

CS 610: GPU Architecture and CUDA Programming

Swarnendu Biswas

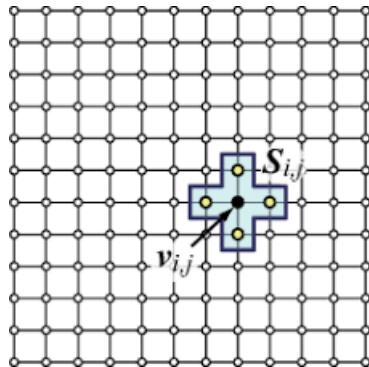
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2026-27-I



Need for Data-Level Parallelism

- Many real-world applications are compute-intensive and data-parallel
- Need to process a lot of data, mostly floating-point operations
- Examples are
 - ▶ Real-time high-definition graphics applications such as your favorite video games
 - ▶ Iterative kernels which update elements according to some fixed pattern called a stencil



Rise of GPGPU Computing

- Researchers used to trick GPUs to perform non-rendering computations, often referred to as general-purpose GPU (GPGPU) computations
- Programming initial GPU devices for other purposes was very convoluted
 - Programming model was very restrictive
 - Examples of limitations: input colors and texture units, writes to arbitrary locations, and floating-point computations
- This spurred the need for a generic highly-parallel computational device with high computational power and memory bandwidth
 - ▶ CPUs are more complex devices catering to a wider audience

Rise of GPGPU Computing

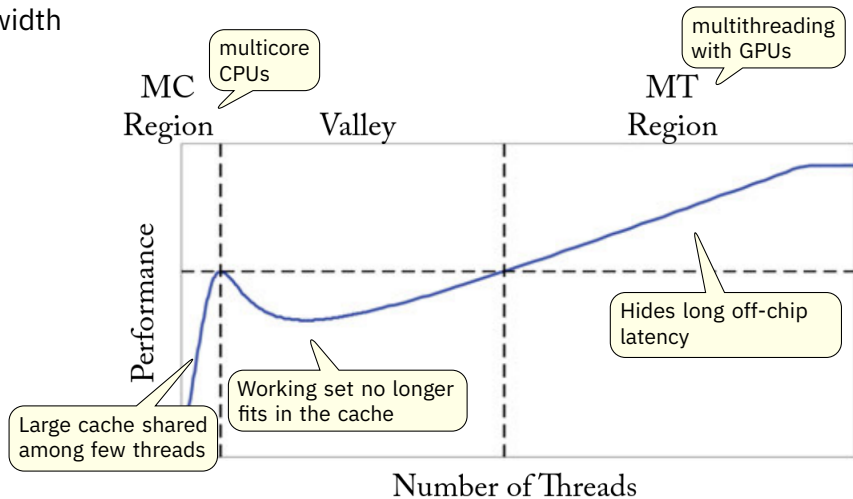
- NVIDIA released GeForce 8800 GTX in 2006 with CUDA architecture
 - + General-purpose ALU and instruction set for general-purpose computation
 - + Allowed arbitrary reads and writes to memory
 - + IEEE compliance for single-precision floating-point arithmetic
- Introduced CUDA C and the toolchain for ease of development with the CUDA architecture
- GPUs are now popularly used across several application domains
 - ▶ For example, game effects, computational science simulations, image processing, machine learning, and linear algebra
 - ▶ Modern GPUs serve both as a programmable graphics processor and a scalable parallel computing platform
- There are several GPU vendors like NVIDIA, AMD, Intel, Qualcomm, and ARM

GPU Architecture

Philosophy and Design Goals

Analytical Model to Compare CPU and GPU Performance

Simple cache model where threads do not share data and there is infinite off-chip memory bandwidth

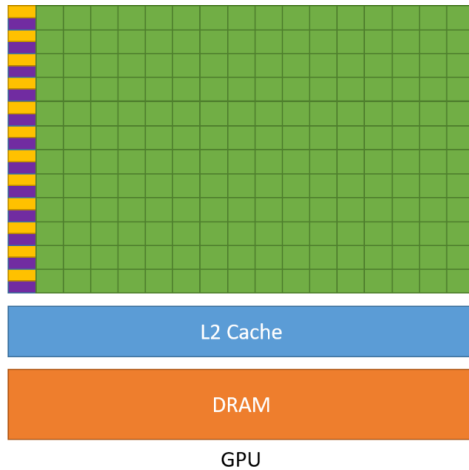
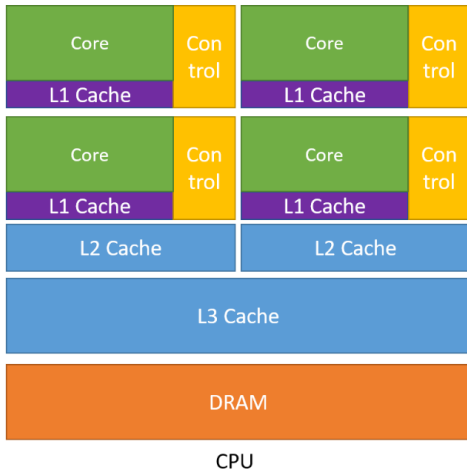


Philosophy of GPUs

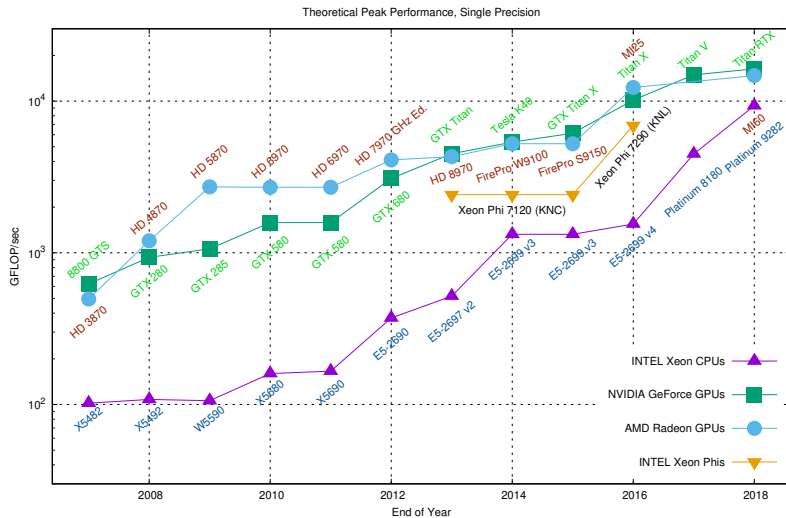
- GPUs are suited for **compute-intensive data-parallel** applications
 - ▶ The same program is executed for each data element
- Many-core chip
 - ▶ Multi-threaded execution on a single core (multiple threads executed concurrently by a core)
 - ▶ SIMD execution within a single core (many ALUs performing the same instruction)
- The focus is on overall **computing throughput** rather than on the speed of an individual core
 - ▶ GPUs do not reduce latency, they aim to **hide latency**
 - ▶ High arithmetic intensity and large number of schedulable units aim to hide the latency of memory accesses

Key Insights in GPU Architecture

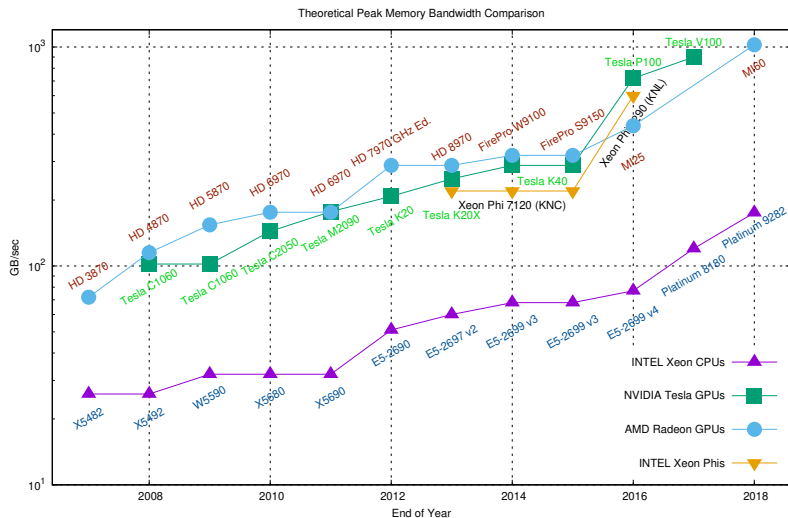
Much more real-estate (i.e., transistors) is devoted to computation rather than data caching and control flow



Comparing FLOPS



Memory Bandwidth for CPU and GPU



Compare GPU to CPU Architecture

- CPUs aim to **reduce memory latency** with increasingly large and complex memory hierarchy
- Disadvantages
 - ▶ The Intel I7-920 processor has some 8 MB of internal L3 cache, almost **30%** of the size of the chip
 - Larger cache structures increases the physical size of the processor
 - Implies more expensive manufacturing costs and increases likelihood of manufacturing defects
- Effect of larger, progressively more inefficient caches ultimately results in higher costs to the end user

Advantages of a GPU

Performance

- Comparing Xeon 8180M and Titan V (based on peak values)
 - ▶ 3.4X operations executed per second compared to the CPU
 - ▶ 5.5X bytes transferred from main memory per second compared to the CPU
 - ▶ Cost- and energy-efficiency
 - 15X as much performance per dollar
 - 2.8X as much performance per watt

Energy

- GPU's higher performance and energy efficiency are due to different allocation of chip area
 - ▶ High degree of SIMD parallelism, simple in-order cores, less control/sync. logic, less cache/scratchpad capacity
 - ▶ SIMD is more energy-efficient than MIMD since a single instruction can launch many data operations
 - ▶ Simpler pipeline with no support for restartable instructions and precise exceptions

Limitations of GPUs

Should we not use GPUs ALL the time?

- GPUs can only execute some types of code fast
 - ▶ SIMD parallelism is not well suited for all algorithms
 - ▶ Need lots of data parallelism, data reuse, and regularity
- GPUs are harder to program and tune than CPUs because of their architecture
 - ▶ Fewer tools and libraries exist

Role of CPUs

- CPU is responsible for initiating computation on the GPU and transferring data to and from the GPU
- Beginning and end of the computation typically require access to input/output (I/O) devices
- There are ongoing efforts to develop APIs providing I/O services directly on the GPU
 - ▶ GPUs are not standalone yet, assumes the existence of a CPU

CPUs vs GPUs

CPU

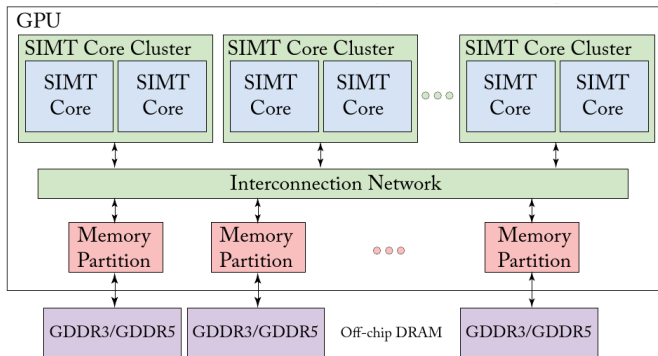
- Designed for running a few potentially complex tasks
 - ▶ Tasks may be unconnected
 - ▶ Suitable to run system software like the OS and applications
- Small number of registers per core private to a task
 - ▶ Context switch between tasks is expensive in terms of time
 - ▶ Register set must be saved to memory and the next one restored from memory

GPU

- Designed for running large number of simple tasks
 - ▶ Suitable for data parallelism
- Has a single set of registers but with multiple banks
 - ▶ A context switch involves setting a bank selector to switch in and out the current set of registers
 - ▶ Orders of magnitude faster than having to save to RAM

GPU Architecture

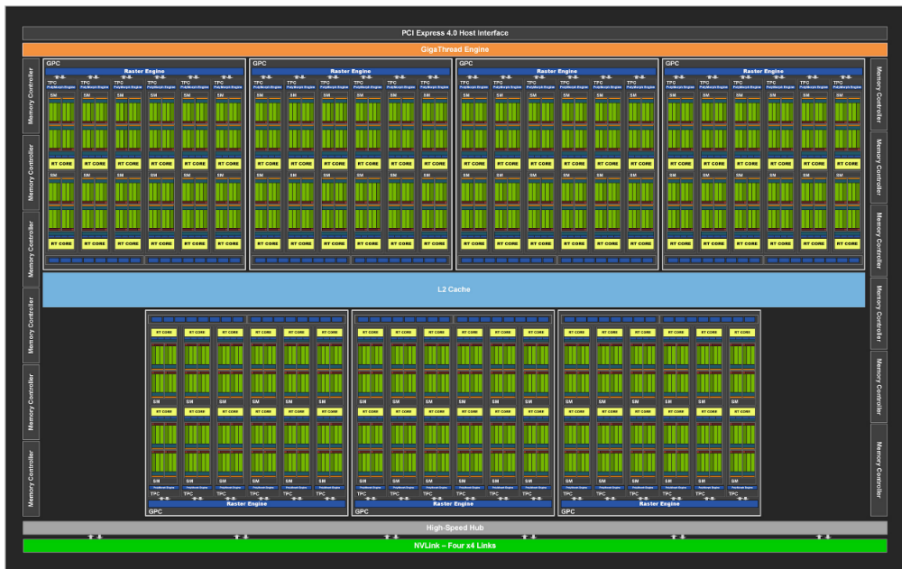
- GPUs consist of Streaming Multiprocessors (SMs)
 - ▶ NVIDIA calls these streaming multiprocessors and AMD calls them compute units
- SMs contain Streaming Processors (SPs) or Processing Elements (PEs)
 - ▶ Each core contains one or more ALUs and FPUs
- GPU can be thought of as a multi-multicore (manycore) system



Ampere Architecture

- A GA102 GPU includes 28.3 billion transistors with a die size of 628.4 mm²
- Includes 7 Graphics Processing Clusters (GPCs), 42 (7*6) Texture Processing Clusters (TPCs), and 84 (7*12) Streaming Multiprocessors (SMs)
- Each SM in GA10x GPUs contain
 - ▶ 128 CUDA cores for a total of $84 \times 128 = 10752$ cores
 - ▶ 256 (4*16384*32 bits) KB register file
 - ▶ Combined 128 KB L1 data cache/shared memory subsystem
- In addition, there are 84 RT Cores, 336 Tensor Cores, 168 FP64 units (two per SM)
- The memory subsystem consists of twelve 32-bit memory controllers (384-bit total)
 - ▶ 512 KB of L2 cache is paired with each 32-bit memory controller, for a total of 6144 KB
- Includes PCIe Gen4 providing up to 16 Gigatransfers/second bit rate

Ampere Architecture



Ampere Architecture



Each SM contains 128 CUDA cores, 4 Tensor cores, 4 Texture units, 1 Ray Tracing core, a 256 KB register file, and 128 KB of L1/Shared memory.

An SM is partitioned into four processing blocks (sub-cores), each with a 64 KB register file, an L0 instruction cache, one warp scheduler, one dispatch unit, and sets of math and other units.

L1 (KB)	Shared memory (KB)
128	0
120	8
112	16
96	32
64	64
28	100

More on Ampere Architecture

- Each SM processing block (also called partition or sub-core) has two datapaths
- One datapath in each partition consists of 16 FP32 CUDA Cores
- The second datapath consists of both 16 FP32 CUDA Cores and 16 INT32 Cores
- A GA102 SM can execute either 128 FP32 operations or 64 FP32 + 64 INT32 operations per clock

- Partitioning the SM allows for better scalability
- Each sub-core can issue instructions independently for its own pipeline, thereby increasing instruction throughput
- Reduces contention on the register file and interconnects, can be independently clock- or power-gated

- A GB202 GPU includes 92.2 billion transistors with a die size of 750 mm²
- Includes 12 Graphics Processing Clusters (GPCs), 96 (12*8) Texture Processing Clusters (TPCs), 192 (12*16) Streaming Multiprocessors (SMs)
- Each SM in GB20x GPUs contain
 - ▶ 128 CUDA cores for a total of $192 \times 128 = 24576$ cores
 - ▶ 256 (4*16384*32 bits) KB register file
 - ▶ Combined 128 KB L1 data cache/shared memory subsystem
- In addition, there are 192 RT Cores, 768 Tensor Cores, and 384 FP64 units (two per SM)
- The memory subsystem consists of 16 32-bit memory controllers (512-bit total)
 - ▶ 8192 KB of L2 cache is paired with each 32-bit memory controller, for a total of 128 MB
- Includes PCIe Gen5 providing up to 32 Gigatransfers/second bit rate

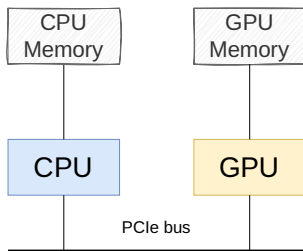
Compute Capability (CC)

- When programming with CUDA, it is very important to be aware of the differences among different versions of hardware
- In CUDA, CC refers to architecture features
 - ▶ For example, number of registers and cores, cache and memory size, and supported arithmetic instructions
- For example, CC 1.x devices have 16 KB local memory per thread, and 2.x and 3.x devices have 512 KB local memory per thread

Discrete vs Integrated GPUs

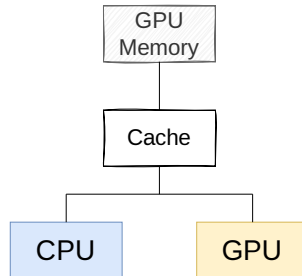
Discrete

- More performant, consumes more energy
- Cost of PCIe transfers influences the granularity of offloading and the performance



Integrated

- Less performant because of energy considerations
- CPU and GPU share physical memory (DRAM or LLC) and can avoid the cost of data transfers over a PCIe bus



CUDA Programming

Programming NVIDIA GPUs

CUDA Philosophy

Massively parallel

The computations can be broken down into hundreds or thousands of independent units of work

Computationally intensive

The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory

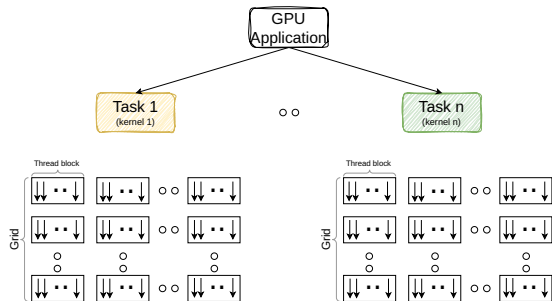
Single Instruction Multiple Thread (SIMT)

CUDA Programming Model

Allows fine-grained data parallelism and thread parallelism nested within coarse-grained data parallelism and task parallelism

- Independent grids (i.e., kernels) represent task parallelism
- Concurrent thread blocks represent coarse-grained data parallelism
- Concurrent threads/warps represent fine-grained data parallelism or thread parallelism

- Partition the problem into coarse sub-problems that can be solved independently
- Assign each sub-problem to a “block” of threads to be solved in parallel
- Each sub-problem is also decomposed into finer work items that are solved in parallel by all threads within the block



Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
    printf("Hello world!\n");
}

int main() {
    hwkernel<<<1, 1>>>();
}
```

```
$ nvcc hello-world.cu
$ ./a.out

$
```

Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
    printf("Hello world!\n");
}

int main() {
    hwkernel<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

```
$ nvcc hello-world.cu
$ ./a.out
Hello world!

