GPU Architecture and GPU Programming

Vito Wu

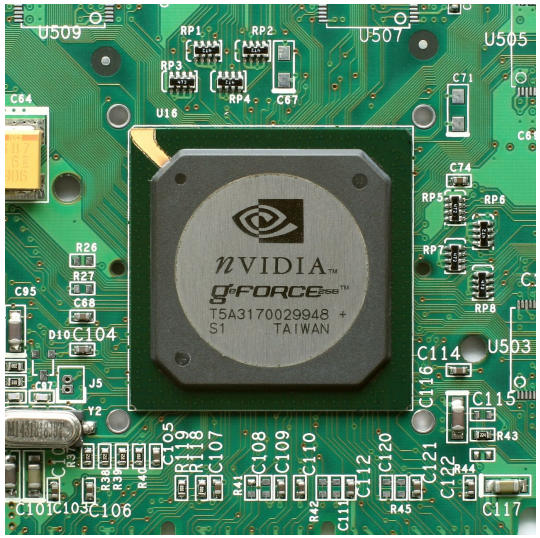
The Chinese University of Hong Kong, Shenzhen

October 25, 2020

Table of Content

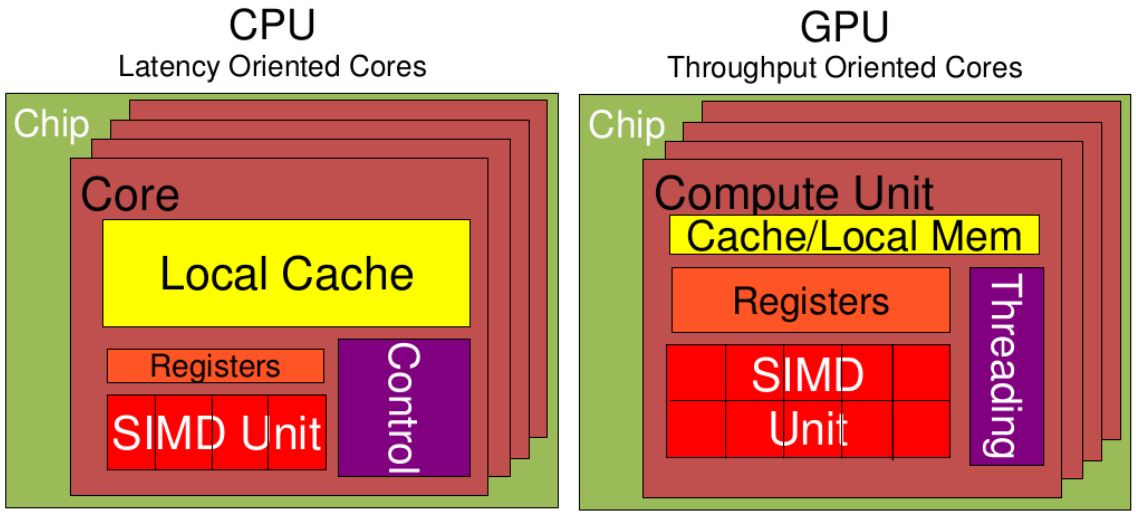
History of GPU

- In 1998, Geforce 256 was released by NVIDIA, which is marked as the first GPU in the world
- **3D Revolution (1995–2006):** Video, 2D GUI acceleration and 3D functionality were all integrated into one chip.
- **General Purpose GPU (2006-Present):** In 2007, Nvidia released its CUDA development environment, the earliest widely adopted programming model for GPU computing. Two years later, OpenCL became widely supported.
- The most powerful GPU in recent years are: NVIDIA A100 (Ampere architecture) and NVIDIA V100 (Volta architecture)



