16 special units (SFU)
- 16 load/store units

Pascal: *Streaming Multiprocessor*

- SIMT-based execution
  - Single-Instruction Multiple Thread
- Similar to SIMD
- Synchronous execution on compute cores
- Each SM manage threads through packs of 32
  - Notion of *warp*
  - 2 warp scheduler per SM
  - 2 dispatch units/scheduler ➔ 2 issued instruction per cycle

Pascal: Memory Hierarchy

- Threads are scheduled on one compute core (on one SM)
- Memory access through *load/store* instruction
- 2 possibilities
  - Shared-memory access➔ no cache
  - Access to global device memory ➔ go through 2 levels of cache
- L1$ and shared memory corresponds to the same physical memory or distinct space (depending on version)
- *Read-only cache* (48 KB)



CUDA Programming

- Host-Side Programming

Overview

- Kernel-based programming
- Two parts:
  - Host programming
  - Compute kernels
- Need to transfer data back and forth
- Compilation chain dedicated to computer kernels (slightly different source language)
- Link to an software ecosystem
  - Runtime system (user space)
  - Driver (kernel space)

Programming Steps

- Template of CUDA code

1. Initialization
   - Memory allocation
   - Read input data
   - …
2. GPU memory allocation
   - Dynamic allocation from host
3. Host → device transfers
4. Remote execution of compute kernel
   - Possibility to launch multiple kernels
5. Device → host transfers

Compilation Process

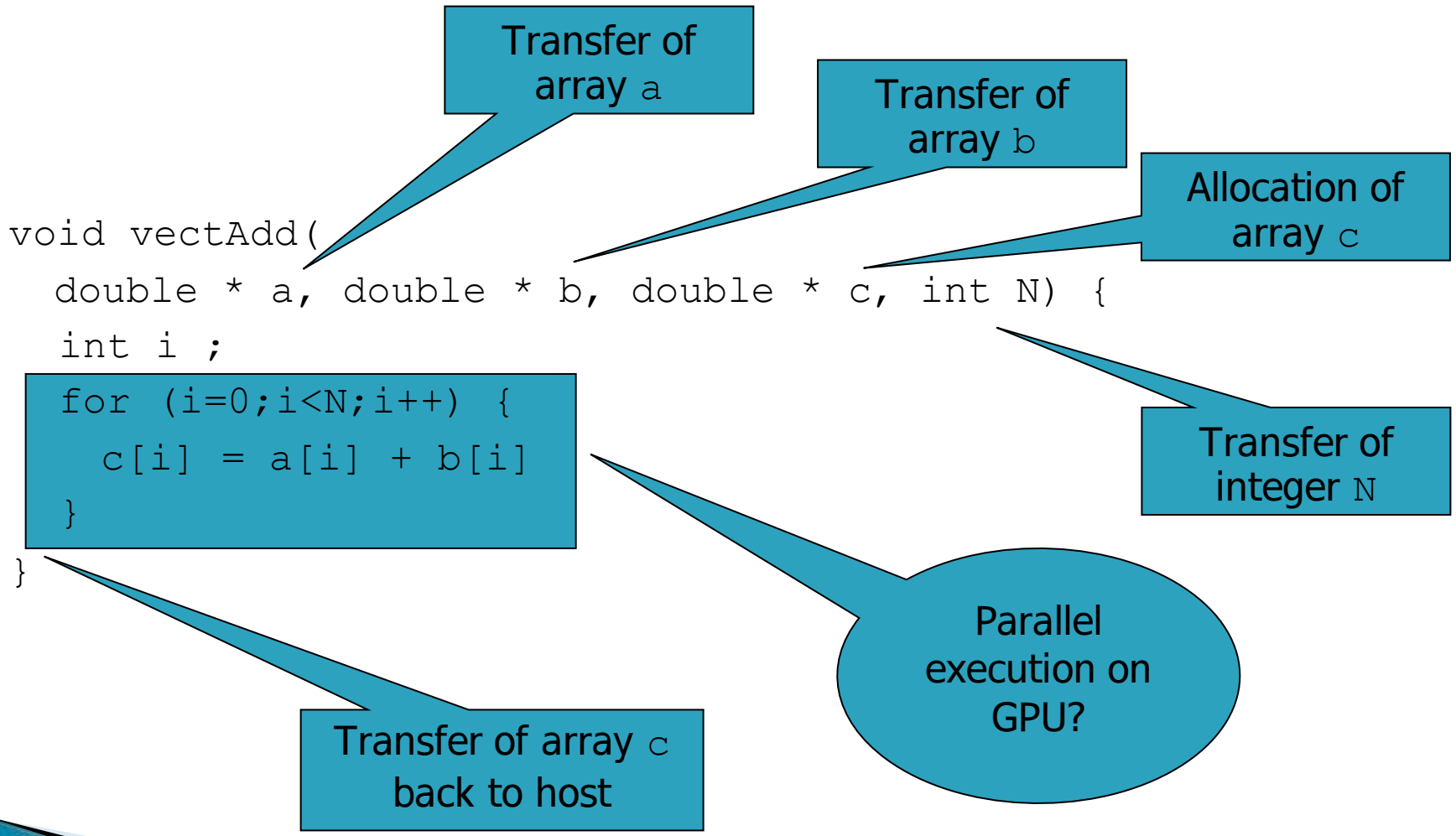
- Code structure
  - 2 parts
    - Host part
    - Device part
  - Two parts can be mixed inside same source file
- Host programming
  - C or C++
  - FORTRAN support developed by PGI
- Device programming
  - CUDA language
  - Superset of C99
- Compilation chain: NVCC
  - Use of `nvcc` compiler
  - Options available in the SDK documentation