$
```

- CPU thread returns immediately after launching the kernel
- Use `cudaDeviceSynchronize()` (or its variants) to block the caller CPU thread

Function Declarations

	Executed on	Callable from
<code>__device__ float deviceFunc()</code>	Device	Device
<code>__global__ void kernelFunc()</code>	Device	Host [§]
<code>__host__ float hostFunc()</code>	Host	Host

- `__global__` defines a kernel function, must return void
- `__device__` functions can have return values
- `__host__` is default, and can be omitted
- Prepending `__host__` `__device__` causes the system to compile separate host and device versions of the function

[§]A kernel function can also be called from the device if dynamic parallelism is enabled

Classical Copy-Then-Execute Model

- (i) Prepare input data in CPU memory
- (ii) Copy data from CPU to GPU memory (e.g., `cudaMemcpy()`)
- (iii) Launch GPU kernel and execute
- (iv) Copy results from GPU memory to CPU memory (e.g., `cudaMemcpy()`)
- (v) Use the results on the CPU
- (vi) Repeat the above steps based on the application logic

Kernels

- Kernels are special functions that a CPU can call to execute on the GPU
 - ▶ Executed N times in parallel by N different CUDA threads
 - ▶ Cannot return a value
- Each thread has a unique thread ID that is accessible within the kernel through the built-in threadIdx variable

```
__global__ void VecAdd(float* A, float* B, float* C) {  
    int i = threadIdx.x;  
    ...  
}  
int main() {  
    auto *h_A = new float[N];  
    ...  
    // Kernel invocation with 1 thread block and N threads per block  
    VecAdd<<<1, N>>>(d_A, d_B, d_C);  
}
```

Kernels

```
KernelName<<<m, n>>>(arg1, arg2, ...)
```

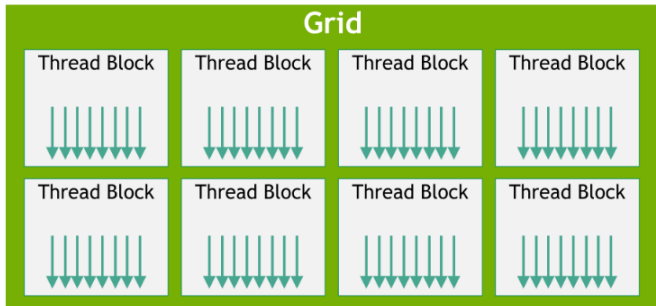
GPU spawns m blocks with n threads that run a copy of the same function

- CPU can continue processing while GPU runs the kernel
- Kernel call returns when all threads have terminated

```
kernel1<<<X,Y>>>(...);  
// kernel starts execution, CPU continues to next statement  
kernel2<<<X,Y>>>(...);  
// kernel2 placed in queue, will start after kernel1 finishes,  
// CPU continues  
cudaMemcpy(...);  
// CPU blocks until memory is copied, memory copy starts after all  
// preceding CUDA calls finish
```

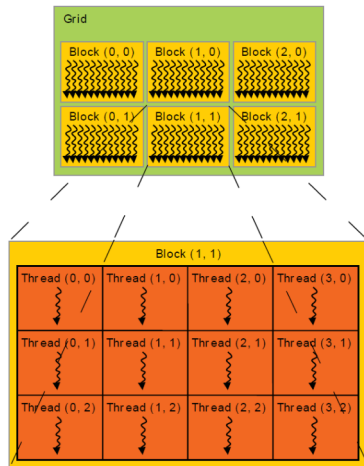
Thread Hierarchy

- A kernel executes in parallel across a set of parallel threads
 - ▶ Threads that are generated by a kernel launch are collectively called a **grid**
- Threads are organized in **thread blocks** (aka cooperative thread arrays)
 - ▶ A thread block is a set of concurrently executing threads that can communicate among themselves through shared memory and barrier synchronization
- A grid is an array of thread blocks that execute the same kernel



Thread Hierarchy

- Threads in a block reside on the same SM, max 1024 threads in a block
- Thread blocks are organized into 1D, 2D, or 3D grids
 - ▶ Grid dimension is given by the `gridDim` variable
- Identify block within a grid with the `blockIdx` variable
- Block dimension is given by the `blockDim` variable



Dimension and Index Variables

Type is dim3

Dimension

- `gridDim` specifies the number of blocks in the grid
- `blockDim` specifies the number of threads in each block
- The keywords are 3-component vectors
 - ▶ For example, `threadIdx.[x,y,z]` gives the index of the thread in a block, in the particular direction

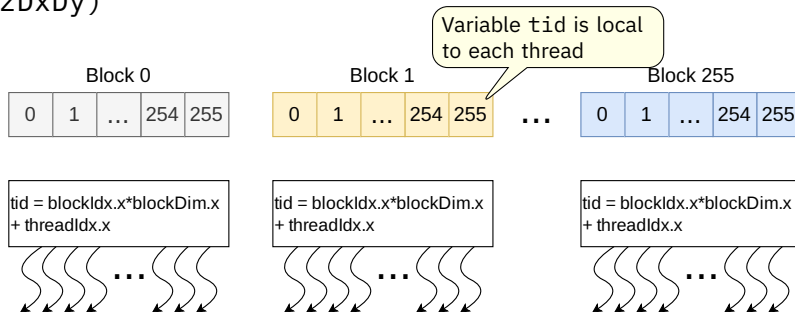
Index

- `blockIdx` gives the index of the block in the grid
- `threadIdx` gives the index of the thread within the block

Finding Thread IDs

How to find out the linearized thread IDs within a block?

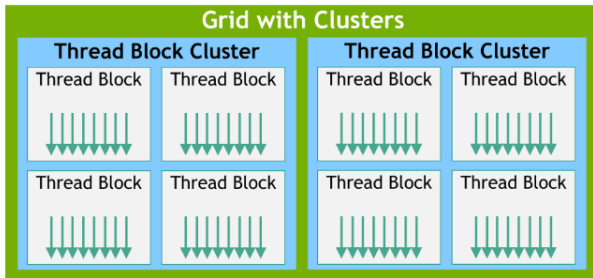
- 1D block: thread ID is `threadIdx.x`
- 2D block of size (D_x, D_y) : thread ID of a thread of index (x, y) is $(x + yD_x)$
- 3D block of size (D_x, D_y, D_z) : thread ID of a thread of index (x, y, z) is $(x + yD_x + zD_xD_y)$



Thread Block Clusters

CC 9.0 introduced an **optional** level in the thread hierarchy called Thread Block Clusters

- Thread blocks in a cluster are always co-scheduled on a GPC
 - ▶ Thread blocks in a cluster can access the distributed Shared Memory
 - ▶ Allows thread blocks in the cluster to synchronize with hardware support
- The number of thread blocks in a cluster can be specified with the compile-time kernel attribute `__cluster_dims__(X,Y,Z)`
- A maximum of 8 thread blocks is guaranteed to be portable



Execution Configuration Uses Integer Arithmetic

- Dimension variables are vectors of integral type
- Assume data is of length N , and say the kernel execution configuration is $\langle\langle N/TPB, TPB \rangle\rangle$
 - ▶ Each block has TPB threads and there are N/TPB blocks

Suppose $N = 64$ and $TPB = 32$

Implies there are 2 blocks of 32 threads

Suppose $N = 65$ and $TPB = 32$

Implies there are 2 blocks of 32 threads

- Ensure that the grid covers the array length by rounding up the number of blocks from N/TPB to $(N+TPB-1)/TPB$
- Use a control statement in the kernel to ensure that the thread index is within the maximum array index

Launching Kernels for 2D Data: Matrix Addition

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main() {
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Timing a CUDA Kernel

```
...  
cudaEvent_t start = nullptr, stop = nullptr;  
cudaEventCreate(&start);  
cudaEventCreate(&stop);  
  
cudaEventRecord(start, 0);  
MatMul<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);  
cudaEventRecord(stop);  
cudaEventSynchronize(stop);  
  
float kernel_time = 0.0F;  
cudaEventElapsedTime(&kernel_time, start, stop);  
  
cudaEventDestroy(start);  
cudaEventDestroy(stop);  
...
```

Choosing Optimal Execution Configuration

- The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system
 - ▶ It is okay to have a greater number of threads
- Choose number of threads in a block to be a multiple of 32
- No fixed rule, needs exploration and experimentation to find the best grid configuration

```
__global__ void MatMul(float* A, float* B, float* C) {  
    ...  
}  
int main() {  
    ...  
    constexpr uint32_t TB_DIM = 64;  
    dim3 threadsPerBlock(TB_DIM, TB_DIM);  
    dim3 blocksPerGrid((N + TB_DIM - 1) / TB_DIM, (N + TB_DIM - 1) / TB_DIM);  
    MatMul<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);  
    ...  
}
```

Reporting Errors

- All CUDA API calls return an error code of type `cudaError_t`
 - ▶ Error in the API call itself or error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error with `cudaGetLastError()`
- Get a string to describe the error with `char *cudaGetErrorString(cudaError_t)`

What is the canonical way to check for errors using the CUDA runtime API?

Query device information

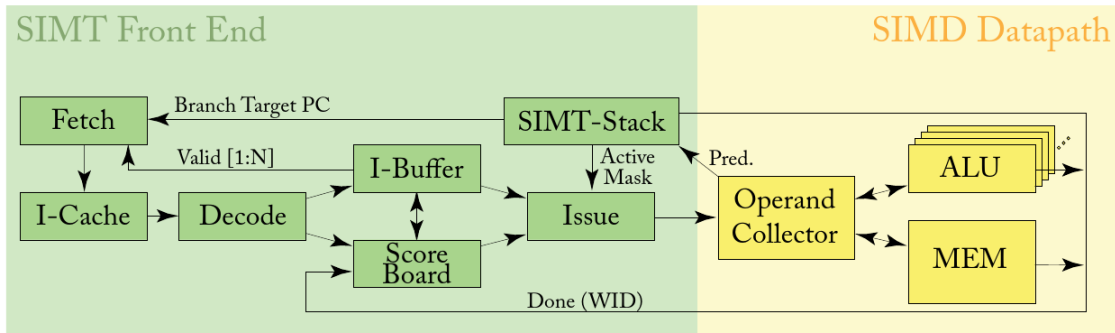
- The application can query and select GPUs
 - ▶ `cudaGetDeviceCount(int *count)`
 - ▶ `cudaSetDevice(int device)`
 - ▶ `cudaGetDevice(int *device)`
 - ▶ `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`
- Multiple host threads can share a device
- A single host thread can manage multiple devices
 - ▶ `cudaSetDevice(i)` to select current device
 - ▶ `cudaMemcpy(...)` for peer-to-peer copies

Warp Scheduling

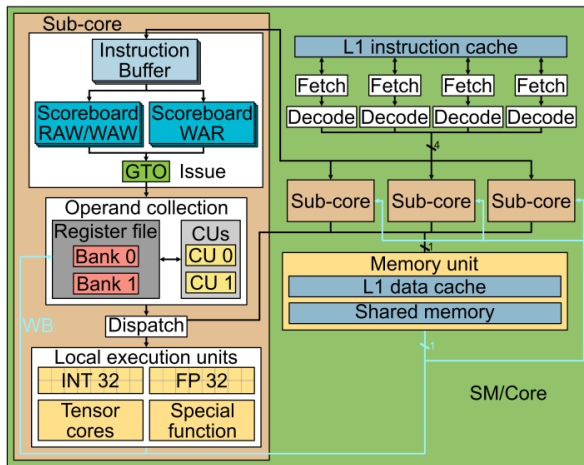
Understanding SIMT

- In vectorization, users program the SIMD hardware directly with vector ISA, or use auto-vectorization or intrinsics
 - ▶ We program with the vector width in mind, excepting auto-vectorization
- GPUs employ SIMD hardware to exploit data-level parallelism
- SIMT can be thought of as “SIMD with multithreading”
 - ▶ Software analog compared to the hardware perspective of SIMD
 - ▶ Distinct threads executing together can access non-contiguous memory or diverge into different execution branches
- Multiple types of parallelism
 - ▶ Independent grids (i.e., kernels) or concurrent thread blocks represent coarse-grained data parallelism or task parallelism
 - ▶ Concurrent threads/warps represent fine-grained data parallelism or thread parallelism

SM Microarchitecture



SM Microarchitecture Modeled in Accel-Sim



M. Khairy et al. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. ISCA 2020.

R. Huerta et al. Dissecting and Modeling the Architecture of Modern GPU Cores. MICRO 2025.

Mapping Blocks and Threads

- A GPU executes one or more kernel grids
- A kernel is partitioned into thread blocks that execute independently of each other
- When a CUDA kernel is launched, the thread blocks are distributed across the SMs
 - ▶ Multiple thread blocks, up to a limit, can execute concurrently on one SM
 - Not all blocks may be resident at the same time
 - ▶ For example, a CUDA device may allow up to eight blocks to be assigned to each SM
 - ▶ CUDA cores in the SM execute threads of a block
- A block begins execution only when it has secured all execution resources necessary for all the threads
- As thread blocks terminate, new blocks are launched on the vacated SMs

CUDA runtime can execute blocks in any order

- Blocks are mostly **not** supposed to synchronize with each other
 - ▶ Allows for simple hardware support for data parallelism

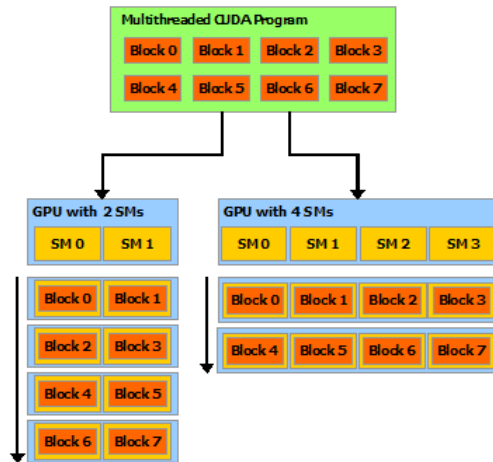
Block Scalability

- A kernel with enough blocks scales across GPUs
- A GPU with more SMs will automatically execute the program faster than a GPU with fewer SMs

Comparing the performance of vector addition kernel

GPU	CC	# SMs	Time (ms)
GeForce GTX 1080	6.1	20	72.5
Quadro RTX 5000	7.5	48	8.6
A40	8.6	84	5.6

 vector-addition.cu



Thread Warps

- Conceptually, threads in a block can also execute in any order
- However, sharing a control unit reduces hardware complexity, cost, and power consumption
- A set (currently 32) of consecutive threads that execute in SIMD fashion is called a **warp**
 - ▶ Called wavefront (with 64 threads) on AMD GPUs
- Warps are scheduling units in an SM
 - ▶ Implementation detail, not part of the programming model
- Warps issue their instructions in program order

Scheduling Thread Warps

- Each SM launches warps of threads, and executes warps on a time-sharing basis
- SM schedules and executes warps that are ready to run
 - ▶ Warps run for fixed-length time slices like processes
 - ▶ Warps whose next instruction has its operands ready for consumption are eligible for execution
 - ▶ If more than one warp is ready for execution, a greedy-then-youngest* priority mechanism is used to select one for execution
 - Suppose an instruction to be executed by a warp has to wait for the result of a previously initiated long-latency operation (e.g., `ld.global`)
 - The warp is not selected for execution, another warp that is not waiting for results is selected for execution
 - ▶ Selection of ready warps for execution does not introduce any idle time into the execution timeline (i.e., zero-overhead scheduling)

* R. Huerta et al. Dissecting and Modeling the Architecture of Modern GPU Cores. MICRO 2025.

Scheduling Thread Warps

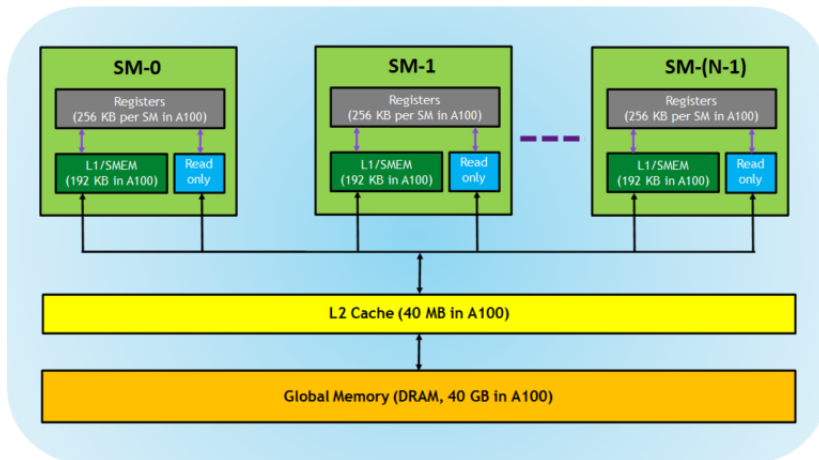
- Goal is to have enough warps so that the GPU scheduler is likely to always find a warp ready to execute
 - ▶ Hides latency of long memory operations with work from other threads (called latency tolerance or hiding)
- Thread blocks execute on an SM, thread instructions execute on a core
 - ▶ CUDA virtualizes the physical hardware
 - ▶ Thread is a virtualized scalar processor (registers, PC, state)
 - ▶ Block is a virtualized multiprocessor (threads, shared memory)
- As warps and thread blocks complete, resources are freed

Warp Scheduling

- For an ideal memory system, increasing the number of warps can increase the throughput
 - ▶ If the number of warps in a core multiplied by the issue time of each warp exceeds the memory latency, the execution units in the core will always remain busy
 - ▶ Round-robin scheduling of warps would suffice
 - ▶ However, locality property either favors or discourages round-robin scheduling
- Increasing the number of warps is impractical because of the need to maintain execution state in hardware (i.e., to achieve zero-overhead scheduling)
 - ▶ Number of threads that can be simultaneously tracked and scheduled in hardware is bounded
 - ▶ Requires resources for an SM to maintain execution status of threads
- Up to 2048 threads can be assigned to each SM on recent CUDA devices
 - ▶ For example, 8 blocks of 256 threads, or 4 blocks of 512 threads
- Assume a CUDA device with 28 SMs, where each SM can accommodate up to 2048 threads
 - ▶ The device can have up to 57344 threads simultaneously **residing** in the device for execution

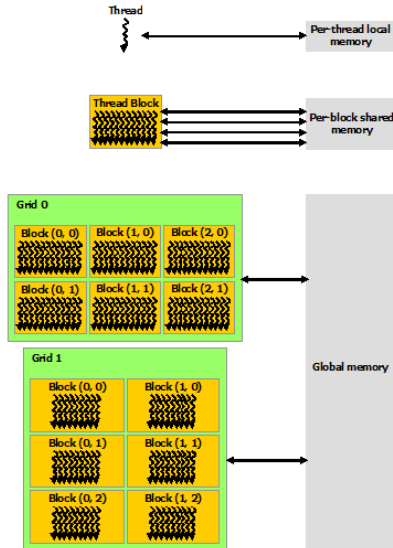
Memory Hierarchy

Memory Hierarchy in NVIDIA A100



Memory Organization

- Host and device maintain their own separate memory spaces
 - ▶ A variable in CPU memory may not be accessed directly in a GPU kernel
- It is the programmer's responsibility to keep copies of variables in sync



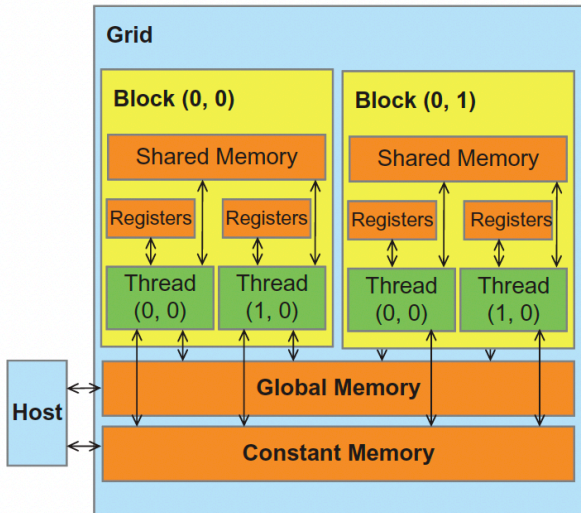
Memory Hierarchy in CUDA

Device code can:

- R/W per-thread **registers**
- R/W per-thread **local memory**
- R/W per-block **shared memory**
- R/W per-grid **global memory**
- Read only per-grid **constant memory**

Host code can

- Transfer data to/from per grid **global** and **constant** memories



Variable Type Qualifiers in CUDA

	Memory	Scope	Lifetime
<code>int localVar</code>	Register	Thread	Kernel
<code>__device__ __local__ int localVar</code>	Local	Thread	Kernel
<code>__device__ __shared__ int sharedVar</code>	Shared	Block	Kernel
<code>__device__ int globalVar</code>	Global	Grid	Application
<code>__device__ __constant__ int constVar</code>	Constant	Grid	Application

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`
- CPU pointers can only point to memory allocated or declared in global memory

Registers

- 64K 32-bit registers (or 256 KB register file) per SM
 - ▶ A CPU in contrast has a few (1–2 KB) per core
- Registers are allocated per warp, in chunks of 256
- Up to 255 registers per thread (CC 3.5+)
- If a code uses the maximum number of registers per thread (255) and an SM has 64K registers, then the SM can support a maximum of 256 threads
- If we use the maximum allowable number of threads per SM (2048), then each thread can use at most 32 registers per thread
- CUDA provides kernel qualifiers for passing additional information to nvcc for more efficient register usage (e.g., `__launch_bounds__()` and `__maxnreg__()`)

Registers

- 64K 32-bit registers (or 256 KB register file) per SM
 - ▶ A CPU in contrast has a few (1–2 KB) per core
- Registers are allocated per warp, in chunks of 256
- Up to 255 registers per thread (CC 3.5+)
- If a code uses the maximum number of registers per thread (255) and an SM has 64K registers
- If we use the maximum allowable number of threads per SM (2048), then each thread can use at most 32 registers per thread
- CUDA provides kernel qualifiers for passing additional information to nvcc for more efficient register usage (e.g., `__launch_bounds__()` and `__maxnreg__()`)

What if each thread uses 33 registers?

Shared Memory

- Each SM contains a shared memory structure that acts as a software-managed cache
- Shared memory (also called scratchpad memory) is used for efficient communication among threads in a block
 - ▶ Usually 16–100 KB of storage that can be accessed efficiently by all threads in a block
 - ▶ The space is shared among all blocks running on that SM

Variable in shared memory is allocated using the `__shared__` specifier

- Latency is comparable to accessing registers

Say an SM with 4 thread blocks has 16 KB of shared memory