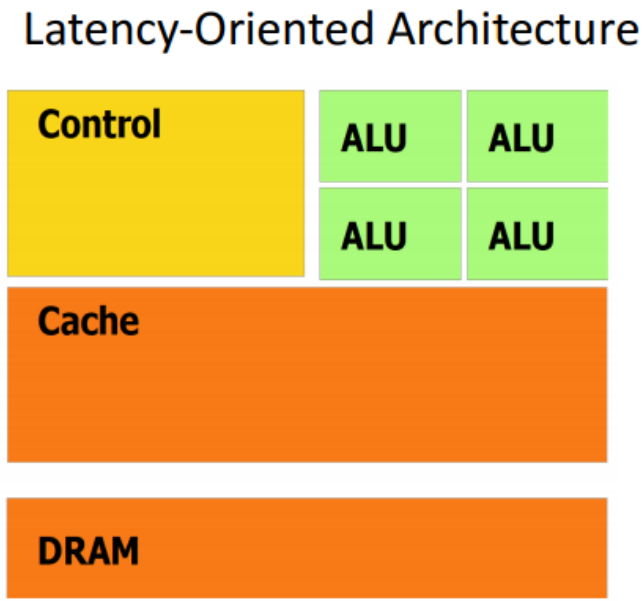
History of GPU

- **Comparison between GPU and CPU**
  - CPU: optimized for instruction latency
  - GPU: optimized for instruction throughput



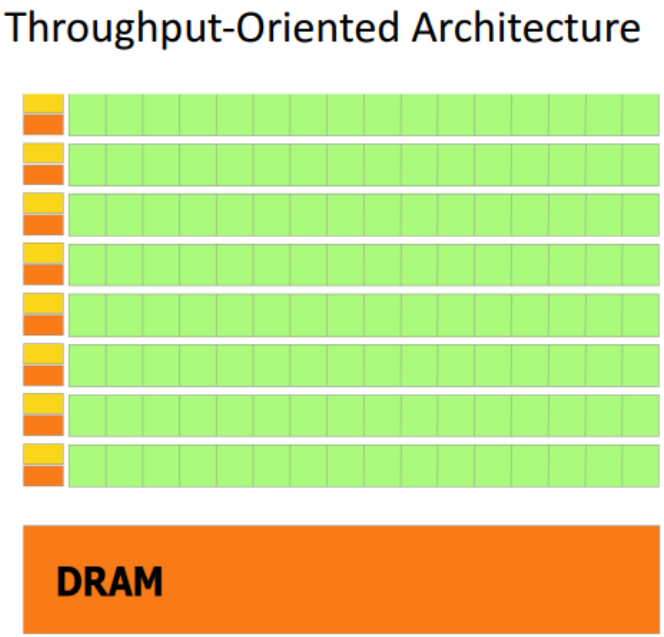
History of GPU

- **Comparison between GPU and CPU**
  - CPU: optimized for instruction latency
    - Powerful ALU (to reduce operation latency)
    - Large caches (to covert long latency memory accesses to short latency cache accesses)
    - Sophisticated control (branch prediction, data forwarding in pipeline)



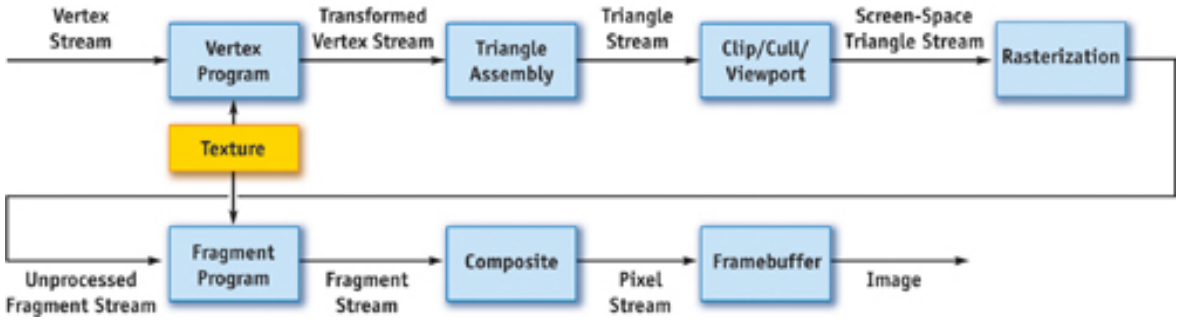
History of GPU

- **Comparison between GPU and CPU**
  - GPU: optimized for instruction throughput
    - Small caches (to boost memory throughput)
    - Simple control (no branch prediction, no data forwarding)
    - Energy efficient ALUs (long latency but heavily pipelined for high throughput)
    - **Require massive number of threads to hide the latencies**