First Example

- Example: vector addition
- Basic operation:
  - c[i] = a[i]+b[i]
- How to program this kernel in sequential?
  - Simple loop
  - Application of basic operation on each vector element

```
void vectAdd(
  double * a, double * b,
  double * c, int N) {
  int i ;
  for (i=0;i<N;i++) {
    c[i] = a[i] + b[i] ;
  }
}
```

First Example



Host Programming

- Management from host (CPU)
  - Need to handle GPU global memory
  - Need to transfer data back and forth
  - Launch the remote device kernel

- Host-side CUDA API
  - CUDA high level / low level
    - *Runtime* or *driver*
  - `cudaMalloc`: allocate memory space on device
  - `cudaMemcpy`: transfer data with device
    - Multiple directions available

Host Programming

```
void vectAdd(
  double * a, double * b,
  double * c, int N) {

  double * d_a ;
  double * d_b ;
  double * d_c ;

  cudaMalloc((void **)&d_a,
  N*sizeof(double));
  cudaMalloc(((void **)&d_b,
  N*sizeof(double));
  cudaMalloc(((void **)&d_c,
  N*sizeof(double);
```

Device pointers

Memory allocation on GPU

```
  cudaMemcpy(d_a, a,
  N*sizeof(double),
  cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b,
  N*sizeof(double),
  cudaMemcpyHostToDevice);
  cudaMemCpy(d_c, c,
  N*sizeof(double),
  cudaMemcpyHostToDevice);

  vectAddKernel(d_a, d_b, d_c,
  N);

  cudaMemcpy(c, d_c,
  N*sizeof(double),
  cudaMemcpyDeviceToHost);

/* cudaFree… */
}
```

Transfer to GPU

Kernel execution

Transfer to host

Execution Model

- Device only has to execute kernel: vector addition

- How to program kernel on device?

```
for (i=0;i<N;i++)
  c[i]=a[i]+b[i];
```

- CUDA model defines notion of thread
  - Each thread will run the kernel function
  - By default, CUDA programs are multithreaded
    - Point of view from threads, and not from loops
    - Different from regular sequential programming

- Kernel pseudo-code:

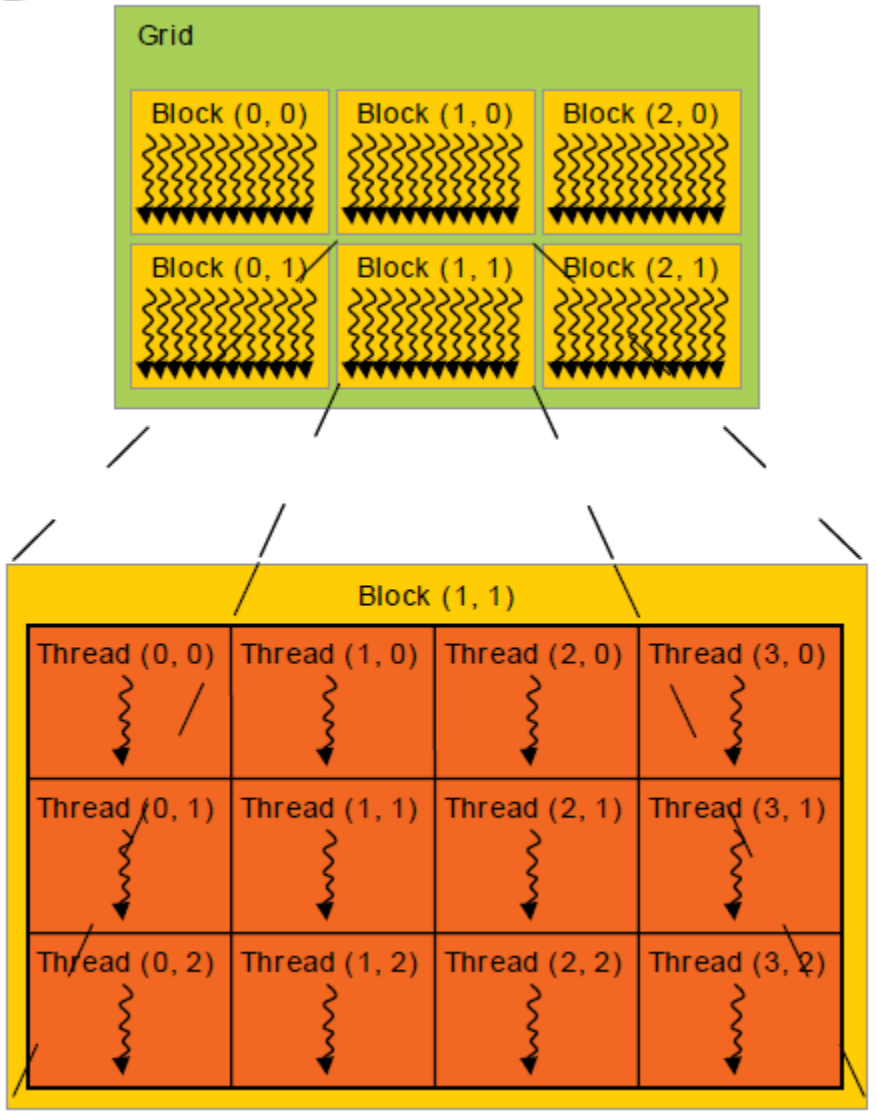
```
int i ;
/* Update i with thread index */
c[i] = a[i] + b[i];
```

Thread Hierarchy

- How to get thread id?
  - Related to thread organization w/ execution model
- CUDA introduces notion of blocks and grid to organize threads in hierarchical manner
- Grid
  - Contains blocks
    - Contains threads
- Grid and blocks may be organized in 1D, 2D or 3D
  - Logical representation

Thread Hierarchy

- Example
  - 1 2D grid
  - 6 2D blocks
- 6 blocks in the grid and 12 threads per block
  - ➔ total of 72 threads
- How to compute thread id from this representation?