```
__shared__ float min[256];  
__shared__ float max[256];  
__shared__ float avg[256];  
__shared__ float stdev[256];
```

Several things like grid configuration, number of registers, and amount of shared memory per block limits occupancy

Dynamic Shared Memory

Kernels using shared memory over 48 KB per block must use dynamic shared memory

- Allocated at run time, and the size is specified as the third argument to the kernel
 - ▶ Implies only one extern `__shared__` array can be declared inside a kernel

```
__global__ void kernel(uint32_t dbl_arr_sz, uint32_t ch_arr_sz) {  
    extern __shared__ char array[];  
    double *dbl_array = (double *)array;  
    char *ch_array = array + dbl_arr_sz * sizeof(double);  
    ...  
}  
int main() {  
    ...  
    uint32_t DBL_ARR_SZ = 256, CH_ARR_SZ = 128;  
    uint32_t SHD_MEM_SZ = DBL_ARR_SZ * sizeof(double) + CH_ARR_SZ * sizeof(char);  
    kernel<<<1,1, SHD_MEM_SZ>>>(DBL_ARR_SZ, CH_ARR_SZ);  
    ...  
}
```



Global Memory

- Variable lock can be accessed by both kernels

- ▶ Resides in global memory space
- ▶ Can be both read and modified by all threads

```
__device__ int lock=0;
__global__ void kernel1(...) {
    // Kernel code
}
__global__ void kernel2(...) {
    // Kernel code
}
```

- On-device memory is accessed via 32, 64, or 128 B transactions
- A warp executes an instruction that accesses global memory
 - ▶ The addresses are coalesced into transactions
 - ▶ Number of transactions depend on the access size and distribution of memory addresses
 - ▶ More transactions mean less throughput
 - For example, if 32 B transaction is needed for a thread's 4 B access, throughput is essentially 1/8th

Memory Access Efficiency

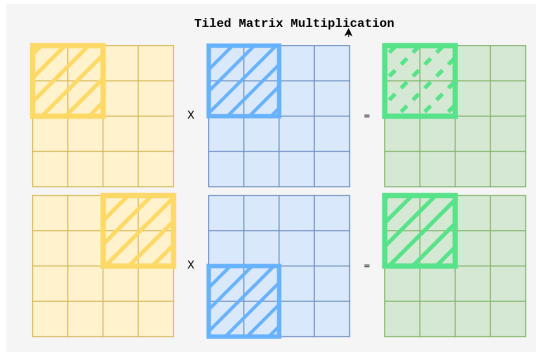
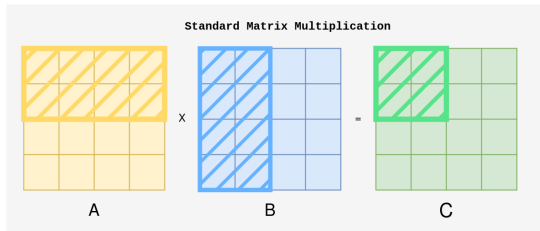
Compute-to-global-memory access ratio is the number of floating-point operations performed for each access to global memory

```
// A and B are pointers to float type  
for (uint32_t k = 0; k < N; k++)  
    tmp += A[row*N+k]*B[k*N+col]);
```

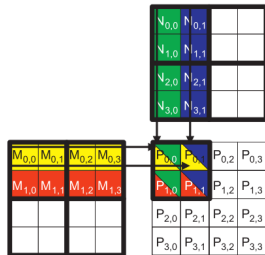
Assume a GPU device with 1 TB/s global memory bandwidth and peak single-precision performance of 12 TFLOPS

- What is the performance we expect with an access ratio of 1?
- We can do $1000/(4+4)$ GFLOPS, which is only $\sim 1\%$ of the peak performance

Improving Matrix Multiplication with Tiling in Shared Memory



Improving Matrix Multiplication with Tiling in Shared Memory



Phase 1

Thread _{0,0}	$M_{sh}[0][0] = M[0][0]$	$N_{sh}[0][0] = N[0][0]$	$tmp_{0,0} += (M_{sh}[0][0]*N_{sh}[0][0] + M_{sh}[0][1]*N_{sh}[1][0])$
Thread _{0,1}	$M_{sh}[0][1] = M[0][1]$	$N_{sh}[0][1] = N[0][1]$	$tmp_{0,1} += (M_{sh}[0][0]*N_{sh}[0][1] + M_{sh}[0][1]*N_{sh}[1][1])$
Thread _{1,0}	$M_{sh}[1][0] = M[1][0]$	$N_{sh}[1][0] = N[1][0]$	$tmp_{1,0} += (M_{sh}[1][0]*N_{sh}[0][0] + M_{sh}[1][1]*N_{sh}[1][0])$
Thread _{1,1}	$M_{sh}[1][1] = M[1][1]$	$N_{sh}[1][1] = N[1][1]$	$tmp_{1,1} += (M_{sh}[1][0]*N_{sh}[0][1] + M_{sh}[1][1]*N_{sh}[1][1])$

Phase 2

Thread _{0,0}	$M_{sh}[0][0] = M[0][2]$	$N_{sh}[0][0] = N[2][0]$	$tmp_{0,0} += (M_{sh}[0][0]*N_{sh}[0][0] + M_{sh}[0][1]*N_{sh}[1][0])$
Thread _{0,1}	$M_{sh}[0][1] = M[0][3]$	$N_{sh}[0][1] = N[2][1]$	$tmp_{0,1} += (M_{sh}[0][0]*N_{sh}[0][1] + M_{sh}[0][1]*N_{sh}[1][1])$
Thread _{1,0}	$M_{sh}[1][0] = M[1][2]$	$N_{sh}[1][0] = N[3][0]$	$tmp_{1,0} += (M_{sh}[1][0]*N_{sh}[0][0] + M_{sh}[1][1]*N_{sh}[1][0])$
Thread _{1,1}	$M_{sh}[1][1] = M[1][3]$	$N_{sh}[1][1] = N[3][1]$	$tmp_{1,1} += (M_{sh}[1][0]*N_{sh}[0][1] + M_{sh}[1][1]*N_{sh}[1][1])$

Constant Memory

- Used for data that will not change during kernel execution
 - ▶ Accessible from all threads within a grid
 - ▶ Constant memory is 64 KB

```
// Global scope  
__constant__ float d_filter[FILTER_WIDTH];  
// Initialize array in constant memory  
cudaMemcpyToSymbol(d_filter, h_filter, FILTER_WIDTH * sizeof(float));
```

- Constant memory is cached
 - ▶ Each SM has a read-only constant cache that is shared by all cores in the SM
 - ▶ Speeds up reads from the constant memory space which resides in GPU global memory
 - ▶ Read from constant memory will incur a global memory latency on the first miss
 - ▶ Subsequent reads will hit in the constant cache, which is almost as fast as registers

Local Memory

- Local memory is off-chip per-thread memory
 - ▶ More like thread-local global memory, so it requires memory transactions and consumes bandwidth
- Automatic variables are placed in local memory
 - (i) Arrays when it is not known whether indices are constant quantities
 - (ii) Large structures or arrays that consume too much register space
 - (iii) In case of register spilling
- Inspect PTX assembly code (compile with `-ptx`)
 - ▶ Check for `ld.local` and `st.local` mnemonic

Device Memory Management

- Global device memory can be allocated with `cudaMalloc()`
- Freed by `cudaFree()`
- Data transfer between host and device is with `cudaMemcpy()`
- Initialize memory with `cudaMemset()`
- There are asynchronous versions (e.g., `cudaMemcpyAsync()`)
- Check the CUDA Programming Guide[§] for additional APIs

[§]CUDA C++ Programming Guide: Device Memory

GPU Caches

- GPUs have private L1 and shared L2 data caches
 - ▶ Texture and constant cache are available on all devices
- L1 and L2 cache lines are 128 B wide
- L1 caches in GPUs are **not** coherent, L2 cache is coherent
- Shared memory is partitioned out of unified data cache and its size can be configured, remaining portion is the L1 cache
- On a global store, the L1 cache uses a write-through policy
- L2 cache is shared by all SMs
- Uses a writeback policy for global memory stores

More on GPU Caches

- GPU caches are **sectored**, where each sector is 32 B
- There is one tag for the entire cache line, but each sector has its own status bits (e.g., valid and dirty)
- If only one 32 B sector is modified, only that sector is written back
- On a cache miss, not all 4 sectors in the cache line have to be filled, just the ones that need updating
- + Reduces tag array overhead and saves bandwidth in case of scattered accesses

Cache Operators on Memory Instructions

- Accesses to global memory are cached both in L1 and L2 caches by default
- Use compile time options for finer control on the caching behavior
- Load from global memory
 - `d1cm=ca` loads are cached in both L1 and L2 (default), default eviction policy
 - `d1cm=cg` loads bypass L1 and are only cached in L2
 - `d1cm=cs` hints that data accessed is streaming and marks the line for early eviction
 - `d1cm=cv` refetches data from global memory on every load, bypassing L1 and invalidating L2
 - `d1cm=lu` hints that this is the final load for this address, marking it for early eviction and avoids needless write-backs
- Store to global memory
 - `dscm=wb` bypass or invalidate L1, L2 uses writeback, default eviction policy
 - `dscm=cg` cache only in L2, bypassing L1
 - `dscm=cs` marks the line for early eviction
 - `dscm=wt` writes through the L2 cache to global memory

Configuring Shared Memory Size

- The portion of shared memory in the unified data cache is configurable per kernel
- The driver automatically configures the desired shared memory capacity for each kernel to avoid occupancy bottlenecks
 - ▶ For Volta, the unified data cache is of 128 KB, and the shared memory capacity can be set to 0, 8, 16, 32, 64 or 96 KB
 - ▶ For Turing, the unified data cache has a size of 96 KB, and the shared memory capacity can be set to either 32 KB or 64 KB
- An user can provide hints with `cudaFuncSetAttribute()`

CUDA Compilation

Binary compatibility of GPU applications is not guaranteed across different generations

💡 **Nvcc is a driver program based on LLVM**

- Compiles and links all input CUDA files
- Requires a general-purpose C/C++ host compiler for the host code
 - ▶ Uses GCC/G++ by default on Linux platforms

❗ **High-level overview of the compilation steps**

- (i) Input program is preprocessed for device compilation
- (ii) The device code is compiled to a CUDA binary (.cubin) and/or PTX (Parallel Thread Execution) assembly code, which are encoded in a fatbinary
- (iii) Input program is processed for compilation of the host code
- (iv) Host code and the embedded fatbinary are linked together to generate the executable

NVCC Compilation Workflow

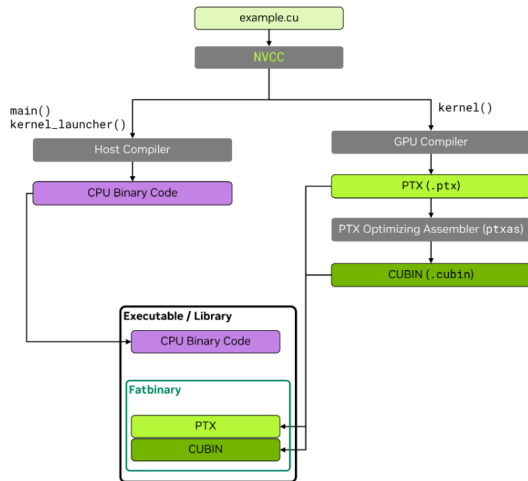
```
#include <iostream>

__global__ void kernel() {
    printf("Hello from kernel\n");
}

void kernel_launcher() {
    kernel<<<1, 1>>>();
    cudaDeviceSynchronize();
}

int main() {
    kernel_launcher();
    return 0;
}
```

```
$ nvcc hello-kernel.cu
```



Disassemble the Binaries

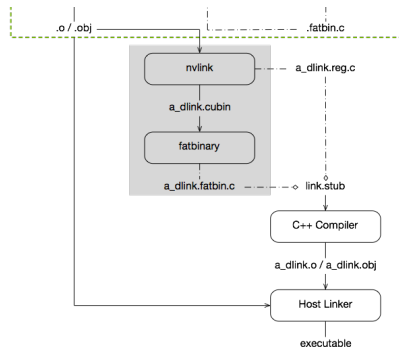
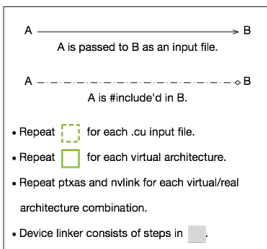
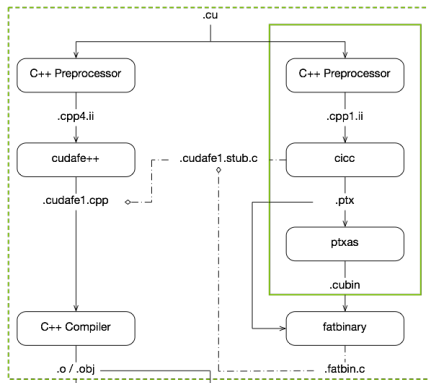
```
$ nvcc -arch=compute_86 hello-kernel.cu  
$ cuobjdump a.out
```

```
swarnendu@gpu3:~/cs623-2026-27-1/compilation$ cuobjdump bin/hello-kernel-compute.out  
  
Fatbin ptx code:  
=====  
arch = sm_86  
code version = [8,5]  
host = linux  
compile_size = 64bit  
compressed  
ptxasOptions =  
swarnendu@gpu3:~/cs623-2026-27-1/compilation$ |
```

```
$ nvcc -arch=sm_86 hello-kernel.cu  
$ cuobjdump a.out
```

```
swarnendu@gpu3:~/cs623-2026-27-1/compilation$ cuobjdump bin/hello-kernel-sm.out  
  
Fatbin elf code:  
=====  
arch = sm_86  
code version = [1,7]  
host = linux  
compile_size = 64bit  
  
Fatbin elf code:  
=====  
arch = sm_86  
code version = [1,7]  
host = linux  
compile_size = 64bit  
  
Fatbin ptx code:  
=====  
arch = sm_86  
code version = [8,5]  
host = linux  
compile_size = 64bit  
compressed  
ptxasOptions =  
swarnendu@gpu3:~/cs623-2026-27-1/compilation$ |
```

CUDA Compilation Trajectory



High-Level Steps in Compilation

Preprocessor The input CUDA file containing host and device functions is preprocessed the host C/C++ preprocessor

Host/device split `cudafe++` splits the input to host and device code. For the host, it extracts all CPU code, replaces CUDA-specific syntax (i.e., `<<<>>>`) with C function calls, and outputs a clean `.cpp` file that is compiled by the host compiler. For the device path, it extracts all `__global__` and `__device__` code, producing a device-only C++ file.

Compilation `cicc` translates device C++ code to PTX code. It uses NVVM IR for optimizations.

Assemble `ptxas` compiles the PTX assembly to a GPU-specific cubin file.

Linking `nvcc` embeds the CUBIN and PTX code in a fatbinary file, alongside the host-side compiled C++ object code.

Important NVCC Options

Options	Description
-std {c++17 . . . }	Select a particular C++ dialect
-G	Generate debug information for device code, turns off device optimizations
-ccbin	Specify the directory of the host compiler executable
-cuda	Generate host file that can be compiled by a host compiler
-ptx	Generate PTX code from CUDA source
-cubin	Generate Cubin from CUDA or PTX source
-fatbin	Generate fat binary from CUDA source, PTX, or cubin files
-arch ARCH	Specify the virtual GPU architecture for generating assembly
-code CODE	Specify the real GPU architecture to assemble and optimize for
-Xcompiler	Pass options to the host compiler
-Xptxas	Pass options to the PTX assembler

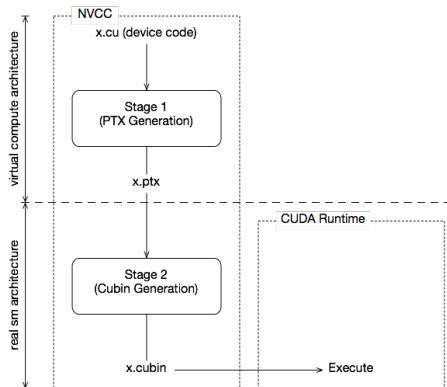
Binary Application Compatibility

- Binaries compiled with nvcc are compatible within a major CC version if the minor version is the same or higher
 - ▶ A cubin compiled for sm_86 can be loaded on any GPU with CC sm_8x where $x \geq 6$
 - ▶ The cubin compiled for sm_86 cannot be loaded on a device with CC 8.0
- NVIDIA does not guarantee **binary** compatibility across GPU generations
 - ▶ A cubin compiled for sm_86 will not load and run on Hopper or later architectures (i.e., CC 9.0 onward)
 - ▶ Instruction set and instruction encodings of a generation may differ from other generations

Two-Stage Compilation with Virtual and Real Architectures

nvcc relies on a two stage compilation model for ensuring application compatibility across GPU generations

- (i) Code is compiled to a virtual assembly called PTX
- (ii) PTX code is assembled for a real GPU architecture



Improving Application Portability with JIT Compilation

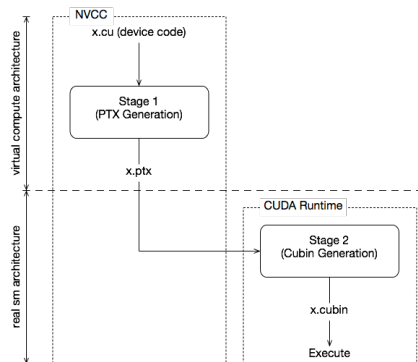
Two-stage compilation does not help improve application portability

- Generated code is bound to one GPU generation (e.g., sm_75)

Strategies to improve portability

Embed PTX Postpone assembly of PTX code until application run time if only a virtual GPU architecture is specified

Embed multiple cubins Generate multiple cubins for possible target architectures and embed them in a fat binary



PTX Assembly

- GPU compilation is performed via an intermediate representation called PTX^{*}
 - ▶ PTX is an acronym for Parallel Thread eXecution
 - ▶ Goal is to provide an architecture-indent ISA for compilers
 - ▶ Can be considered as assembly for a virtual GPU architecture
- PTX code can also be inlined into CUDA with asm[†]
- Understanding PTX is important for rigorous performance analysis of kernels

...DeepSeek actually had an excess of computing ...This is actually impossible to do in CUDA. DeepSeek engineers had to drop down to PTX, a low-level instruction set for NVIDIA GPUs that is basically like assembly language. ...

– Ben Thompson, Stratechery, 2025.

^{*}CUDA PTX ISA

[†]Inline PTX Assembly in CUDA

Understanding PTX Assembly

opcode[.modifier].type destination, source1, source2, ...;

Generate PTX code with -ptx option 

- Memory regions in PTX are explicit
- PTX also allows cache policy hints for memory operations

%tid.x	thread index within block
%ntid.x	block dimension
%ctaid.x	block index
%nctaid.x	grid dimension
%laneid	lane index within warp
%warpid	warp index within SM context

Predication avoids changing control flow for small conditions