History of GPU

- **Graphic Applications: Game, SFX, Animations, ...**
  - Rendering process is streaming and pipelinable

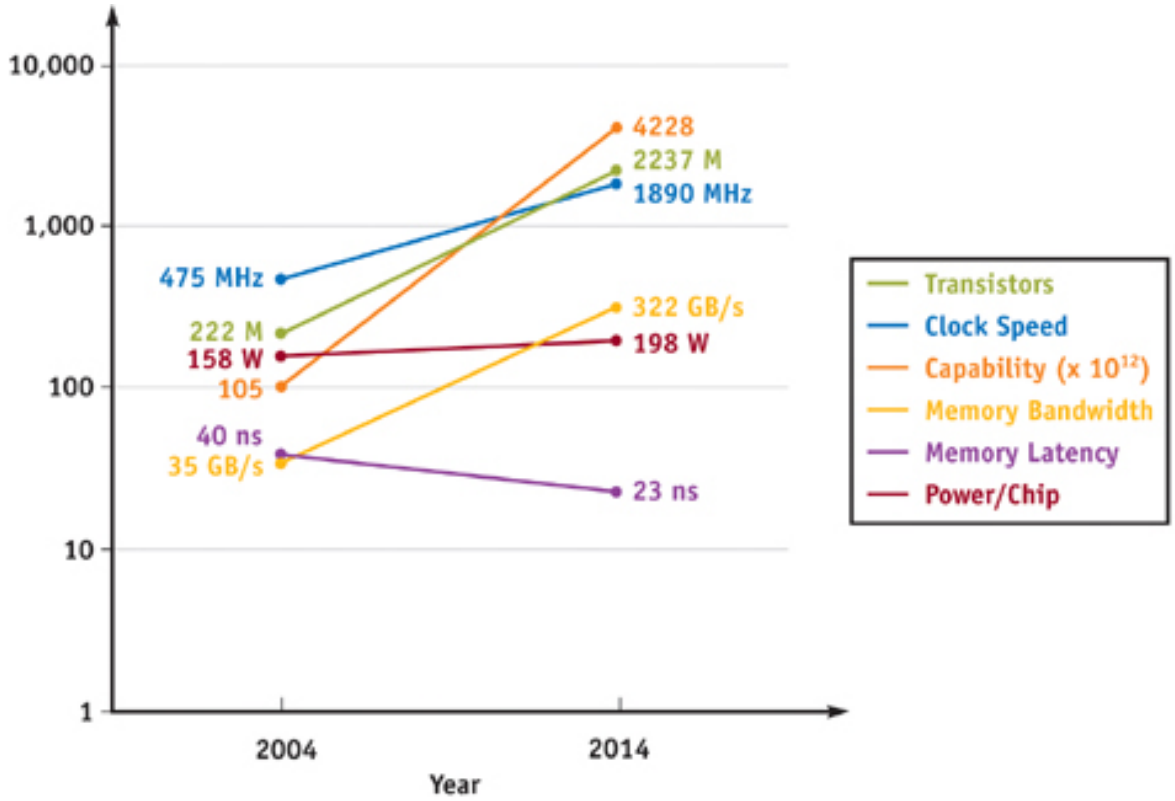


  - Rendering pipeline involves:
    1. Scene description
    2. **Vertex processing**
    3. **Rasterization**
    4. **Fragment processing**
    5. Output to screen