CUDA Programming

- Kernel declaration
  - Keyword: `__global__`
- Compute thread index according to grid/block geometry
  - Example w/ 1D grid and 1D blocks
  - Organization should map the computational part
- Test to check that threads do not write beyond array size
  - Corresponds to upper bound from sequential loop

```
__global__ void
  vecAddKernel( double *a,
  double *b, double *c, int
  N) {

  int i ;

  i = blockIdx.x *
  blockDim.x + threadIdx.x
  ;

  if ( i<N ) {
       c[i] = a[i]+b[i];
  }
}
```

Kernel Integration

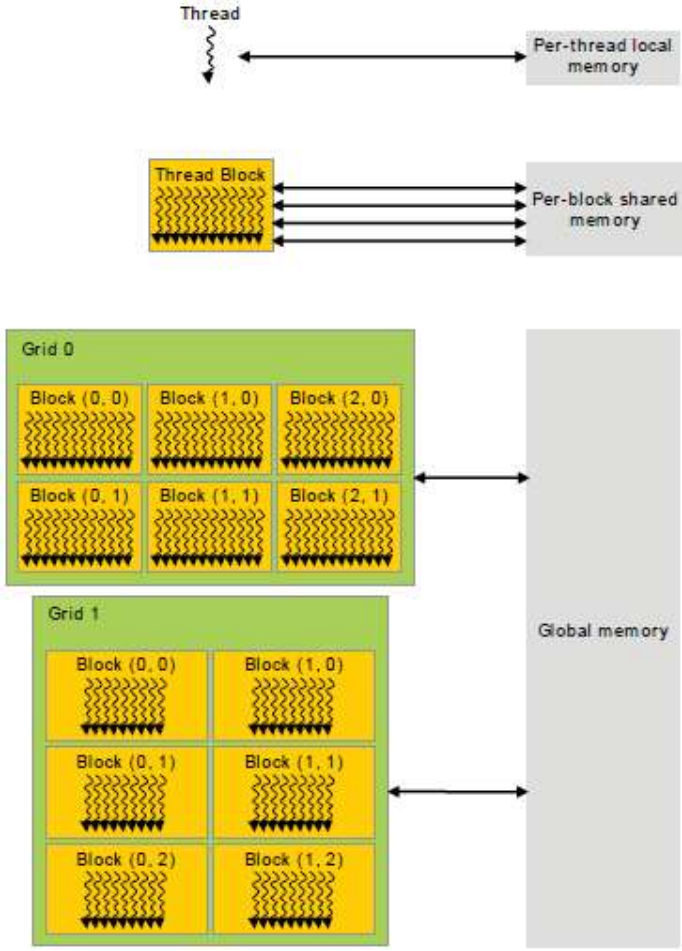
- How to choose representation of grid and blocks?
- Need to specify it during kernel remote launch
  - Host programming is in charge to define computational dimensions
- Syntax to launch a CUDA kernel from host to device

```
my_kernel<<<Dg, Db>>>(arg1, arg2, arg3 );
```

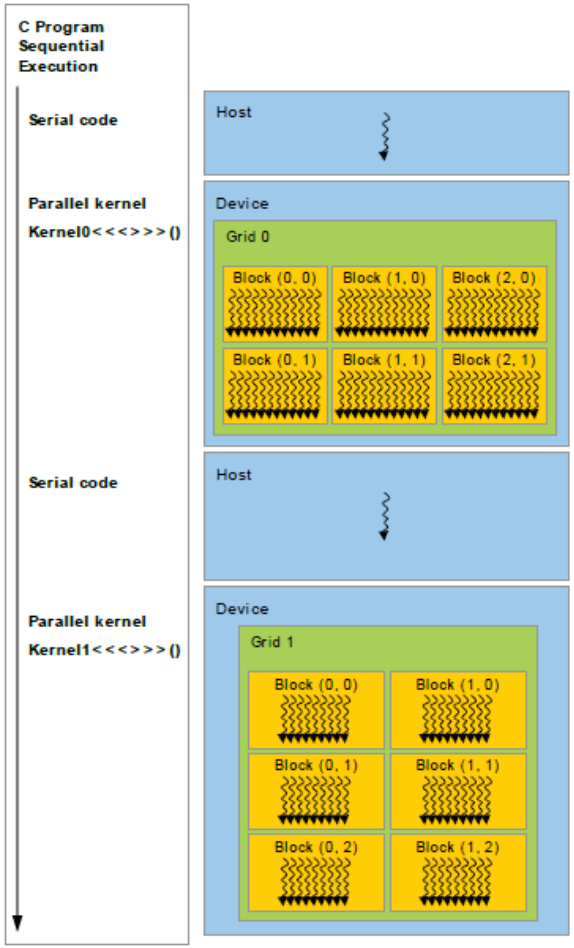
  - Dg: dimensions and size of grid (`dim3` type)
  - Db: dimensions and size of blocks in grid (`dim3` type)
- Total number of blocks?
- Number of thread per block?
- Total number of threads?