```
setp.lt.s32 %p1, %r1, %r2;  
@%p1 add.s32 %r3, %r3, 1;
```

Branches are required for large control flow regions and loops

Examples of PTX Instructions

add.s32 %r1, %r2, %r3;	
mul.lo.s32 %r1, %r2, %r3;	// Integer multiplication, keep low 32 bits
mad.lo.s32 %r1, %r2, %r3, %r4;	// Multiply, keep low 32 bits, add
fma.rn.f32 %f1, %f2, %f3, %f4;	// Fused multiply-add
setp.eq.s32 %p1, %r1, %r2;	
selp.s32 %r1, %r2, %r3, %p1;	// r1 = p1 ? r2 : r3
and.b32 %r1, %r2, %r3;	
shr.u32 %r1, %r2, 3;	// Unsigned shift right
ld.global.f32 %f1, [%rd1];	// Other memory spaces: shared, local, constant
st.shared.f32 [%rd1], %f1;	// Other memory spaces: global, local, constant
mov.f32 %f1, %f2;	
ld.global.ca.f32 %f1, [%rd1];	// Cache at both L1 and L2
ld.global.cg.f32 %f1, [%rd1];	// Cache only at L2
ld.global.cs.f32 %f1, [%rd1];	// Streaming, mark for early eviction
ld.global.cv.f32 %f1, [%rd1];	// Do not cache, fetch from global memory

Understanding -arch and -code

```
nvcc -arch=compute_86 hello-world.cu
```

Generates only PTX for CC 8.6 and embeds in the binary

```
nvcc -arch=compute_86 -code=sm_86 hello-world.cu
```

Generates SASS code compliant with CC 8.6

```
nvcc -arch=sm_86 hello-world.cu
```

- Implies `nvcc -arch=compute_86 -code=sm_86,compute_86 hello-world.cu`
- Embeds both the PTX and the SASS code for CC 8.6 in the binary

```
nvcc -arch=compute_75 -code=sm_86 hello-world.cu
```

Generates SASS code compliant with CC 8.6

```
nvcc -arch=compute_75 -code=sm_75,sm_86 hello-world.cu
```

Generates SASS for the two GPU architectures and embeds the cubin files in the executable

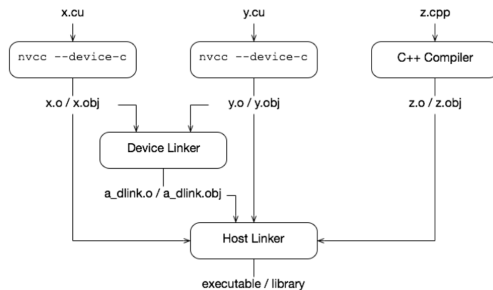
Separate Compilation in CUDA

- CUDA releases prior to version 5 did not support separate compilation
 - ▶ CUDA code could not call device functions or access variables across files
 - ▶ nvcc expected all GPU code and symbols to be present in the compilation unit that uses them
 - ▶ Such compilation is referred to as whole program compilation
- Newer CUDA releases support separate compilation, but whole program compilation is still the default
 - ▶ Use `extern` and `static` to control the visibility of symbols
- Separate compilation allows more flexible code structure, reduces compile time, and can lead to smaller binaries

Separate Compilation in CUDA

Choice 1: Compile source files into standalone intermediate object files containing relocatable device code without linking, and then perform device linking with nvlink and host linking

```
$ nvcc -arch=sm_86 -dc a.cu -o a.o  
$ nvcc -arch=sm_86 -dc b.cu -o b.o  
$ nvcc -arch=sm_86 a.o b.o -o a-dc.out  
$ ./a-dc.out
```



Choice 2: Generate relocatable code and perform linking

```
$ nvcc -arch=sm_86 -rdc=true a.cu b.cu -o a-rdc.out  
$ ./a-rdc.out
```



a.cu



b.h



b.cu

- SASS is the low-level instruction representation corresponding to the machine code executed on NVIDIA GPUs
 - ▶ SASS is most possibly an acronym for Source and ASsembly
 - ▶ SASS is poorly documented, and there have been numerous attempts to reverse engineer
 - ▶ Instruction scheduling, register allocation, scoreboard management, loop fusion and unrolling happens on SASS code
- CUDA includes two binary utilities for examining cubin files
 - `cuobjdump` Parses both cubin files and host binaries
 - `nvdiasm` Only parses cubin files but provides richer output

```
$ cuobjdump -sass vector-addition.cubin  
$ nvdiasm vector-addition.cubin
```

Synchronization in CUDA

Data Races in GPU Code

- Two accesses from two different threads **conflict** when they access the same shared variable where at least one access is a write
- Accesses are not ordered by synchronization operations are **concurrent** (i.e., can happen at the same time)

Data race

A pair of concurrent and conflicting accesses form a data race

A data race can occur between threads within a single warp, warps in the same block, threads in different blocks, threads in different kernels, threads in different GPUs, or between CPU and GPU threads

Examples of Data Races

```
__global__ void intrawarp(int *d) {  
    d[0] = threadIdx.x;  
}
```

```
__global__ void interwarp(int *gc) {  
    *gc = *gc + 1;  
}
```

Data races are considered undefined behavior in CUDA

The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.

- John Regehr

Data Races are Evil!

Transformations possible by a hypothetical optimizing compiler

```
__global__ void racyKernel(int *ret) {  
    __shared__ int sum;  
    sum = 0;  
    __syncthreads();  
  
    for (int i = 0; i < 4; i++)  
        sum += 1; // atomicAdd(&sum, 1);  
}  
  
__syncthreads();  
  
// Expected sum == (4 * blockDim.x)  
if (threadIdx.x == 0)  
    ret[blockIdx.x] = sum;  
}
```

1

```
__global__ void optRacyKernel1(int *ret) {  
    int sum = 0;  
    for (int i = 0; i < 4; i++)  
        sum += 1;  
    if (threadIdx.x == 0)  
        ret[blockIdx.x] = sum;  
}
```

2

```
__global__ void optRacyKernel2(int *ret) {  
    if (threadIdx.x == 0)  
        ret[blockIdx.x] = 4;  
}
```

Intra-Block Synchronization

- `__syncthreads()` waits until all threads in the thread block have reached this point
- All global and shared memory accesses made by the participating threads prior to `__syncthreads()` are visible to all threads in the block
- A `__syncthreads()` statement must be executed by **all** threads in a block
 - ▶ If `__syncthreads()` is in an `if` statement, then either all threads in the block execute the then path that includes the `__syncthreads()` or none of them does
 - ▶ If `__syncthreads()` statement is in each path of an `if-then-else` statement, then either all threads in a block execute the `__syncthreads()` on the then path or all of them execute the `else` path
 - The two `__syncthreads()` are different barrier synchronization points



Data Races from PTX Perspective

Data race

Two conflicting memory operations that are not related in causality order and are not morally strong

- Morally **strong** memory operations establish memory ordering
 - ▶ Morally strong operations have one of the following opcode qualifiers: `.relaxed`, `.acquire`, `.release`, `.acq_rel`, `.volatile`, or `.mmio`
- **Weak** memory operations use the `.weak` qualifier and indicate a memory instruction with no synchronization
 - ▶ Do not impose ordering constraints relative to other threads
 - ▶ A weak store by one thread is not guaranteed to become visible to another thread without the help of additional synchronization or morally strong operations

Incorrect Producer-Consumer Interaction in PTX

Producer Thread

```
mov.u32 %r1, 42;  
// Morally weak store  
st.global.u32 [%rd1], %r1;  
  
mov.u32 %r2, 1;  
st.global.u32 [%rd2], %r2;
```

Consumer Thread

```
LOOP:  
// Morally weak load  
ld.global.u32 %r3, [%rd2];  
setp.eq.u32 %p1, %r3, 0;  
@%p1 bra LOOP;  
  
ld.global.u32 %r4, [%rd1];
```

Scoped Synchronization

- CUDA provides scope qualifiers that limit the subset of the threads that are guaranteed to observe the synchronization
- CUDA exposes three scopes: block, device, and system
- For example, an atomic RMW operation with block scope (e.g., `atomicAdd_block()`) is only visible to threads within the same thread block

Correct Producer-Consumer Interaction in PTX

Producer Thread

```
mov.u32 %r1, 42;  
st.global.u32 [%rd1], %r1;  
  
mov.u32 %r2, 1;  
// Morally strong store  
st.global.release.gpu.u32 [%rd2], %r2;
```

Consumer Thread

```
LOOP:  
// Morally strong load  
ld.global.acquire.gpu.u32 %r3, [%rd2];  
setp.eq.u32 %p1, %r3, 0;  
@%p1 bra LOOP;  
  
ld.global.u32 %r4, [%rd1];
```

Atomic Operations

Atomic operations allow performing atomic read-modify-write (RMW) operations on one 32-bit, 64-bit, or 128-bit data residing in global or shared memory

- Prevent data races associated with multiple threads concurrently accessing a global or shared memory variable
- For example, `atomicAdd()`, `atomicSub()`, `atomicMin()`, `atomicMax()`, `atomicInc()`, `atomicDec()`, `atomicExch()`, `atomicCAS()`

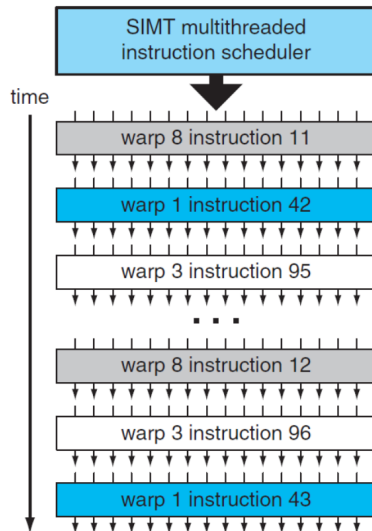
Atomic functions do not act as memory fences and do not imply synchronization or ordering constraints for memory operations

Possible lock implementation

```
*mutex=0;  
while(!atomicCAS(mutex, 0, 1)) {}  
// critical section  
atomicExch(mutex, 0);
```

Lockstep Execution

- All threads in a warp run in lockstep
- Warps share an instruction stream, i.e., same instruction is fetched for all threads in a warp during the instruction fetch cycle
 - ▶ Prior to Volta, warps used a single shared program counter
- In the execution phase, each thread will either execute the instruction or will execute nothing
- Individual threads in a warp have their own instruction address counter and register state
- Warp threads are fully synchronized, i.e., there is an implicit barrier after each step/instruction



Thread Divergence

- If some threads take the then branch and other threads take the else branch, they cannot operate in lockstep
 - ▶ Some threads must wait for the others to execute, serializes execution at that point
- The programming model does not prevent thread divergence
- Divergence occurs only within a warp

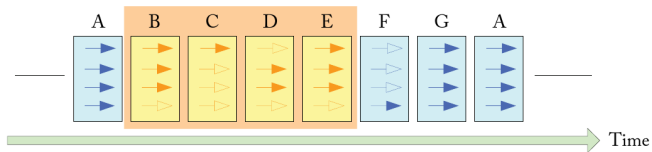
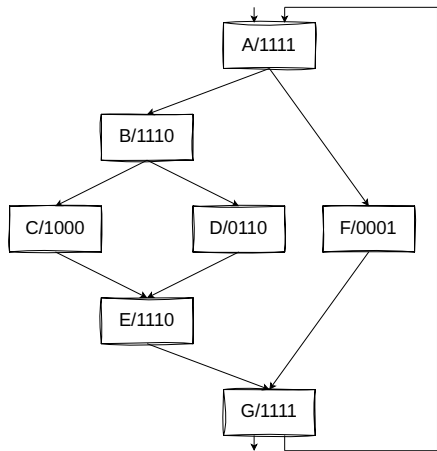
```
if (threadIdx.x < 16)
    A[threadIdx.x] = 1;
else
    A[threadIdx.x] = 2;
```

Example Code and Corresponding PTX

```
do {  
    t1 = tid * N;           // A  
    t2 = t1 + i;  
    t3 = data1[t2];  
    t4 = 0;  
    if (t3 != t4) {  
        t5 = data2[t2];    // B  
        if (t5 != t4) {  
            x += 1;        // C  
        } else {  
            y += 2;        // D  
        }  
    } else {  
        z += 3;           // F  
    }  
    i++;                   // G  
} while (i < N);
```

```
A:    mul.lo.u32    t1, tid, N;  
      add.u32      t2, t1, i;  
      ld.global.u32 t3, [t2];  
      mov.u32      t4, 0;  
      setp.eq.u32   p1, t3, t4;  
@p1   bra          F;  
B:    ld.global.u32 t5, [t2];  
      setp.eq.u32   p2, t5, t4;  
@p2   bra          D;  
C:    add.u32      x, x, 1;  
      bra          E;  
D:    add.u32      y, y, 2;  
E:    bra          G;  
F:    add.u32      z, z, 3;  
G:    add.u32      i, i, 1;  
      setp.le.u32   p3, i, N;  
@p3   bra          A;
```

SIMT Stack-based Execution



SIMT Stack-based Execution

How does GPU hardware enable threads within a warp to follow different code paths?

- A per-warp stack is used to manage divergent control flow
- Only threads at the top of the stack are eligible for scheduling

TOS	Reconv PC	Next PC	Active Mask
	–	G	1111
	G	B	1110
→	G	F	0001

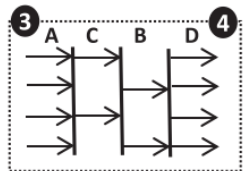
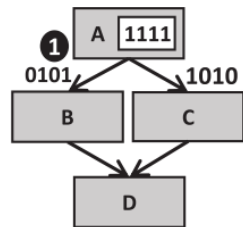
TOS	Reconv PC	Next PC	Active Mask
	–	G	1111
	G	E	1110
	E	D	0110
→	E	C	1000

TOS	Reconv PC	Next PC	Active Mask
	–	G	1111
→	G	E	1110

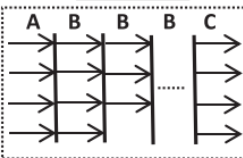
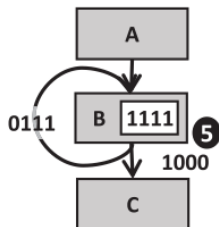
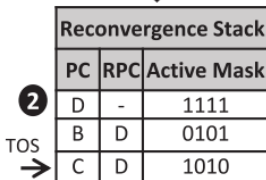
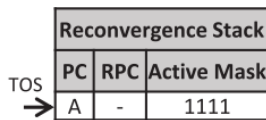
Threads that diverge can be forced to continue executing in lockstep from a reconvergence point

Is the order of insertion in the SIMT stack important?

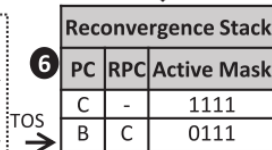
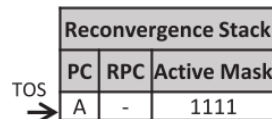
Stack-Based Reconvergence



(a) If-else Example



(b) Loop Example



Reconvergence Scheduling Constraints

Stack-based SIMT execution constrains thread scheduling

Serialization If the threads in the taken branch execute first, then the threads in the not-taken branch block until the taken branch reaches the reconvergence point (or vice versa)

Forced reconvergence When the threads in the taken branch reach the reconvergence point, they block waiting for the threads in the not-taken branch to reach the reconvergence point (or vice versa)

SIMT Deadlock

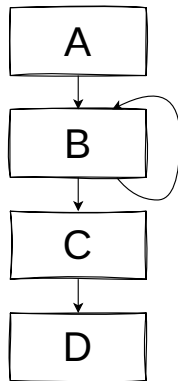
Stack-based implementation of SIMT execution can lead to a deadlock

- Happens because of a cyclic dependence between the scheduling constraints and a synchronization operation

```
A: *mutex=0;  
B: while(!atomicCAS(mutex, 0, 1)) {}  
C: // critical section  
D: atomicExch(mutex, 0);
```

TOS	Reconv PC	Next PC	Active Threads
	C	B	1-31
→	C	C	0

TOS	Reconv PC	Next PC	Active Threads
→	C	B	1-31



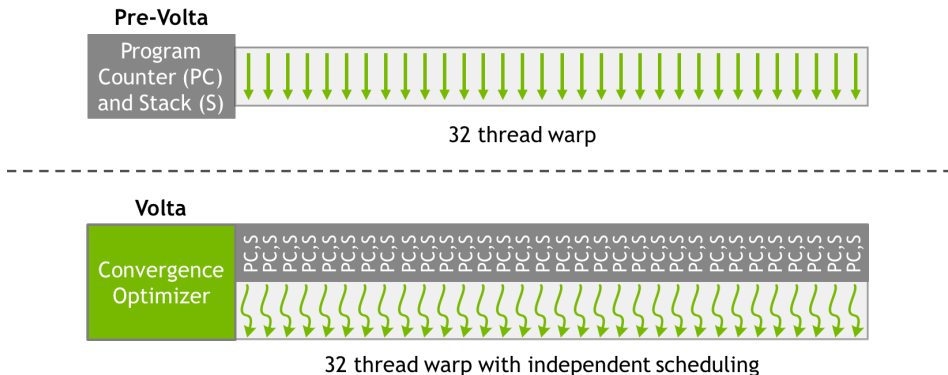
Avoiding SIMT Deadlock

```
*mutex=0;  
while(!atomicCAS(mutex, 0, 1)) {}  
// critical section  
atomicExch(mutex, 0);
```

```
int done = 0;  
*mutex=0;  
while (!done) {  
    if (!atomicCAS(&mutex, 0, 1)) {  
        // critical section  
        atomicExch(mutex, 0);  
        done = 1;  
    }  
}
```

Volta SIMT Model

- Architectures older than Volta maintained a shared program counter (PC) and call stack per warp
- Volta onward, the execution state (i.e., PC and call stack) is maintained per thread



Independent Thread Scheduling (ITS)

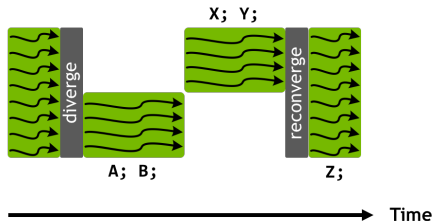
- Volta architecture introduces ITS among threads in a warp
- ITS allows intra-warp synchronization which was previously not possible
- Threads can now diverge and reconverge at sub-warp granularity
- ITS thus makes it easy to implement complex, fine-grained sharing on GPUs

- The SIMT stack is replaced with per-warp convergence barriers
 - ▶ The metadata includes barrier participation mask, barrier convergence state, and per-thread states like PC and active status

SIMT Models across NVIDIA GPU Architectures

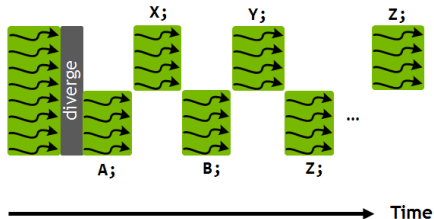
Till Pascal

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;
```



Volta onward

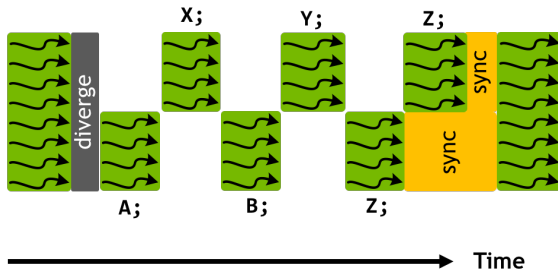
```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;
```



Intra-warp Synchronization

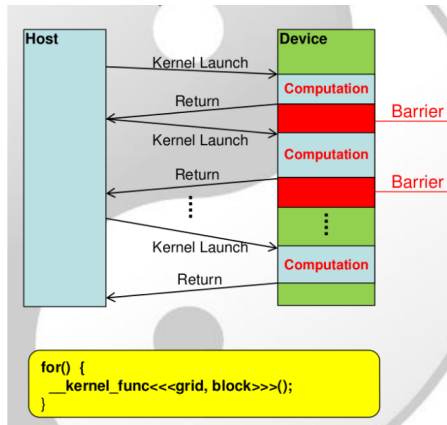
Use `__syncwarp()` to reconverge threads within a warp

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;  
__syncwarp()
```



Host-side Synchronization

- `cudaDeviceSynchronize()` synchronizes all threads in a grid
 - ▶ There are other cheaper variants
- For threads from different grids, writes from a kernel happen before reads from subsequent grid launches



CUDA Memory Model

- A **memory consistency model** is a set of rules that govern how systems process memory operation requests from multiple processors
 - ▶ Determines the order in which memory operations appear to execute
 - ▶ Specifies the allowed behaviors of multithreaded programs executing with shared memory
 - ▶ Memory models are defined both for hardware (e.g., x86-TSO) and programming languages (e.g., C++ and Java)
- CUDA implements a weakly-ordered memory model
- The order in which a thread writes data is not necessarily the order in which the data is observed being written by another CUDA or host thread
- The behavior of racy CUDA programs is undefined

Weakly-Ordered Memory Model in CUDA

```
__device__ int X = 1, Y = 2;
```

Thread 1

```
__device__ void writeXY() {  
    X = 10;  
    Y = 20;  
}
```

Thread 2

```
__device__ void readXY() {  
    int B = Y;  
    int A = X;  
}
```

What are the possible values of A and B?

A racy program has **undefined** behavior, and has no defined semantics

Memory Fences

Semantics of `__threadfence_block()`

- All writes made by the calling thread before the call are observed by all threads in the block as occurring before all writes made by the calling thread after the call
- All reads from all memory made by the calling thread before the call are ordered before all reads from all memory made by the calling thread after the call

```
__device__ int X = 1, Y = 2;
```

Thread 1

```
__device__ void writeXY() {  
    X = 10;  
    __threadfence();  
    Y = 20;  
}
```

Thread 2

```
__device__ void readXY() {  
    int B = Y;  
    __threadfence();  
    int A = X;  
}
```

What are the possible values of A and B?

What can we say about the usage of `__threadfence()`?