- **General Purpose Computing: DL, Crypto, Simulations, ...**
  - Characteristics of GPGPU applications
    - Huge amount of data to process (image processing, etc); or
    - Huge amount of simple operations (training and inferring model, bruteforce decryption, etc); or

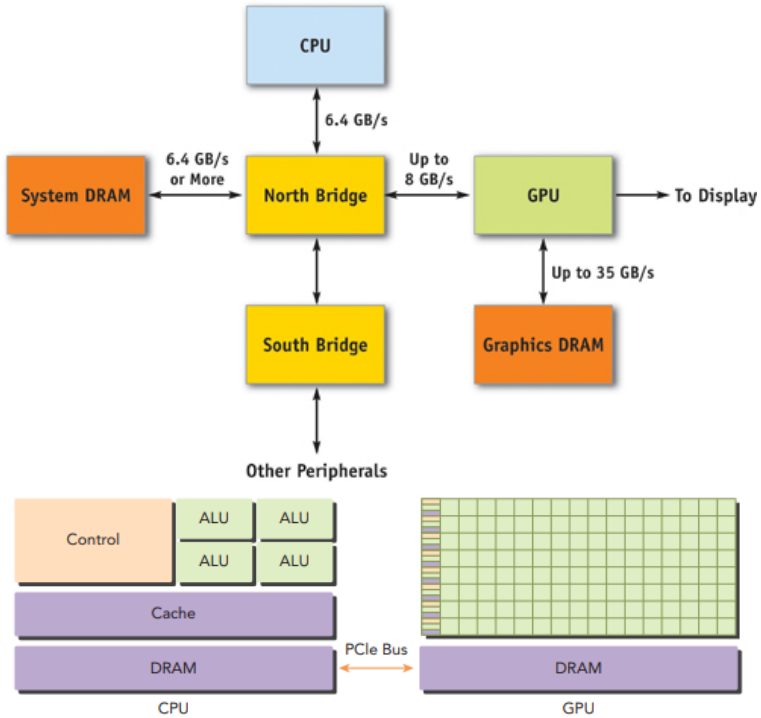
History of GPU

- **Technology Trends: Stream Computing is becoming the future!**
  - From **HW aspect**, compute ability increases 75% per year, but storage ability only increases 40%
    - The gap between memory throughput and instruction latency is larger and larger
  - From **SW aspect**, the scale of data to process is increased exponentially