Memory Hierarchy

- Per-element vision
- Thread
  - Private local memory (stack)
  - Private register file
- Block
  - Private shared memory
- Grid
  - Global memory



Multi-Kernel Execution Model



Asynchronism

- By default some functions have asynchronous behavior
  - Kernel execution
  - Device-to-device copies
  - Memory initialization
- Possibility to wait for the end of operation execution
  - Sort of barrier

```
cudaThreadSynchronize();
```

- Force synchronism
  - `CUDA_LAUNCH_BLOCKING=1`
- Consecutive calls to kernel *vecAdd*
  - On different vectors?
  - If RAW dependencies exists?

Advanced Asynchronism

- Motivations
  - Ability to transfer data during kernel execution
  - Ability to execute multiple compute kernels at the same type
    - If target device allows it (starting w/ Fermi)
    - If there are no data dependences between simultaneous kernels
- Solution: *streaming*
- Stream declaration
  - Interaction w/ device on a particular stream
  - Driver is aware of what can be done in parallel
  - `cudaMemCpy(d_c, c, N*sizeof(double), cudaMemcpyHostToDevice, stream[0]);`
  - `my_kernel<<<Dg,Db,stream[0]>>>(arg1, arg2, arg3);`

Profiling and Timers

- Profiling
  - Use provided tools like `nvprof` (see Lab)
  - Use CUDA events to track timing in CUDA functions

- Main type: `cudaEvent_t`

- Event creation
  - `cudaEventCreate(cudaEvent_t * e)`
- Event activation
  - `cudaEventRecord(cudaEvent_t e, cudaStream_t s)`
- Waiting for even to be activated
  - `cudaEventSynchronize(cudaEvent_t e);`
- Compute time spent between events
  - `cudaEventElapsedTime(float * ms, cudaEvent_t start, cudaEvent_t stop);`

Error Handling

- In CUDA, almost all functions return an error code
  - Return type: `cudaError_t`
- If everything is ok → `cudaSuccess`
- Otherwise, possibility to access to error description
  - `const char * cudaGetErrorString (cudaError_t error);`
- For functions not returning error information (e.g., kernel call)
  - Access to last error through `cudaGetLastError()`
  - Return type: `cudaError_t`

Debugging

- How to debug a CUDA code?
- Add calls to `printf`?
  - Possible but not fully safe
- Add synchronization
- Check all error codes
  - Main way to check errors
- Transfer back to host on regular basis
  - Allow full checking w/ printing…

CUDA Programming

- Device-Side Programming

CUDA Programming

- Compute kernel
  - Based on C99 standard
  - Some restrictions
  - Some extensions
- Extensions
  - New keywords
  - Variables defined by default
  - New function
- See *CUDA C Programming Guide* document
  - Appendix B for language extension
  - Appendix F for C/C++ standard support

Basic Kernel

- Basic syntax for a compute kernel
  - Function with *void* return type
  - Attribute to declare the function to execute on device
    - `__global__`
  - Input arguments

- Example

```
__global__ void vecAddKernel( double *a,
      double *b, double *c, int N ) {
  int i ;
  i = blockIdx.x * blockDim.x + threadIdx.x ;
  if ( i<N ) {
      c[i] = a[i]+b[i];
  }
}
```

Compilation

- Use dedicated compiler chain: NVCC
- Compiler will be in charge of splitting code parts for the host and device
- File extensions
  - .cu → file containing CUDA code (and host code)
  - .c / .cpp → file containing only host code (including CUDA API)

Extensions – Keywords

- Definition of new keywords
- Categories
  - Function attributes
  - Variable attributes
  - Types
- Set of variables defined by default
  - Rely on new defined data types