```
__device__ int X = 1, Y = 2;
```

Thread 1

```
__device__ void writeXY() {  
    X = 10;  
    __threadfence();  
    Y = 20;  
}
```

Thread 2

```
__device__ void readXY() {  
    int B = Y;  
    __threadfence();  
    int A = X;  
}
```

Is using `__threadfence()` always (i) correct and (ii) required?

__syncthreads() vs __threadfence()

__syncthreads()

- Establishes **visibility** of memory operations across threads in a block
- Includes a barrier plus memory fence functionality (i.e., __threadfence_block())

__threadfence()

- Ensures **ordering** of memory operations by a thread
- __threadfence_block() does not imply a barrier

Concurrency with CUDA Streams

Overlap host and device computation with data transfers

Classic Copy-Then-Execute Model

```
1 cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);  
2 kernel<<<1,N>>>(d_a);  
3 cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- Data transfer on line 1 is blocking or synchronous
 - ▶ Host thread cannot launch the kernel until the copy is done
- Kernel launch on line 2 is asynchronous
- Data transfer on line 3 cannot begin until the kernel completes due to device-side ordering

Overlap Host and Device Computation

```
cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);  
kernel<<<1,N>>>(d_a);  
cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

```
cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);  
kernel<<<1,N>>>(d_a);  
// Host gets work done  
h_func(h_b);  
cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Utilize GPU Hardware

- Overlap kernel execution with memory copy between host and device
- Overlap execution of multiple kernels if there are enough resources
- Recent GPUs support overlapped execution
 - ▶ Check the fields `asyncEngineCount` and `concurrentKernels` from the `cudaDeviceProp` structure

Asynchronous commands in CUDA

- kernel launches
- memory copies from host to device of a memory block of 64 KB or less
- memory copies performed by functions that are suffixed with `Async`
- memory copies between two addresses to the same device memory
- memory set function calls

CUDA Streams

- All GPU device operations run in a stream
- A stream is a sequence of operations that execute on the device **in the order** in which they were issued by the host
 - ▶ Operations across streams can interleave and run concurrently
- The default “null” stream is used if no custom stream is specified
 - ▶ No operation in the default stream will begin until all previously issued operations in any stream have completed
 - ▶ An operation in the default stream must complete before any other operation in any stream will begin

```
// Both kernel launches are on the default stream  
kernel1<<< blocks, threads, bytes >>>();    // default stream  
kernel2<<< blocks, threads, bytes, 0 >>>(); // stream 0
```

Using a Non-default Stream

Manipulate non-default streams from the host

```
cudaStream_t stream1;  
cudaError_t result;  
result = cudaStreamCreate(&stream1);  
result = cudaStreamDestroy(stream1);
```

Issue a data transfer to a non-default stream

```
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);
```

Specifying a stream during kernel launch is optional

```
kernel1<<<blocks, threads, bytes>>>(); // default/NULL stream  
kernel2<<<blocks, threads, bytes, stream1>>>();
```

Non-default Streams

- Operations in a non-default stream are **non-blocking** with the host
- Use `cudaDeviceSynchronize()`
 - ▶ Blocks host until **all** previously issued operations on the device have completed
- Cheaper alternatives
 - ▶ `cudaStreamSynchronize()`, `cudaEventSynchronize()`,...

```
cudaStream_t stream1;  
cudaError_t res;  
res = cudaStreamCreate(&stream1);  
res = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);  
increment<<<1,N,0,stream1>>>(d_a);  
cudaStreamSynchronize(stream1); // Blocks the host thread  
res = cudaStreamDestroy(&stream1);
```

Overlapping Kernel Execution and Data Transfers

```
for (int i = 0; i < nStreams; ++i) {  
    int offset = i * streamSize;  
    cudaMemcpyAsync(&d_a[offset], &h_a[offset], streamBytes,  
        cudaMemcpyHostToDevice, stream[i]);  
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);  
    cudaMemcpyAsync(&h_a[offset], &d_a[offset], streamBytes,  
        cudaMemcpyDeviceToHost, stream[i]);  
}
```

Overlapping Kernel Execution and Data Transfers

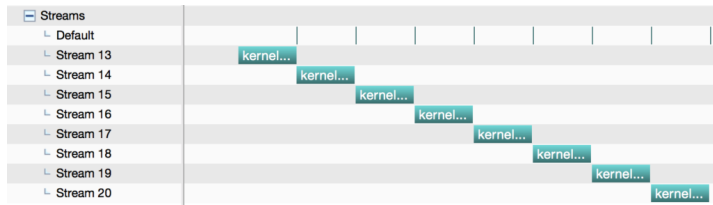
```
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &h_a[offset], streamBytes,
        cudaMemcpyHostToDevice, stream[i]);
}
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
}
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&h_a[offset], &d_a[offset], streamBytes,
        cudaMemcpyDeviceToHost, stream[i]);
}
```

Streams and Concurrency in CUDA 7+

- Prior to CUDA 7, all host threads shared the default stream
 - ▶ The default stream **implicitly synchronizes** with all other streams on the device
 - ▶ Two commands from different streams cannot run concurrently if the host thread issues any CUDA command to the default stream between them
- CUDA 7+ provides an option to have a per-host-thread default stream
 - ▶ Commands issued to the default stream by different host threads can run concurrently
 - ▶ Commands in the default stream may run concurrently with commands in non-default streams
 - ▶ Pass the option `--default-stream per-thread` to `nvcc` to enable per-thread default streams in CUDA 7+

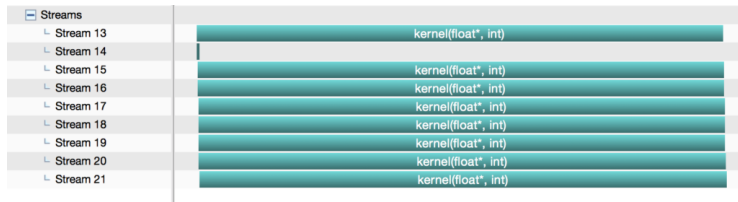
Multi-Stream Example: Legacy Behavior

```
for (int i = 0; i < num_streams; i++) {  
    cudaStreamCreate(&streams[i]);  
    cudaMalloc(&data[i], N * sizeof(float));  
    // launch one worker kernel per stream  
    kernel<<<1, 64, 0, streams[i]>>>(data[i], N);  
    // launch a dummy kernel on the default stream  
    kernel<<<1, 1>>>(0, 0);  
}
```



Multi-Stream Example: Per-Thread Default Stream

```
for (int i = 0; i < num_streams; i++) {  
    cudaStreamCreate(&streams[i]);  
    cudaMalloc(&data[i], N * sizeof(float));  
    // launch one worker kernel per stream  
    kernel<<<1, 64, 0, streams[i]>>>(data[i], N);  
    // launch a dummy kernel on the default stream  
    kernel<<<1, 1>>>(0, 0);  
}
```



Achievable Concurrency

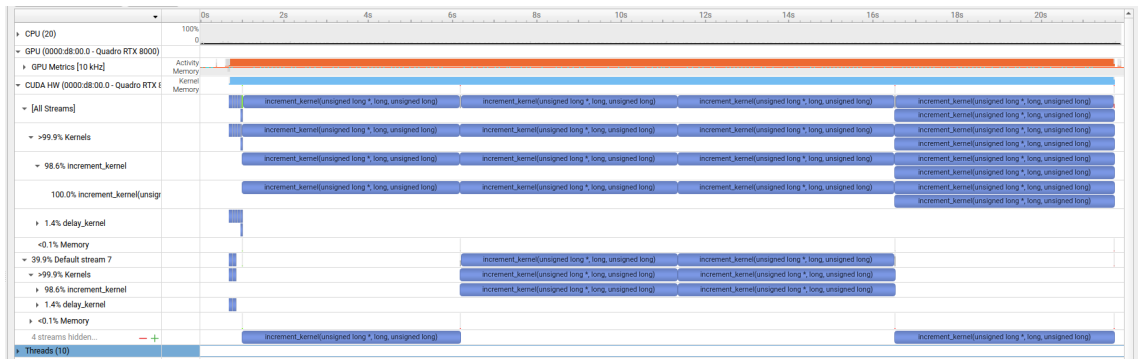
Is it possible to launch two kernels that do independent tasks concurrently?

```
// host and device initialization  
...  
kernel1<<<...>>> (params); // launch kernel1  
kernel2<<<...>>> (params); // launch kernel2  
...
```

We can launch concurrent kernels in different streams

- There must be resources available to run multiple kernels concurrently
- The maximum concurrency is 16 kernels on Fermi, 32 on Kepler, and 128 from Maxwell to Blackwell

Concurrent Kernels



Can we run two CUDA kernels from two different applications concurrently?

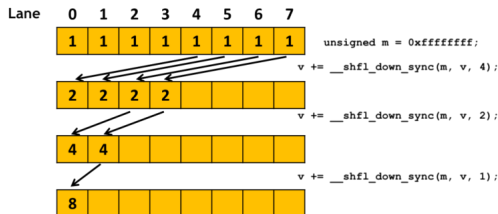
- A kernel from one CUDA context (analogue of host processes for the device) cannot execute concurrently with a kernel from another CUDA context
- The GPU may time slice (interleave execution) to provide forward progress to each context
 - ▶ The full GPU hardware may not be utilized
- Must enable Multi-Process Service (MPS) to run kernels from multiple processes simultaneously
- Multi-Instance GPU (MIG) can be used to partition a GPU into isolated hardware units
 - ▶ Two kernels will run on different physical partitions

Warp Primitives

Efficient intra-warp computation

Warp-Level Primitives

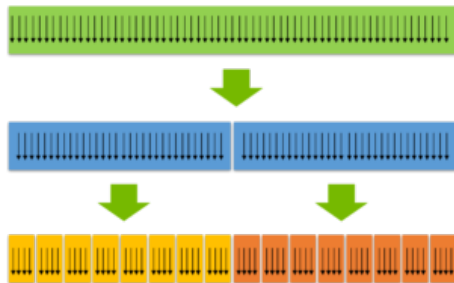
- `_ballot_sync(mask, predicate)`
 - ▶ Evaluate predicate for all non-exited threads in mask and return an integer whose N^{th} bit is set if predicate evaluates to non-zero for the N^{th} thread
- `__activemask()`
 - ▶ Returns a 32-bit integer mask of all currently active threads in the calling warp
- `__syncwarp(mask)`
 - ▶ Guarantees memory ordering among threads participating in the barrier
 - ▶ Threads within a warp that wish to communicate via memory can store to memory, execute `__syncwarp()`, and then safely read values stored by other threads



Copy a variable from a warp lane with higher ID relative to caller

Cooperative Group

- It may be more natural and efficient to group threads into sizes based on the application
- Cooperative Groups extends the thread hierarchy to provide flexible, dynamic grouping of threads
 - ▶ Helps limit collective functions (both communication and synchronization) to desired thread group



Performance Bottlenecks in CUDA

Key Ideas for Extracting Performance

- Desired application characteristics for device execution
 - (i) Large data-parallel computation
 - (ii) Complex computation kernel to justify the H2D and D2H data movement costs
 - Keep data on the device to avoid repeated transfers
- Try and reduce resource consumption (check with `-res-usage`)
- Exploit SIMT, reduce thread divergence in a warp
- Strive for good locality, use tiling to exploit shared memory
 - ▶ Improve throughput by reducing global memory traffic
 - ▶ Copy blocks of data from global memory to shared memory and operate on them (e.g., matrix multiplication kernel)
- Optimize memory accesses
 - Global memory memory coalescing
 - Shared memory avoid bank conflicts

What can we say about this code?

```
__global__ void dkernel(float *vector, int vectorsize) {  
    int id = blockIdx.x * blockDim.x + threadIdx.x;  
    switch (id) {  
        case 0: vector[id] = 0; break;  
        case 1: vector[id] = vector[id] * 10; break;  
        case 2: vector[id] = vector[id - 2]; break;  
        case 3: vector[id] = vector[id + 3]; break;  
        ...  
        case 31: vector[id] = vector[id] * 9; break;  
    }  
}
```

Dealing with Thread Divergence

Thread divergence renders execution sequential because the SIMD hardware takes multiple passes through the divergent paths

```
if (threadIdx.x / WARP_SIZE > 2) {}
```

Condition evaluating to different truth values may not be bad

- Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

```
if (threadIdx.x > 2) {}
```

Conditions evaluating to different truth values for threads in a warp is bad

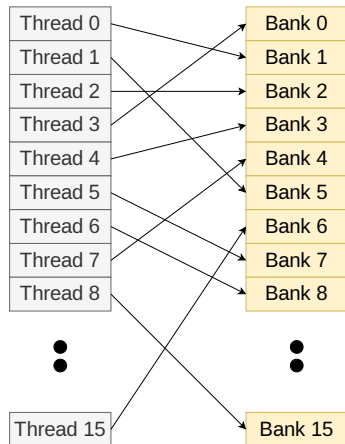
- Creates two different control paths for threads in a block; branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp

Parallel Memory Architecture

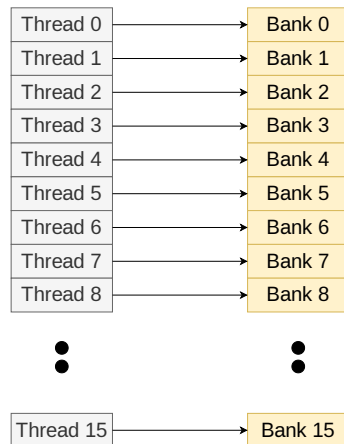
- In a parallel architecture, many threads access memory
- Memory is divided into **banks** to achieve high bandwidth
 - ▶ Each bank can service one address per cycle
 - ▶ A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
 - Conflicting accesses are serialized

Example of Bank Addressing

Random access

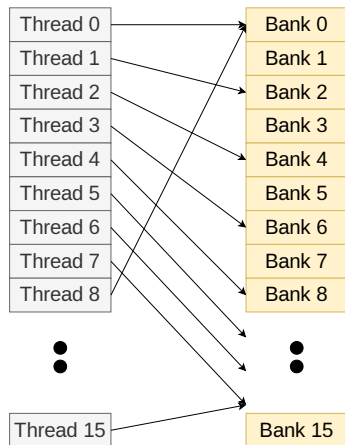


Linear access, stride=1

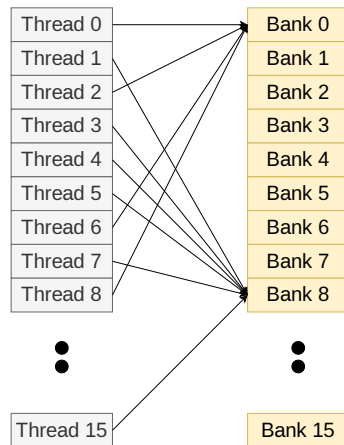


Example of Bank Addressing

Linear access, stride=2



Linear access, stride=8



Bank Conflicts in Shared Memory

NVIDIA GPUs have 32 shared memory banks per SM

- Each 4-byte shared memory word w is mapped to bank $(w/4)\%32$

Fast case

- If all threads of a warp access different banks, there is no bank conflict
- If all threads of a warp access an identical address, there is no bank conflict (broadcast)

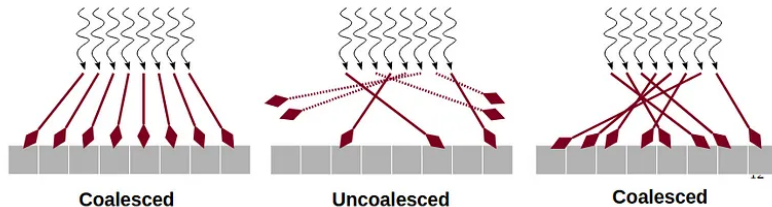
Slow case

- Bank conflict arises when multiple threads access the same bank
 - ▶ Must serialize the accesses
- Cost = max # of simultaneous accesses to a single bank

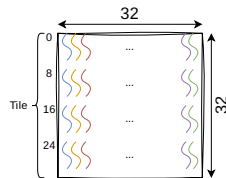
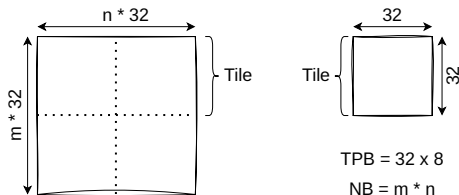
Give low priority to fix low-degree bank conflicts since resolving it will increase instructions

Memory Coalescing

- Issuing a memory instruction from a single warp can generate up to 32 data cache accesses
- Coalescing reduces the number of memory requests by merging accesses from multiple lanes into cache-line-sized chunks when there is spatial locality across the warp
 - ▶ Coalesced memory accesses imply a warp accesses adjacent data in a cache line
 - ▶ In the best case, this results in one memory transaction
- Uncoalesced memory accesses imply a warp accesses scattered data in different cache lines leading to memory divergence
 - ▶ This may result in 32 different memory transactions



Copying a Matrix



Each 32×8 thread block copies a 32×32 tile

```
__global__ void copy(float *odata, const float *idata) {  
    int x = blockIdx.x * TILE_DIM + threadIdx.x;  
    int y = blockIdx.y * TILE_DIM + threadIdx.y;  
    int width = gridDim.x * TILE_DIM;  
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS) // BLOCK_ROWS is 8  
        odata[(y+j)*width + x] = idata[(y+j)*width + x];  
}
```

reads from `idata` and writes to `odata` are coalesced

Matrix Transpose

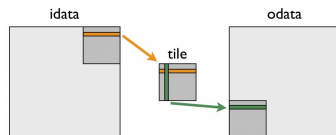
```
__global__ void transposeNaive(float *odata, const float *idata) {  
    int x = blockIdx.x * TILE_DIM + threadIdx.x;  
    int y = blockIdx.y * TILE_DIM + threadIdx.y;  
    int width = gridDim.x * TILE_DIM;  
    for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)  
        odata[x*width + (y+j)] = idata[(y+j)*width + x];  
}
```

reads from idata are coalesced,
but writes to odata have a stride of
1024

Optimizing Matrix Transpose

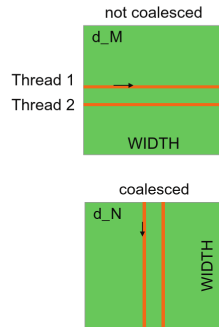
```
__global__ void transposeCoalesced(float *odata, const float *idata) {  
    __shared__ float tile[TILE_DIM][TILE_DIM];  
    int x = blockIdx.x * TILE_DIM + threadIdx.x;  
    int y = blockIdx.y * TILE_DIM + threadIdx.y;  
    int width = gridDim.x * TILE_DIM;  
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)  
        tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];  
    __syncthreads();  
    x = blockIdx.y * TILE_DIM + threadIdx.x; // transpose block offset  
    y = blockIdx.x * TILE_DIM + threadIdx.y;  
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)  
        odata[(y+j)*width+x] = tile[threadIdx.x][threadIdx.y+j];  
}
```

 transpose-nvidia.cu



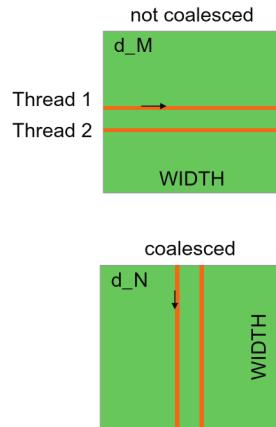
Matrix Multiplication Example

```
__global__ void matmulKernel(float* A, float* B, float* C) {  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    float tmp = 0;  
    if (row < N && col < N) {  
        // Each thread computes one element of the matrix  
        for (int k = 0; k < N; k++) {  
            tmp += A[row * N + k] * B[k * N + col];  
        }  
        C[row * N + col] = tmp;  
    }  
}
```