Function Attributes

- Keyword to add during function declaration and definition
  - Coherency between multiple declarations and definition!
  - Between return type and function name
- Function executed on device and callable from host (i.e., starting point for kernels)
  - `__global__`
- Function dedicated to host or device (combinable)
  - `__host__`
  - `__device__`
- Behavior by default → equivalent to `__host__`

Function Restrictions

- Functions with `__global__` attribute
  - Return type `void`
  - Call with execution context (number/dimensions of blocks, number of threads per block…)
  - Asynchronous call
  - Arguments stores in shared memory (256B) or constant memory (4KB)
    - Depends on CUDA capabilities of the target device
  - Impossible to get its address (function pointer)

- Functions executing on device
  - No static variable
  - No variable number of arguments
  - Restricted recursion (only for functions declared with `__device__` attribute)

Variable Attribute

- New variable attributes to handle placement
- Device-resident variable
  - `__device__`
  - By default in global memory (shared by all threads during application lifetime)
- Constant memory resident variable
  - `__constant__`
  - Lifetime of application
  - Cannot be defined on device
- Shared-memory variable
  - `__shared__`
  - Shared between threads in the same block
  - One copy per block
  - Lifetime of block
- By default, variable declared on device is stored in a register

Volatile Variables

- Shared-data synchronization
- Example of concurrent accesses

```
// myArray is an array of non-zero integers
// located in global or shared memory
__global__ void MyKernel(int* result) {
   int tid = threadIdx.x;
   int ref1 = myArray[tid] * 1;
   myArray[tid + 1] = 2;
   int ref2 = myArray[tid] * 1;
   result[tid] = ref1 * ref2;
}
```

- What contains `result[tid]` ?
  - `myArray[tid]` is stored in a register ➔ `ref1==ref2`
- If declared volatile ➔ ok (or put a *memory fence*)
  - No guarantee on execution order

Variable Restrictions

- Dynamic management of variables in shared memory

```
extern __shared__ char array[];
__device__ void func() {
  short* array0 = (short*)array;
  float* array1 = (float*)&array[128];
  int* array2 = (int*)&array[64];
}
```

- Need to handle by hand placement of data in shared memory
  - No dynamic allocation
  - Need to respect alignment rules

Data Types

- New types
  - Vectors
  - Multi-dimension integers

- Vectors
  - Name convention: base type + width
  - Examples: `int2`, `float4`
  - Need to respect alignment rules
  - Functions available to create such types
    - Example: `int2 make_int2(int x, int y);`

Data Types

- 3D integers
  - `dim3`
  - Equivalent to vector type `uint3`
  - Access to coordinates through fields `x`, `y` and `z`
  - By default, initialized to 1

- Example

```
dim3 a ;
a.x = 4 ;
```

Predefined Variables

- Grid dimensions
  - `dim3 gridDim`
  - Contains number of blocks along 3 dimensions
- Index of block in grid
  - `dim3 blockIdx`
  - Contains coordinates of current block in grid
- Block dimensions
  - `dim3 blockDim`
  - Contains number of threads along 3 dimensions
- Index of thread in block
  - `uint3 threadIdx`
  - Contains coordonaites of current thread in its blocks
- Warp size
  - `int warpSize`

Available Functions

- Memory barrier
- Synchronizations
- Math operations
- Atomic operations
- *Timing*
- I/O

Memory Barrier

- Weakly-ordered memory model
  - Order of store instructions may be viewed as different from one thread to another
- Memory barrier
  - Mechanism to guarantee that memory accesses performed before the barrier appear to all threads as launched before the barrier
- Different levels of barrier
  - Block: `void __threadfence_block();`
  - Device: `void __threadfence();`
  - Device + host: `void __threadfence_system();`

Synchronization

- Synchronization among all threads in the same block
  - `void __syncthreads();`
  - Include data synchronization
  - Be careful of control flow !
    - Act as a collective operations within a block
- For devices w/ CUDA capabilities >= 2.0
  - `int __syncthreads_count(int predicate);`
  - `int __syncthreads_and(int predicate);`
  - `int __syncthreads_or(int predicate);`