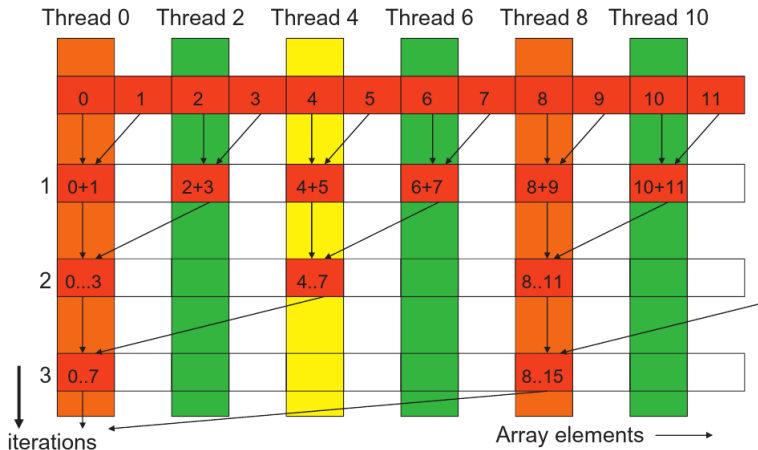


Optimizing Global Memory Accesses

- Try to ensure that memory requests from a warp can be coalesced
 - ▶ Using optimizations like tiling to make use of the faster shared memory
 - ▶ Stride-one access across threads in a warp is good
 - ▶ Use structure of arrays rather than array of structures



Implementing a Reduction Kernel in CUDA



Reduction Kernel

```
__global__ void reduction1(const uint64_t *d_input, uint64_t *d_output) {
    uint64_t tid = threadIdx.x;
    uint64_t pindex = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ uint64_t partialSum[TPB];
    partialSum[tid] = d_input[pindex];
    __syncthreads();
    unsigned int t = threadIdx.x;
    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        if (t % (2*stride) == 0)
            partialSum[t] += partialSum[t+stride];
        __syncthreads();
    }
    if (tid == 0) {
        d_output[blockIdx.x] = partialSum[tid];
    }
}
```

only even threads
are active

Possible Optimizations on the Reduction Kernel



Optimizing Parallel Reduction in CUDA

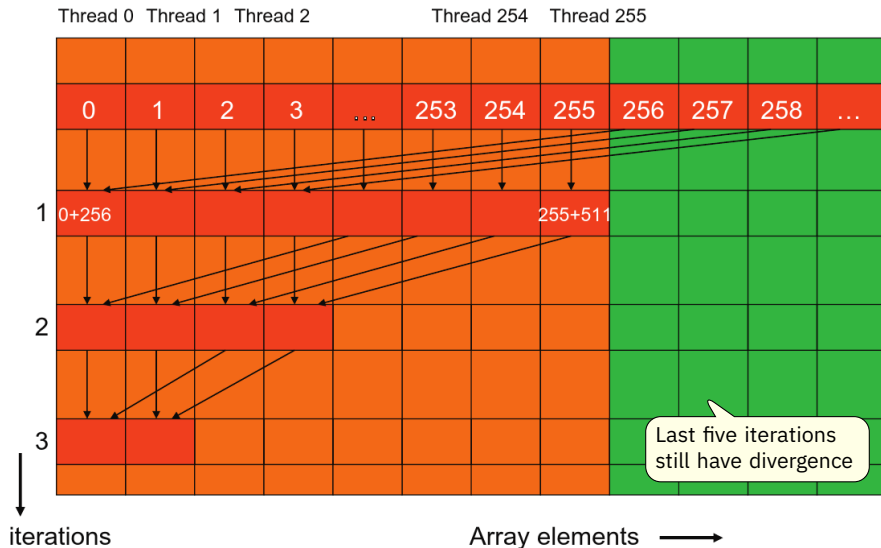
Mark Harris
NVIDIA Developer Technology

- (i) basic implementation with modulo operator
- (ii) strided access starting with each thread accessing two adjacent locations
- (iii) strided access with reversed loop index
- (iv) halve the number of threads
- (v) unroll the last few loop iterations and avoid synchronization
- (vi) ...

Reduction Kernel

```
__global__ void reduction3(const uint64_t *d_input, uint64_t *d_output) {
    uint64_t tid = threadIdx.x;
    uint64_t pindex = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ uint64_t partialSum[TPB];
    partialSum[tid] = d_input[pindex];
    __syncthreads();
    for (uint64_t stride = (blockDim.x >> 1); stride >= 1; stride >>= 1) {
        if (tid < stride) {
            partialSum[tid] += partialSum[tid + stride];
        }
        __syncthreads();
    }
    if (tid == 0) {
        d_output[blockIdx.x] = partialSum[tid];
    }
}
```

Execution of the Revised Reduction Kernel



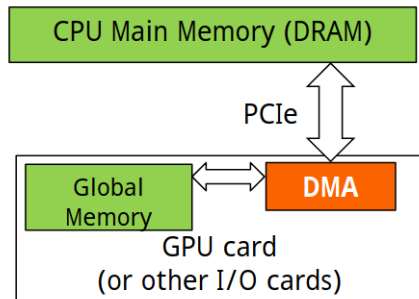
Avoid Divergence in the Last Few Iterations

```
for (unsigned int stride = blockDim.x/2; stride > 32; stride /= 2) {  
    if (t < stride)  
        partialSum[t] += partialSum[t+stride];  
    __syncthreads();  
}  
if (tid < 32) {  
    partialSum[tid] += partialSum[tid+32];  
    partialSum[tid] += partialSum[tid+16];  
    partialSum[tid] += partialSum[tid+8];  
    partialSum[tid] += partialSum[tid+4];  
    partialSum[tid] += partialSum[tid+2];  
    partialSum[tid] += partialSum[tid+1];  
}
```

Efficient Data Management

Data Transfer Between CPU and GPU

- Direct Memory Access (DMA) hardware transfers bytes between physical memory address spaces
 - ▶ Uses system interconnect, typically PCIe in today's systems
- `cudaMemcpy()` copies data as one or more DMA transfers
 - ▶ Source and destination pointers are virtual
- Virtual memory complicates data transfer
 - ▶ Pages in virtual address space are mapped into and out of the physical memory
 - ▶ The OS could accidentally page out the data that is being accessed by a DMA unit and page in another virtual page into the same physical location



Enabling DMA Transfer in CUDA with Pinned Memory

Pinned pages

Virtual memory pages that are specially marked so that they cannot be paged out (also called Page-Locked Memory or Locked Pages)

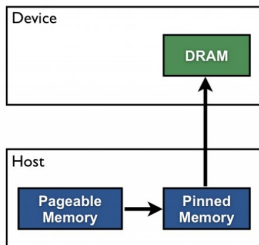
CPU memory that serves as the source or destination of a DMA transfer with `cudaMemcpy()` must be allocated as pinned memory

- (i) The entire source and destination virtual address ranges are translated, and mapped to physical pages
- (ii) The CUDA driver checks if the entire memory page is pinned (i.e., unswappable)
- (iii) If the source or destination of `cudaMemcpy()` in the host is not pinned, it needs to be first copied to a pinned memory leading to extra overhead
 - ▶ `cudaMemcpy()` is faster if the host memory is allocated in pinned memory because no extra copy is needed

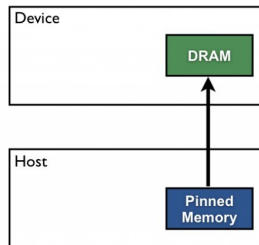
Pinned Memory with CUDA

- Allocate pinned memory with `cudaHostAlloc()` with the option `cudaHostAllocDefault` and free with `cudaFreeHost()`
- Pinned memory is a limited resource—over-subscription can have serious consequences

Pageable Data Transfer



Pinned Data Transfer



cudaMemCpy() with Pageable Memory

```
swarnendu@nilgiri:~/cuda-examples/src/vector$ nvprof ./vector-addition.out
==1941000== NVPROF is profiling process 1941000, command: ./vector-addition.out
Time taken (ms): h2d: 237.56 Kernel: 8.40618 d2h: 220.426
==1941000== Profiling application: ./vector-addition.out
==1941000== Profiling result:
   Type  Time(%)   Time      Calls      Avg      Min      Max  Name
GPU activities:  67.51%  474.62ms        2  237.31ms  237.16ms  237.46ms  [CUDA memcpy HtoD]
                31.30%  220.01ms        1   220.01ms  220.01ms  220.01ms  [CUDA memcpy DtoH]
                1.19%   8.3743ms        1   8.3743ms  8.3743ms  8.3743ms  vecAdd(float const *, float const *, float
*, unsigned int)
  API calls:    82.20%  695.45ms        3  231.82ms  220.42ms  237.64ms  cudaMemcpy
                16.13%  136.47ms        3  45.490ms  111.40us  136.24ms  cudaMalloc
                1.00%   8.4519ms        3   2.8173ms  2.8090us  8.3707ms  cudaEventSynchronize
                0.64%   5.3911ms        3   1.7970ms  758.27us  2.5189ms  cudaFree
                0.02%   142.30us       101   1.4080us   123ns   59.020us  cuDeviceGetAttribute
                0.00%   39.372us        6   6.5620us   1.7480us  18.458us  cudaEventRecord
                0.00%   32.013us        1   32.013us   32.013us  32.013us  cudaLaunchKernel
                0.00%   22.907us        1   22.907us   22.907us  22.907us  cuDeviceGetName
                0.00%   9.5030us        1   9.5030us   9.5030us  9.5030us  cuDeviceGetPCIBusId
                0.00%   9.2820us        2   4.6410us    865ns   8.4170us  cudaEventCreate
                0.00%   5.1920us        3   1.7300us    787ns   2.4430us  cudaEventElapsedTime
                0.00%   1.5260us        3    508ns    180ns   1.1610us  cuDeviceGetCount
                0.00%    601ns         1    601ns    601ns    601ns  cuDeviceTotalMem
                0.00%    598ns         2    299ns    125ns    473ns  cuDeviceGet
                0.00%    218ns         1    218ns    218ns    218ns  cuDeviceGetUuid

swarnendu@nilgiri:~/cuda-examples/src/vector$
```

cudaMemCpy() with Pinned Memory

```
swarnendu@nilgiri:~/cuda-examples/src/vector$ nvprof ./vector-addition-hostalloc-default.out
==1941051== NVPROF is profiling process 1941051, command: ./vector-addition-hostalloc-default.out
Time taken (ms): h2d: 87.2645 Kernel: 8.52237 d2h: 81.3356
==1941051== Profiling application: ./vector-addition-hostalloc-default.out
==1941051== Profiling result:
```

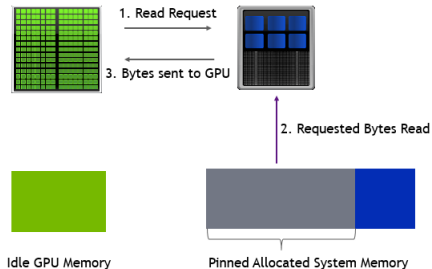
	Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:		66.03%	174.50ms	2	87.249ms	87.246ms	87.252ms	[CUDA memcpy HtoD]
		30.77%	81.317ms	1	81.317ms	81.317ms	81.317ms	[CUDA memcpy DtoH]
		3.20%	8.4589ms	1	8.4589ms	8.4589ms	8.4589ms	vecAdd(float const *, float const *, float
*, unsigned int)								
API calls:		68.93%	1.46052s	3	486.84ms	437.10ms	585.73ms	cudaHostAlloc
		18.30%	387.85ms	3	129.28ms	125.50ms	136.48ms	cudaFreeHost
		12.08%	255.92ms	3	85.306ms	81.333ms	87.328ms	cudaMemcpy
		0.40%	8.4608ms	3	2.8203ms	3.1570us	8.4543ms	cudaEventSynchronize
		0.25%	5.2468ms	3	1.7489ms	627.15us	2.5135ms	cudaFree
		0.03%	555.08us	3	185.03us	106.57us	322.72us	cudaMalloc
		0.01%	144.27us	101	1.4280us	166ns	57.635us	cuDeviceGetAttribute
		0.00%	57.403us	1	57.403us	57.403us	57.403us	cudaLaunchKernel
		0.00%	38.628us	6	6.4380us	1.7470us	26.804us	cudaEventRecord
		0.00%	15.428us	2	7.7140us	1.1860us	14.242us	cudaEventCreate
		0.00%	13.944us	1	13.944us	13.944us	13.944us	cuDeviceGetName
		0.00%	10.629us	1	10.629us	10.629us	10.629us	cuDeviceGetPCIBusId
		0.00%	4.1260us	3	1.3750us	769ns	2.4630us	cudaEventElapsedTime
		0.00%	3.1800us	3	1.0600us	241ns	2.3700us	cuDeviceGetCount
		0.00%	945ns	1	945ns	945ns	945ns	cuDeviceTotalMem
		0.00%	812ns	2	406ns	156ns	656ns	cuDeviceGet

Zero-Copy Memory with CUDA

Zero copy memory

Pinned memory that is mapped into the device address space

- Both host and device have fine-grained direct access
- + Can leverage host memory when there is insufficient device memory
- + Avoids explicit data transfers between host and device
- Should be used for occasionally when the data is read-only or the GPU memory is really scarce



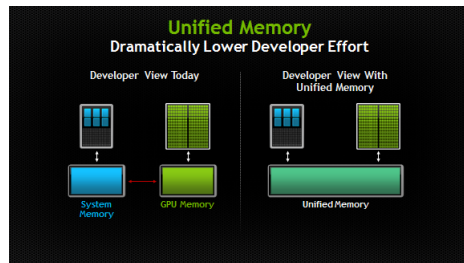
cudaMemCpy() with Zero-Copy Memory

```
swarnendu@nilgiri:~/cuda-examples/src/vector$ nvprof ./vector-addition-hostalloc-mapped.out
==1948543== NVPROF is profiling process 1948543, command: ./vector-addition-hostalloc-mapped.out
Invocation i: 0 Time taken (ms): kernel: 378.814
Invocation i: 1 Time taken (ms): kernel: 373.741
Invocation i: 2 Time taken (ms): kernel: 373.747
Invocation i: 3 Time taken (ms): kernel: 373.714
Invocation i: 4 Time taken (ms): kernel: 373.742
Invocation i: 5 Time taken (ms): kernel: 373.717
Invocation i: 6 Time taken (ms): kernel: 376.482
Invocation i: 7 Time taken (ms): kernel: 373.599
Invocation i: 8 Time taken (ms): kernel: 376.099
Invocation i: 9 Time taken (ms): kernel: 376.226
==1948543== Profiling application: ./vector-addition-hostalloc-mapped.out
==1948543== Profiling result:
      Type  Time(%)   Time     Calls   Avg      Min      Max  Name
GPU activities: 100.00%  3.74542s      10  374.54ms  373.60ms  376.60ms  vecAdd(float const *, float const *, float
*, unsigned int)
  API calls:
    57.77%  3.76791s      10  376.79ms  375.56ms  378.79ms  cudaEventSynchronize
    30.61%  1.99639s       3  665.46ms  583.01ms  742.25ms  cudaHostAlloc
     6.53%  425.82ms       3  141.94ms  141.12ms  143.34ms  cudaFreeHost
     5.07%  330.84ms       1  330.84ms  330.84ms  330.84ms  cudaSetDeviceFlags
     0.01%  447.18us      10  44.718us  27.583us  94.893us  cudaLaunchKernel
     0.00%  207.81us      20  10.390us  2.7920us  30.620us  cudaEventRecord
     0.00%  139.87us     101  1.3840us   140ns   55.603us  cuDeviceGetAttribute
     0.00%  59.182us      10  5.9180us  3.9300us  14.372us  cudaEventElapsedTime
     0.00%  41.062us       2  20.531us   940ns   40.122us  cudaEventCreate
```



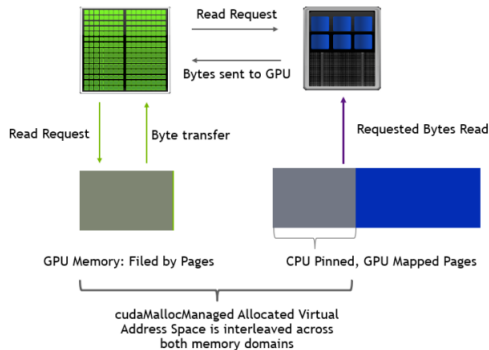
Managed Memory

- Unified Virtual Memory (UVM) provides a single memory space accessible by all GPUs and CPUs in the system
- Use `cudaMallocManaged()` to allocate data in unified memory or use `__managed__` keyword in the global scope
 - ▶ Returns a pointer that can be accessed from both host and device code

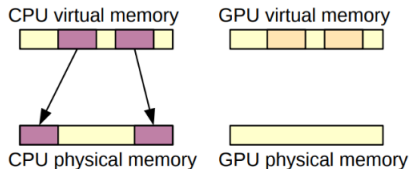


Managed Memory

- UVM features have been evolving starting from CUDA 6+
 - ▶ 6.x: use a single pointer in both CPU functions and GPU kernels
 - ▶ 8.x: added 49-bit virtual addressing and on-demand page migration
- CUDA runtime **automatically migrates data** allocated in Unified Memory between host and device, different from unlike unified virtual addressing (UVA)

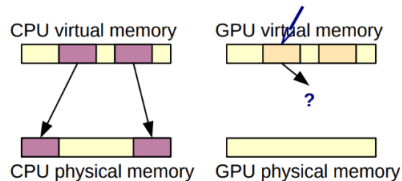


On-demand Paging



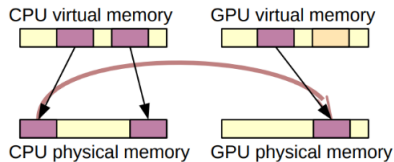
1

Data allocated in CPU memory



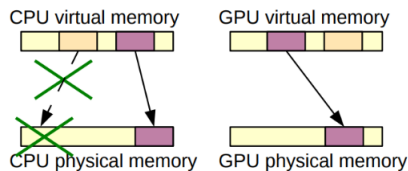
2

GPU touches unallocated page, triggers page fault



3

Page fault handler allocates page in GPU mem, copies contents



4

If GPU modifies page contents, invalidate CPU copy. Next CPU access will cause data to be copied back from GPU mem.

cudaMemcpy() with Managed Memory

```
swarnendu@nilgiri:~/cuda-examples/src/vector$ nvprof ./vector-addition-managed.out
==1948976== NVPROF is profiling process 1948976, command: ./vector-addition-managed.out
Invocation: 0 Time taken (ms): kernel: 1219.01
==1948976== Profiling application: ./vector-addition-managed.out
==1948976== Profiling result:
Type      Time(%)      Time      Calls      Avg      Min      Max      Name
GPU activities: 100.00%  1.21679s  1  1.21679s  1.21679s  1.21679s  vecAdd(float const *, float const *, float*, int)
  API calls: 73.02%  1.21899s  1  1.21899s  1.21899s  1.21899s  cudaEventSynchronize
    17.84%  297.79ms  3  99.265ms  23.511us  297.70ms  cudaMallocManaged
    9.12%  152.21ms  3  50.737ms  50.422ms  50.941ms  cudaFree
    0.01%  142.95us  101  1.4150us  145ns  55.431us  cuDeviceGetAttribute
    0.00%  75.898us  1  75.898us  75.898us  75.898us  cudaLaunchKernel
    0.00%  59.680us  2  29.840us  5.5550us  54.125us  cudaEventCreate
    0.00%  46.808us  2  23.404us  3.9890us  42.819us  cudaEventRecord
    0.00%  30.918us  1  30.918us  30.918us  30.918us  cuDeviceGetName
    0.00%  9.5370us  1  9.5370us  9.5370us  9.5370us  cuDeviceTotalMem
    0.00%  9.3320us  1  9.3320us  9.3320us  9.3320us  cuDeviceGetPCIBusId
    0.00%  7.1650us  1  7.1650us  7.1650us  7.1650us  cudaEventElapsedTime
    0.00%  2.6680us  3  889ns  183ns  1.3890us  cuDeviceGetCount
    0.00%  883ns  2  441ns  367ns  516ns  cuDeviceGet
    0.00%  350ns  1  350ns  350ns  350ns  cuDeviceGetUuid

==1948976== Unified Memory profiling result:
Device "Quadro RTX 5000 (0)"
Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
36250  54.919KB  4.0000KB  0.9961MB  1.898617GB  224.7973ms  Host To Device
18408  170.67KB  4.0000KB  0.9961MB  2.996094GB  261.0419ms  Device To Host
4693  -  -  -  -  539.8935ms  Gpu page fault groups
Total CPU Page faults: 15349
```



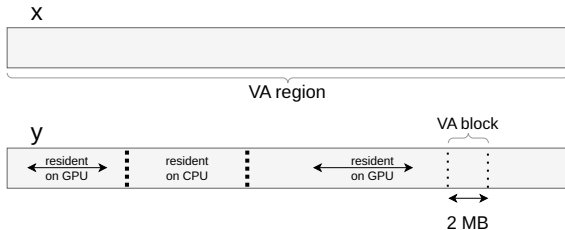
cudaMemcpy() with Managed Memory

```
~/cuda-examples/src/vector$ nvprof --print-gpu-trace ./vector-addition-managed.out 2>&1 | tee output.txt
```

```
...
==1949017== Profiling result:
   Start   Duration      Grid Size    Block Size    Regs*    SSMem*    DSMem*          Device    Context
Stream      Unified Memory  Virtual Address  Name
499.24ms      -              -              -              -          -          -              -
-              PC 0xe0f0cdc8    0x7fd774000000    [Unified Memory CPU page faults]
499.73ms      -              -              -              -          -          -              -
-              PC 0xe0f0cdc8    0x7fd774010000    [Unified Memory CPU page faults]
499.76ms      -              -              -              -          -          -              -
-              PC 0xe0f0cdc8    0x7fd774020000    [Unified Memory CPU page faults]
...
1.59851s      -              -              -              -          -          -              -
-              PC 0xe0f0cdf0    0x7fd76fe80000    [Unified Memory CPU page faults]
1.59862s      -              -              -              -          -          -              -
-              PC 0xe0f0cdf0    0x7fd76ff00000    [Unified Memory CPU page faults]
1.60122s    1.22684s      (1048576 1 1)    (256 1 1)        16          0B          0B    Quadro RTX 5000
1              7              -              -              -          -          -    *, float*, int) [117]
1.60123s    334.53us      -              -              -          -          -    Quadro RTX 5000
-              8    0x7fd730000000    [Unified Memory GPU page faults]
1.60142s    10.016us      -              -              -          -          -    Quadro RTX 5000
-              96.000000KB    0x7fd730000000    [Unified MemoryMemcpy HtoD]
1.60143s    4.6720us      -              -              -          -          -    Quadro RTX 5000
-              32.000000KB    0x7fd730018000    [Unified MemoryMemcpy HtoD]
...
```

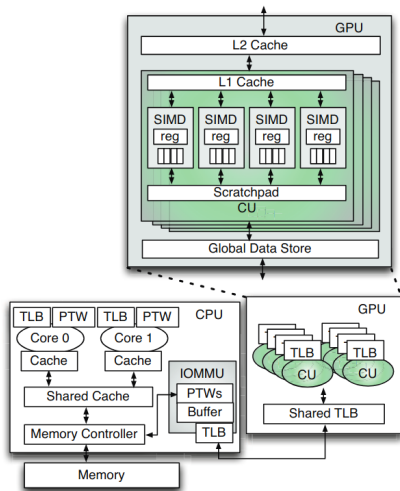
Managing UVM Buffers

```
__global__ void add(int n, float* x, float*  
    int index = blockIdx.x * blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    for (int i = index; i < n; i += stride)  
        y[i] = x[i] + y[i];  
}  
int main(void) {  
    int N = (1 << 20);  
    float *x, *y;  
    cudaMallocManaged(&x, N * sizeof(float));  
    cudaMallocManaged(&y, N * sizeof(float));  
    for (int i = 0; i < N; i++)  
        x[i] = 1.0f; y[i] = 2.0f;  
    int blockSize = 256;  
    int numBlocks = (N + blockSize - 1) / blockSize;  
    add<<<numBlocks, blockSize>>>(N, x, y);  
    cudaDeviceSynchronize();  
    ...  
}
```



Support for Virtual Address Translation

- One or more SMs have a private L1 TLB
 - ▶ Two SMs share an L1 TLB in NVIDIA GPUs (called μ -TLB)
- The L1 TLBs are backed by a larger L2 TLB that is shared across all the SMs in the GPU
- IOMMU is a hardware component on the CPU that services address translation requests for accesses to the DRAM by any accelerator (e.g., GPU)
 - ▶ IOMMU supports multiple page table walkers (e.g., 8–16) to concurrently service multiple page table walk requests (TLB misses)
 - ▶ IOMMU employs small page walk caches for the first few levels of the page tables

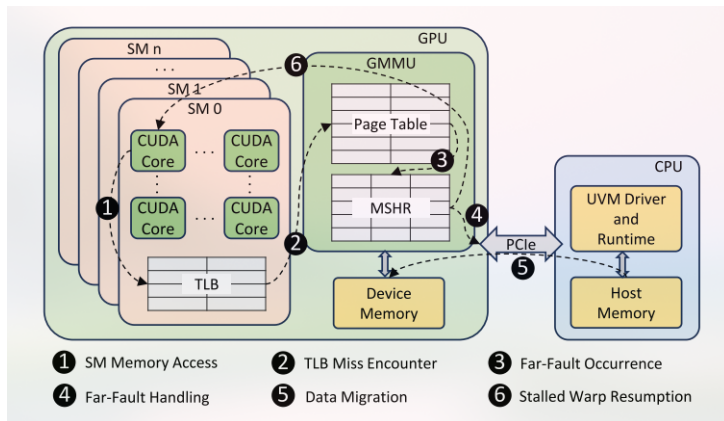


Steps During Address Translation in GPUs

- (i) The LD/ST unit in a SM will generate virtual global memory addresses when a warp executes LDG/STG instruction
- (ii) A hardware coalescer unit merges multiple requests to the same virtual page generated by the same SIMD instruction
- (iii) The coalesced translation request will look up the private per-SM L1 TLB
- (iv) On a miss in the μ -TLB, the shared L2 TLB will be looked up
- (v) On a miss in the shared L2 TLB, a page table walk will be performed
 - ▶ The translation request is still on the GPU
- (vi) On a miss, a **far fault** is raised and the fault is enqueued in a on-GPU **fault buffer**
- (vii) The UVM driver reads a group of faults from the fault buffer periodically to amortize overhead
- (viii) The fault batch is first preprocessed (duplicates are removed) and are sorted by fault addresses, and then the faults are serviced

Handling Far Faults in UVM

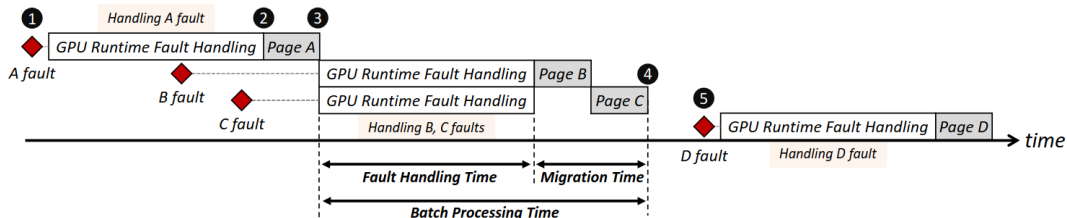
When a GPU kernel executes a global memory operation, if the GPU-side MMU (GMMU) finds that the physical page is not in the GPU memory, a **far fault** is triggered



Steps During Address Translation in GPUs

- (ix) Fault servicing includes scheduling H2D data migration, unmapping pages from the host, and installing the PTE on the device page table
- (ix) The faulting instruction is replayed on the relevant SM after the fault is serviced

Handling far fault by the UVM driver is expensive ($\sim 10\text{--}50\ \mu\text{s}$) because it requires long latency communications between the CPU and GPU over the PCIe bus

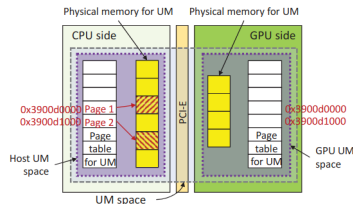


Memory Access Divergence

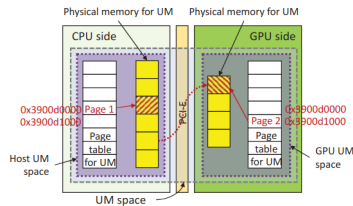
- An SIMD memory instruction cannot complete until data for all threads are available
 - ▶ Problematic for irregular applications with little scope for coalescing
 - ▶ Execution of one instruction requires multiple cache accesses when accesses fall on distinct cache lines and multiple virtual-to-physical address translations when accesses fall on distinct pages
- Negative impact from divergence can impact address translation more than cache access
 - ▶ Irregular memory accesses can lead from 1 to warp size (32/64) address translation requests, most will miss in the TLB
 - ▶ A page table walk on a TLB miss can take up to four memory accesses, for a total of 128–256 memory accesses per instruction

Unified Virtual Memory (UVM)

- On pre-Pascal GPUs, `cudaMallocManaged()` allocated managed space on the device that was active at the time of the call
- Pascal onward, `cudaMallocManaged()` does not immediately allocate physical memory
 - ▶ Pages and page table entries are not created until the first access by the GPU or the CPU
- Hardware supports page faulting and migration
 - ▶ The GPU stalls the accessor threads when they access absent pages
 - ▶ The Page Migration Engine migrates pages to the device before resuming the threads



(a) After the host has accessed page 1 and page 2.



(b) After the GPU has accessed page 2.

Pre-Pascal Behavior of cudaMallocManaged()

```
__global__ void add(int n, float* x, float* y) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}

int main(void) {
    int N = (1 << 20);
    float *x, *y;
    cudaMallocManaged(&x, N * sizeof(float));
    cudaMallocManaged(&y, N * sizeof(float));
    for (int i = 0; i < N; i++)
        x[i] = 1.0f; y[i] = 2.0f;
    int blockSize = 256;
    int numBlocks = (N + blockSize - 1) / blockSize;
    add<<<numBlocks, blockSize>>>(N, x, y);
    cudaDeviceSynchronize();
    ...
}
```

x and y are allocated on GPU memory,
driver sets up page table entries

Page fault on CPU, x and y
are copied to CPU memory

Moves data back
to CPU memory

Lacked support for page faults, so all data
is copied to GPU before kernel launch

Pre-Pascal Behavior of cudaMallocManaged()

```
__global__ void add(int n, float* x, float* y) {  
    int index = blockIdx.x * blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    for (int i = index; i < n; i += stride)  
        y[i] = x[i] + y[i];  
}  
  
int main(void) {  
    int N = (1 << 20);  
    float *x, *y;  
    cudaMallocManaged(&x, N * sizeof(float));  
    cudaMallocManaged(&y, N * sizeof(float));  
    for (int i = 0; i < N; i++)  
        x[i] = 1.0f; y[i] = 0.0f;  
    int blockSize = 1024;  
    int numBlocks = (N + blockSize - 1) / blockSize;  
    add<<<numBlocks, blockSize>>>(N, x, y);  
    cudaDeviceSynchronize();  
    ...  
}
```

- No concurrent access
- No on-demand migration to GPU
- No oversubscription

Moves data back
to CPU memory

Lacked support for page faults, so all data
is copied to GPU before kernel launch

Behavior of cudaMallocManaged() for Pascal+

```
__global__ void add(int n, float* x, float* y) {  
    int index = blockIdx.x * blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    for (int i = index; i < n; i += stride)  
        y[i] = x[i] + y[i];  
}
```

Page migration cost is now part of the kernel execution time

```
int main(void) {  
    int N = (1 << 20);  
    float *x, *y;  
    cudaMallocManaged(&x, N * sizeof(float));  
    cudaMallocManaged(&y, N * sizeof(float));  
    for (int i = 0; i < N; i++)  
        x[i] = 1.0f; y[i] = 2.0f;  
    int blockSize = 256;  
    int numBlocks = (N + blockSize - 1) / blockSize;  
    add<<<numBlocks, blockSize>>>(N, x, y);  
    cudaDeviceSynchronize();  
    ...  
}
```

Physical memory for x and y are not immediately allocated, allocated only on first access or prefetch

Page fault on CPU, x and y are allocated on CPU memory

Supports page faults, so no data migration overhead before kernel launch

Profiling with nvprof on Turing GPU

```
swarnendu@nilgiri:~/cuda-examples/src/uvms$ nvprof ./add-grid.out
==2284609== NVPROF is profiling process 2284609, command: ./add-grid.out
==2284609== Profiling application: ./add-grid.out
==2284609== Profiling result:
   Type  Time(%)   Time     Calls   Avg       Min       Max  Name
GPU activities: 100.00% 3.2329ms    1 3.2329ms 3.2329ms 3.2329ms add(float const *, float const *, float*, unsigned int)
  API calls: 97.02% 138.15ms    3 46.049ms 22.308us 138.02ms cudaMallocManaged
              2.28% 3.2434ms    1 3.2434ms 3.2434ms 3.2434ms cudaDeviceSynchronize
              0.53% 755.20us    3 251.73us 216.47us 315.78us cudaFree
              0.09% 133.92us   101 1.3250us   133ns 52.233us cuDeviceGetAttribute
              0.05% 71.813us    1 71.813us 71.813us 71.813us cudaLaunchKernel
              0.02% 24.813us    1 24.813us 24.813us 24.813us cuDeviceGetName
              0.01% 9.2100us    1 9.2100us 9.2100us 9.2100us cuDeviceGetPCIBusId
              0.00% 2.4970us    3 832ns    192ns 2.0830us cuDeviceGetCount
              0.00% 933ns     2 466ns    145ns 788ns    cuDeviceGet
              0.00% 584ns     1 584ns    584ns 584ns    cuDeviceTotalMem
              0.00% 282ns     1 282ns    282ns 282ns    cuDeviceGetUuid

==2284609== Unified Memory profiling result:
Device "Quadro RTX 5000 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      192   64.000KB  4.0000KB  964.00KB  12.00000MB  1.347288ms  Host To Device
       72  170.67KB  4.0000KB  0.9961MB  12.00000MB  1.024603ms  Device To Host
        21      -      -      -      -      3.202094ms  Gpu page fault groups
Total CPU Page faults: 72
swarnendu@nilgiri:~/cuda-examples/src/uvms$
```

Ways to Reduce Page Migration Overhead

(i) Initialize data on the device

```
__global__ void init(int n, float *x, float *y) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride) {
        x[i] = 1.0f; y[i] = 2.0f;
    }
}
```

```
swarnendu@nilgiri:~/cuda-examples/src/uvvm$ nvprof ./add-grid-init.out
==2284736== NVPROF is profiling process 2284736, command: ./add-grid-init.out
==2284736== Profiling application: ./add-grid-init.out
==2284736== Profiling result:
   Type  Time(%)      Time       Calls         Avg           Min           Max    Name
GPU activities:  98.70%   2.8188ms        1  2.8188ms   2.8188ms   2.8188ms  init(float*, float*, float*, unsigned int)
                  1.30%   37.217us        1  37.217us   37.217us   37.217us  add(float const *, float const *, float*, unsigned int)
API calls:       97.29%   137.84ms        3  45.948ms   14.291us   137.74ms  cudaMallocManaged
                  2.01%    2.8438ms        1    2.8438ms   2.8438ms   2.8438ms  cudaDeviceSynchronize
                  0.53%    749.47us        3    249.82us   214.63us   316.77us  cudaFree

==2284736== Unified Memory profiling result:
Device "Quadro RTX 5000 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    121   101.55KB  4.0000KB  0.9922MB  12.00000MB  1.259259ms  Host To Device
    72    170.67KB  4.0000KB  0.9961MB  12.00000MB  1.025338ms  Device To Host
    20      -      -      -      -      -  2.793011ms  Gpu page fault groups
Total CPU Page faults: 72
swarnendu@nilgiri:~/cuda-examples/src/uvvm$
```

Ways to Reduce Page Migration Overhead

- (ii) Use warm-up iterations or take the average of kernel multiple runs
- (iiii) Prefetch data on the GPU with `cudaMemPrefetchAsync()`

```
int device = -1;
cudaGetDevice(&device);
cudaMemPrefetchAsync(X, N*sizeof(float), device,
                     NULL);
cudaMemPrefetchAsync(Y, N*sizeof(float), device,
                     NULL);
cudaMemPrefetchAsync(Z, N*sizeof(float), device,
                     NULL);
```

```
swarnendu@nilgiri:~/cuda-examples/src/uvms$ nvprof ./add-grid-prefetch.out
==2285986== NVPROF is profiling process 2285986, command: ./add-grid-prefetch.out
==2285986== Profiling application: ./add-grid-prefetch.out
==2285986== Profiling result:
   Type  Time(%)   Time    Calls    Avg      Min      Max   Name
GPU activities: 100.00% 29.855us    1 29.855us 29.855us 29.855us add(float const *, float const *, float*, unsigned int)
  API calls:   98.05% 137.99ms    3 45.996ms 35.369us 137.84ms cudaMallocManaged
              0.95% 1.3409ms    1 1.3409ms 1.3409ms 1.3409ms cudaDeviceSynchronize
              0.53% 743.23us    3 247.74us 214.38us 313.59us cudaFree
              0.30% 426.23us    3 142.08us 5.0270us 249.26us cudaMemPrefetchAsync

--
==2285986== Unified Memory profiling result:
Device "Quadro RTX 5000 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      6    2.0000MB  2.0000MB  2.0000MB  12.00000MB  1.042556ms  Host To Device
     72    170.67KB  4.0000KB  0.9961MB  12.00000MB  1.025625ms  Device To Host
Total CPU Page faults: 72
swarnendu@nilgiri:~/cuda-examples/src/uvms$
```

Explicit Memory Hints in UVM

Advise runtime on expected memory access behaviors with `cudaMemAdvise(ptr, count, hint, device)`

- Possible hints

- `cudaMemAdviseSetReadMostly` Specify read duplication

- `cudaMemAdviseSetPreferredLocation` Suggest best location

- `cudaMemAdviseSetAccessedBy` Suggest mapping

- Hints do not trigger data movement by themselves

Explicit Memory Hints in UVM

`cudaMemAdviseSetReadMostly`

- Data will usually be read-only
- UVM will make a “local” copy of the data for each compute device that uses it
- A write will invalidate all copies except the one written

`cudaMemAdviseSetPreferredLocation`

- Suggests which compute device is the best location for the data
- If possible, a remote data mapping will be created when other processors touch it
- If mapping is not possible, data is migrated

`cudaMemAdviseSetAccessedBy`

- Does not cause movement or affect location of data
- Indicated compute device receives a remote mapping to the data
- If the data is migrated, mapping is updated

Driver-based Prefetching

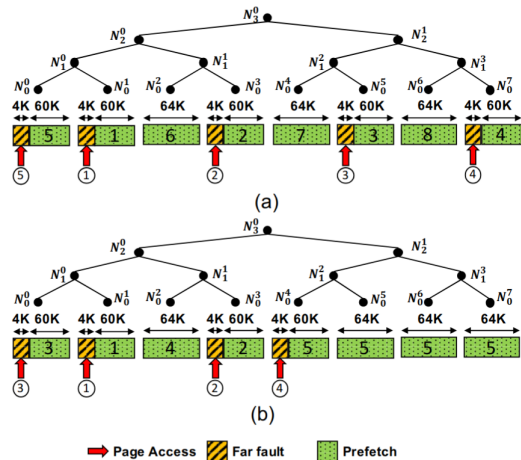
- Providing user hints per-buffer is complicated and error-prone
 - ▶ Set of optimal hints depend on the memory access pattern of the buffer, degree of oversubscription, and the architecture
- CUDA driver can implement different prefetch policies: (i) random, (ii) sequential, or (iii) locality-aware
 - ▶ For example, “random” prefetches a random 4 KB page from the 2 MB large page boundary along with the 4 KB faulting page
- NVIDIA implements a locality-aware tree-based neighborhood prefetcher
 - ▶ Migrate multiples of 4+60 KB blocks contiguous in the virtual address space grouped in a single transfer
 - ▶ All pages being prefetched are local to the current faulty page and are within 2 MB large page boundary (VA Block)

D. Ganguly et al. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. ISCA 2019.

Mao Lin et al. Forest: Access-aware GPU UVM Management. ISCA 2025.