Math Operations

- Set of math functions optimized for GPU
  - Ex: sin(float), cos(double), …
  - Less instructions but less accurate!
- Functions with `__` prefix
  - `sinf(x)` → `__sinf(x)`
- Compilation option to automatically use these functions
  - Transform regular calls to optimized ones
  - `-use_fast_math`
  - Mainly available for single-precision computations

Atomic Operations

- Operation in one instruction that guarantee atomicity
  - E.g., `int atomicAdd(int* address, int val);`
- Available functions
  - Addition: `atomicAdd`
  - Substraction: `atomicSub`
  - Exchange: `atomicExch`
  - Min/Max: `atomicMin`
  - Increment/Decrement: `atomicInc`
  - CAS, …
- Restrictions
  - Available for integers
  - Available for single-precision floats (capabilities 2.X)
  - Since 6.X, double-precision operations

Timing & I/O

- Multiprocessor timestamp
  - `clock_t clock();`
  - Number of cycles
  - Possibility to check number of clocks for a thread
- Formatted output
  - `int printf(const char *format[, arg, ...]);`
  - Require at least CUDA capabilities 2.X
  - Per-thread context
  - Final formatting on the host

Example

- Matrix multiply
- Rely on C data structure
  - Available on CPU and GPU
- Use variables w/ 2 dimensions
- Every thread computes one element of resulting matrix

```
typedef struct {
   int width;
   int height;
   float* elements;
} Matrix;
```

- What if matrices are larger than maximum number of allowed threads?

```
__global__ void MatMulKernel(Matrix
   A, Matrix B, Matrix C)
{

float Cvalue = 0;
int row = blockIdx.y * blockDim.y +
   threadIdx.y;
int col = blockIdx.x * blockDim.x +
   threadIdx.x;

for (int e = 0; e < A.width; ++e)
   Cvalue += A.elements[row *
   A.width + e]
         * B.elements[e * B.width +
   col];
C.elements[row * C.width + col] =
   Cvalue;
}
```

Example

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{

float Cvalue = 0;
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;

for (int r = row ; r < C.height ; r += blockDim.y )
  for (int c = col ; c < C.width ; c += blockDim.x )
      for (int e = 0; e < A.width; ++e)
            Cvalue += A.elements[row * A.width + e]
                   * B.elements[e * B.width + col];
            C.elements[row * C.width + col] = Cvalue;
}
```

Register Allocation

- Each compute kernel uses several registers
- Depending on instructions inside the kernel body
  - Compiler transformations/optimizations
  - Register allocation
- But
  - Limited size of register file
  - Register file is shared among threads inside the same *Streaming Multiprocessor*
- Relation w/ number of threads?

Register Allocation

- Option to set an upper bound to compiler
  - `-maxrregcount=N`

- Attribute to indicate max number of threads and blocks when calling target function

```
__global__ void
__launch_bounds__(maxThreadsPerBlock,
  minBlocksPerMultiprocessor)
MyKernel(...)
{
  ...
}
```

Memory Access

- Because of synchronous execution
  - Thread executing load operation may lower other's speed
  - Several memory accesses may be serialized
  - Long-latency access ➔ need more warps to recover latency

- Possible optimizations
  - *Load coalescing*
  - Avoid bank conflicts

Load coalescing

- Access to global memory
- Concurrent accesses emitted by threads inside the same warp
  - All threads inside the same warp execute the same instruction at the same time
- Memory requests are serialized w/ chunks of 128B
  - Size of cache line
  - Optimization is memory accesses are contiguous

Load coalescing and shared memory

- Shared memory
  - Need to explicitly declare variables inside shared memory
  - Data transfers at the beginning of kernel
  - Update to global memory at the end of kernel
- Optimization: benefit from memory-to-memory transfers to change data layout
  - Even if all data are not necessary
- Be careful to bank conflicts

Best Practices

- Higher priority
  - Think parallel
  - Minimize data transfers between host and device
  - Access main device memory w/ coalesced operations
  - Use shared memory when possible
  - Avoid path divergence

- Lower priority
  - Avoid bank conflict when accessing shared memory
  - Avoid too large number of threads per blocks (multiple of 32)
  - Rely on optimized math functions