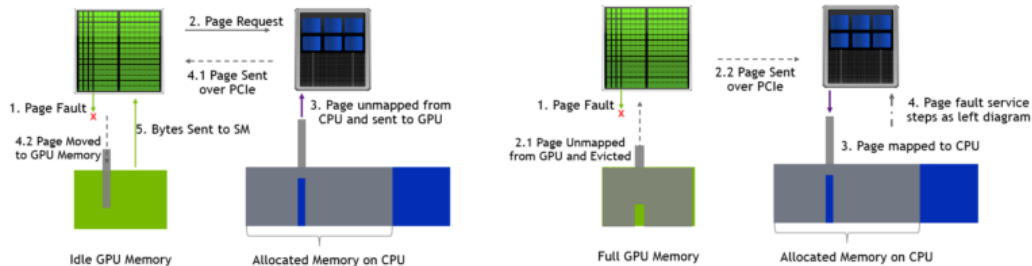
Tree-based Neighborhood Prefetcher

- Allocation with `cudaMallocManaged()` is logically divided into 2 MB large pages
- The 2 MB pages are further divided into logical 64 KB basic blocks to create a full binary tree
- If the user-specified allocation request is not a multiple of 2 MB, the remainder allocation is rounded up to the next $2i * 64 \text{ KB}$
 - ▶ If the requests are for 4 MB and 192 KB, then two 2 MB trees and 1 256 KB trees are created



Memory Oversubscription

- UVM support allows GPUs to oversubscribe memory
- With oversubscription, a memory page is first evicted from GPU memory to system memory, followed by transfer of requested memory from CPU to GPU

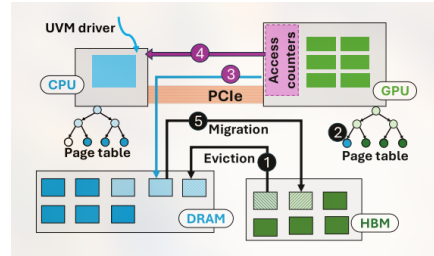


⚠ Limitations of TBN

- Same policy is used for all managed buffers across applications regardless of the memory access patterns
 - The UVM driver selects eviction candidates based on page fault information, and the not by actual page accesses
-
- Aggressive prefetching under memory constraint can be counter-productive, may cause displacement of heavily-accessed pages leading to thrashing
 - CUDA drivers implement LRM 2 MB page replacement policy
 - ▶ LRM does not imply the evicted page is inactive
 - Penalty of faults are greater under oversubscription
 - ▶ Threads need to be stalled for writing back pages along with the latency to migrate new pages
 - An option is to disable prefetching on oversubscription

Access Counter-Based Page Migration in NVIDIA UVM

- Latency of fault servicing and page migration is on the critical path of execution
 - ▶ Memory hints and prefetching may not be able to deal with irregular memory accesses
 - ▶ UVM driver is unable to distinguish heavily-accessed and lightly-accessed pages because it can only see far faults
- NVIDIA GPUs (Volta onward) have 256 hardware access counters that track DRAM access from the GPU over the PCIe interconnect
- The driver proactively migrates the hot pages to GPU global memory
- The tracking granularity (i.e., 64 KB, 2 MB, 16 MB, or 1 GB) and threshold (1–65535) are configured at driver load time



Utility of UVM

- UVM provides a coherent view of a single virtual address space between CPUs and GPUs with automatic data migration via demand paging
 - + Allows GPUs to access a page that resides in the CPU memory as if it were in the GPU memory
 - + Allows applications to run without worrying about the device memory capacity
- UVM aims to improve programmer productivity
 - + Code is less verbose, makes it easy to work with nested data structures (think of C++ classes with dynamically allocated attributes)
- UVM kernels may have poorer performance
 - Negative is the substantial cost of address translation overhead and demand paging

```
struct dataElem {  
    int prop1;  
    int prop2;  
    char *name;  
}
```

Case Study: Sparse Matrix-Vector Multiplication

Sparse Linear Algebra

Operations Involving Sparse Matrices

- Many real-world applications involve **sparse** matrix computation
 - ▶ Small fraction of non-zeros (NZ) relative to the matrix dimensions $M \times N$ (i.e., $NZ \ll M \times N$)
- Using a 2D array to store a sparse matrix with large values of M and N is wasteful
- Several sparse matrix representations have been proposed (e.g., CSR, ELLPACK, and CSR5)
 - + Aims to reduce memory wastage
 - + Aims to optimize the data layout according to the requirements of the compute device

COO and CSR Encodings

COO stores each non-zero element along with its row and column indices

Sparse Matrix A

$$A = \begin{bmatrix} 1 & 0 & 0 & 7 \\ 0 & 2 & 0 & 0 \\ 3 & 0 & 4 & 0 \end{bmatrix}$$

Values [1, 7, 2, 3, 4]

Row Indices [0, 0, 1, 2, 2]

Col Indices [0, 3, 1, 0, 2]

CSR compresses the row array to avoid repetitions

Sparse Matrix A

$$A = \begin{bmatrix} 1 & 0 & 0 & 7 \\ 0 & 2 & 0 & 0 \\ 3 & 0 & 4 & 0 \end{bmatrix}$$

Values [1, 7, 2, 3, 4]

Col Indices [0, 3, 1, 0, 2]

Row Pointers [0, 2, 3, 5]

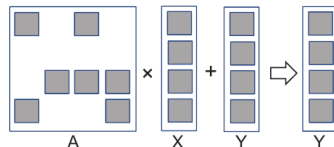
Sparse Matrix-Vector Multiply on CPUs

COO

```
for (int i = 0; i < num_elem; row++)  
    y[row_index[i]] += data[i] * x[col_index[i]];
```

CSR

```
for (int row = 0; row < num_rows; row++) {  
    float dot = 0;  
    int row_start = row_ptr[row];  
    int row_end = row_ptr[row+1];  
    // Loop can be parallelized  
    for (int elem = row_start; elem < row_end; elem++) {  
        dot += data[elem] * x[col_index[elem]];  
    }  
    y[row] += dot;  
}
```



A is a sparse matrix, and x and y are dense vectors

SpMV has irregular and indirect memory accesses and is memory-bound

Implementing SpMV GPU Kernel

```
__global__ void SpMV_CSR(int num_rows, float *data, int *col_index, int *row_ptr, float *
    x, float *y) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < num_rows) { // Number of rows may not be a multiple of the thread block size
        float dot = 0;
        int row_start = row_ptr[row];
        int row_end = row_ptr[row+1];
        for (int elem = row_start; elem < row_end; elem++) {
            dot += data[elem] * x[col_index[elem]];
        }
        y[row] += dot;
    }
}
```

Implementing SpMV GPU Kernel

```
__global__ void SpMV_CSR(int num_rows, float *data, int *col_index, int *row_ptr, float *  
    x, float *y) {  
    int row = blockIdx.x * blockDim.x + threadIdx.x;  
    if (row < num_rows) { // Number of rows may not be a multiple of the thread block size  
        float dot = 0;  
        int row_start = row_ptr[row];  
        in  
        fo  
    }  
    y[  
}  
}
```

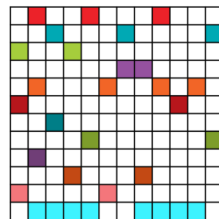
- Uncoalesced global memory accesses for data
- Possibility of significant control divergence or load imbalance

Mapping Threads to Work

CSR Scalar one thread per matrix row, i.e.,
one sparse matrix dot product

CSR Vector multiple cooperative threads
per matrix row

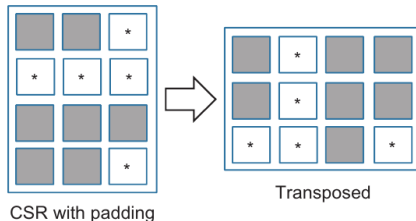
- Requires either shared memory or warp shuffle operations to reduce the partial values computed across threads



Neither CSR scalar nor CSR vector offer ideal performance

- CSR Vector will have no parallelism and reduce occupancy for short rows

Applying Padding and Transposition



```
__global__ void SpMV_ELL(int num_rows, float *data, int *col_index, int num_elem, float *  
    x, float *y) {  
    int row = blockIdx.x * blockDim.x + threadIdx.x;  
    if (row < num_rows) {  
        float dot = 0;  
        for (int i = 0; i < num_elem; i++) {  
            dot += data[row+i*num_rows] * x[col_index[row+i*num_rows]];  
        }  
        y[row] += dot;  
    }  
}
```

ELLPACK and HYB Encodings

ELLPACK is efficient for matrices with a **bounded** maximum number of non-zero elements per row

HYB combines ELLPACK and COO to reduce padding

Sparse Matrix A

$$A = \begin{bmatrix} 1 & 0 & 0 & 7 \\ 0 & 2 & 0 & 0 \\ 3 & 0 & 4 & 0 \end{bmatrix}$$

Values [1, 7, 2, 0, 3, 4]

Col Indices [0, 3, 1, -1, 0, 2]

Other extensions like JDS and SELLPACK have been proposed

ELL Fixed capacity K per row

COO Stores any excess entries exceeding K per row

Challenges With Sparse Algebra

- Irregular memory accesses
 - ▶ Uncoalesced global memory accesses for sparse matrix elements
 - ▶ Requires data-dependent indirect indexing of vector elements
- Load imbalance
 - ▶ Variable number of nonzeros per row leading to uneven work across threads
- Preprocessing overhead
 - ▶ Cost of format identification and conversion may offset performance benefits
- Tensor cores achieve higher arithmetic throughput, but require regular data layouts and dense operands
 - ▶ Accelerates MMA operations (e.g., $D = A \times B + C$) via the high-level **wmma** API or the low-level **mma** PTX interface, both operating at warp granularity
 - ▶ Wasted work processing zeros in sparse structures

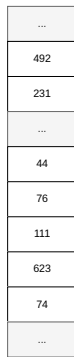
Case Study: Efficient GPU Hash Table

Random-access application

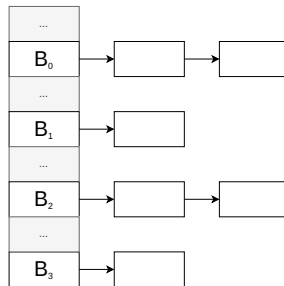
Resolving Collisions in a Hash Table

Closed addressing Each table entry refers to a set of items, called a bucket. Closed addressing is also known as chaining.

Open addressing Each table entry maps to a single item. Open addressing requires a deterministic probing scheme to search for free slots.



Open addressing



Closed Chaining

Probing Techniques

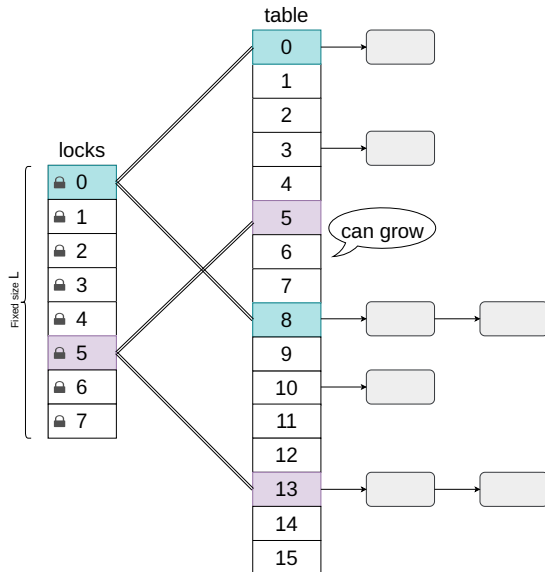
Let c be the number of entries of the hash table, l be the number of probing attempts, and $s(k, l)$ the l -th element in the probe sequence where $s(k, 0) = h(k)$

Linear probing Probing sequence is $s(k, l) = (h(k) + l) \bmod c$. Cache efficient but suffers from clustering.

Quadratic probing Probing sequence is $s(k, l) = (h(k) + l^2) \bmod c$. Incurs more cache misses but avoids primary clustering.

Chaotic probing Probing sequence is $s(k, l) = (h(k) + l \cdot g(k)) \bmod c$ where $g(k)$ is a second hash function. Incurs more cache misses but avoids clustering. Chaotic probing is also known as double hashing.

Designing Fine-Grained Concurrent Hash Table for CPU



- Each lock protects N/L buckets
- Allows more concurrency than coarse-grained locking

Resizing the Lock Array

Remember that a dynamically-sized hash table may require **resizing**

⚠ Why do we not resize the lock array?

- Associating a lock with every hash table bucket would incur large space overhead
- Accesses to the hash table are expected to be more distributed when the table is large implying low contention
- Resizing the lock array is complicated
 - Several locks may be in use, there can be concurrent resizes, need to avoid deadlocks
 - Delays threads that have initiated a concurrent operation
 - The hash index computed before resizing may no longer be valid after resizing

Designing Concurrent Hash Table on GPUs

Requirements

- Need to support high throughput for concurrent accesses to the hash table
- Need to devise nonblocking algorithms that are tuned to the GPU programming model for good performance

Challenges in designing an efficient hash table

- Lock-based synchronization will not scale to thousands of GPU threads
- Accessing linked-list-based data structures imply making random (uncoalesced) memory accesses from threads in a warp
 - ▶ Linked lists are cache-inefficient, each hash key is on a separate node with no locality
- Dynamically allocating linked list nodes for numerous GPU threads is a bottleneck

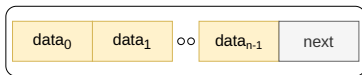
Hash Table with Closed Chaining

i Implementing horizontal chains using linked lists

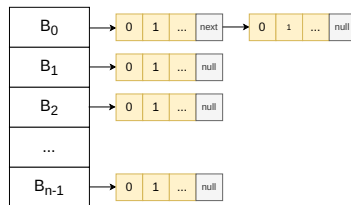
- Manipulating the list requires performing atomic updates on the next pointer



Linked list node



Slab list node



Hash table

SlabHash: Hash Table with Closed Chaining

SlabHash builds a concurrent GPU hash table using slab lists instead of linked lists

- A slab consists of multiple words of data (i.e., unordered set) and a single next pointer, thereby reducing memory overhead
 - ▶ Slabs on NVIDIA GPUs can be of 32 words
- Can search a slab node using a single ballot instruction

Uses a universal hash function of the form $h(k; a, b) = ((ak + b) \bmod p) \bmod B$

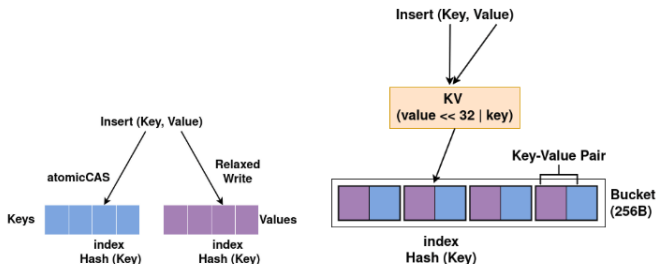
- a, b are random integers such that $1 \leq a < p$ and $0 \leq b < p$,
- p is a random prime number greater than the maximum allowed key,
- B is the number of buckets

Operations on a Slab List

- search** Search from the head of the list, load the next slab if the item cannot be found in the current slab
- insert** Start from the head of the list, use an atomic CAS to insert the new key-value pair into the first empty data slot. An unsuccessful CAS implies someone else occupied the empty slot, so restart looking for an empty slot. Load the next slab node if no empty slots are found. Allocate a new slab node at the end of the list if all slab nodes are full.
- replace** Similar to insert except that the entire slab list needs to be searched
- delete** Similar to insert except that the entire slab list needs to be searched (depending on whether duplicates are allowed)

Memory Layout of a Bucket

- A SoA layout requires an atomic CAS on the key slot and a conditional relaxed write to update the value
- With a packaged AoS, a single 64-bit CAS will suffice if both the key and value fits in 64-bits



More about SlabHash

Threads are assigned independent tasks (i.e., keys), but work is done in parallel per-warp, called warp-cooperative work sharing (WCWS)

- One-to-one mapping maps each thread to a single key, threads in a warp process their 32 keys individually
- + Advantage of WCWS is that it reduces stalls when compared to per-thread processing

- No locality across slab nodes
- Dynamic memory allocation at a massive scale is inefficient, requires a custom slab node allocator
- Lock-free insertion and deletion of nodes in linked lists is complicated because of ABA problem

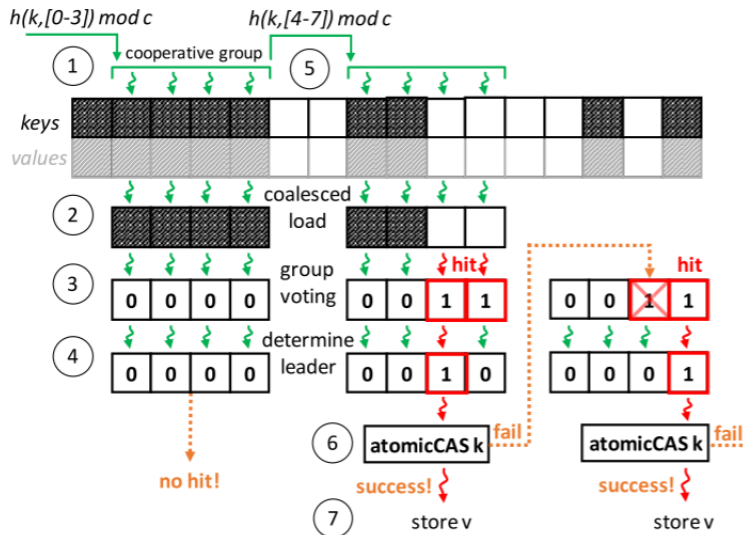
Hash Table with Open Addressing

Open addressing with parallel probing is more amenable to the SIMT execution model on GPUs

Parallel probing scheme

- Mapping each thread to a key will lead to a different probing sequence for threads in a warp, leading to non-coalesced global memory accesses
- Can use an entire warp of 32 threads per input key k , such that each thread with lane ID t probes a different hash table position $h(k, t) \bmod c$, but linear probing suffers from clustering
- WarpCore uses double hashing with an inner intra-warp linear probing, double hashing determines the starting offset for linear probing

Example of Insertion Operation with WarpCore



References I

-  D. Kirk and W. M. Hwu. Programming Massively Parallel Processors. Chapters 1–5, 7, 10, 13, 20, 3rd edition, Morgan Kaufmann.
-  T. Aamodt et al. General-Purpose Graphics Processor Architectures. Chapters 1–4, Springer Cham.
-  D. Patterson and J. Hennessy. Computer Organization and Design. Appendix C, 5th edition, Morgan Kaufmann.
-  J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. Section 4.4, 6th edition, Morgan Kaufmann.
-  NVIDIA Corporation. CUDA C++ Programming Guide, Release 12.9.2.
-  NVIDIA Corporation. CUDA C++ Best Practices Guide, Release 12.9.
-  NVIDIA Corporation. CUDA Runtime API: API Reference Manual, Release 12.9.
-  NVIDIA CUDA Compiler Driver, Release 12.9.
-  NVIDIA Corporation. CUDA PTX ISA, Release 9.3.

References II

-  T. Scudiero and J. Bentz. Understanding PTX, the Assembly Language of CUDA GPU Computing.
-  Viswa Ravichandran. CUDA PTX: Learning to Read NVIDIA's Virtual ISA.
-  Viswa Ravichandran. CUDA Register Mapping: From PTX to SASS.
-  Dheemanth. Intro to PTX Optimization: Part 1.
-  Nikolay Sakharnykh. Maximizing Unified Memory Performance in CUDA.