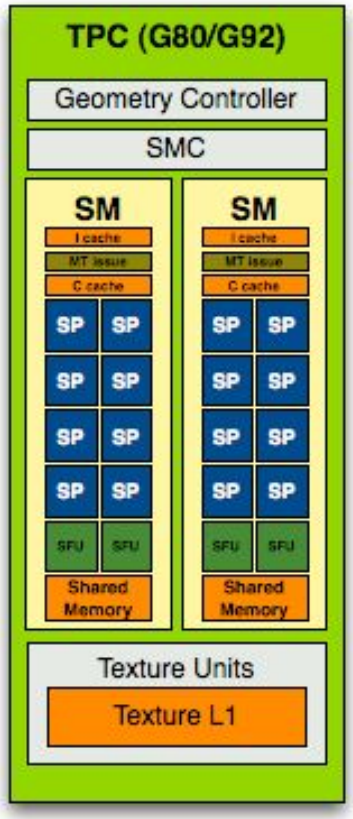
GPU Architecture

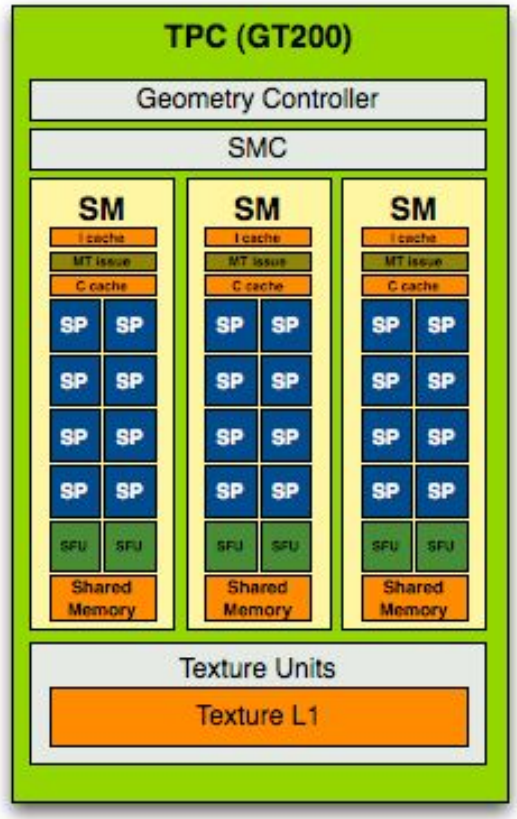
- **Heterogeneous Architecture**
  - A Heterogeneous Application consists of two parts
    - *Host code*
    - *Device code*
  - *Example: CUDA Program Structure*
    1. Allocate GPU memories
    2. Copy data from CPU memory to GPU memory
    3. Invoke CUDA kernel to perform program-specific computation
    4. Copy data back from GPU memory to CPU memory
    5. Destroy GPU memories



GPU Architecture

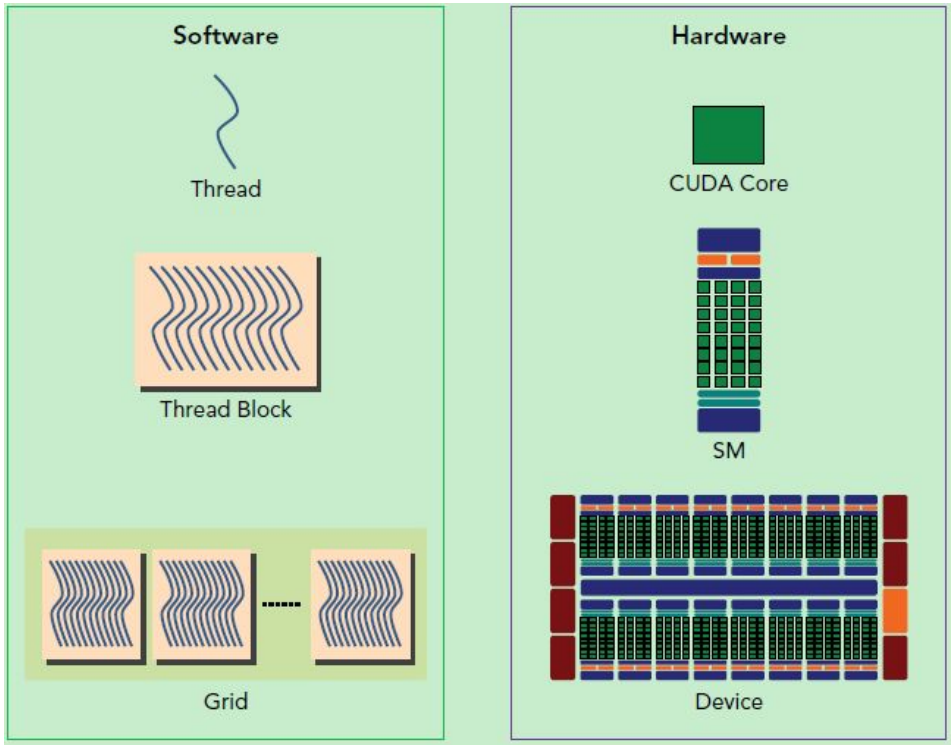
- **Streaming Multiprocessors**
  - The GPU architecture is built around a scalable array of *Streaming Multiprocessors* (SM)
  - A SM contains
    - Scalar processors
    - Shared Memory / L1 Cache
    - Register File
    - Load/Store Units
    - Special Function Units
    - Warp Scheduler





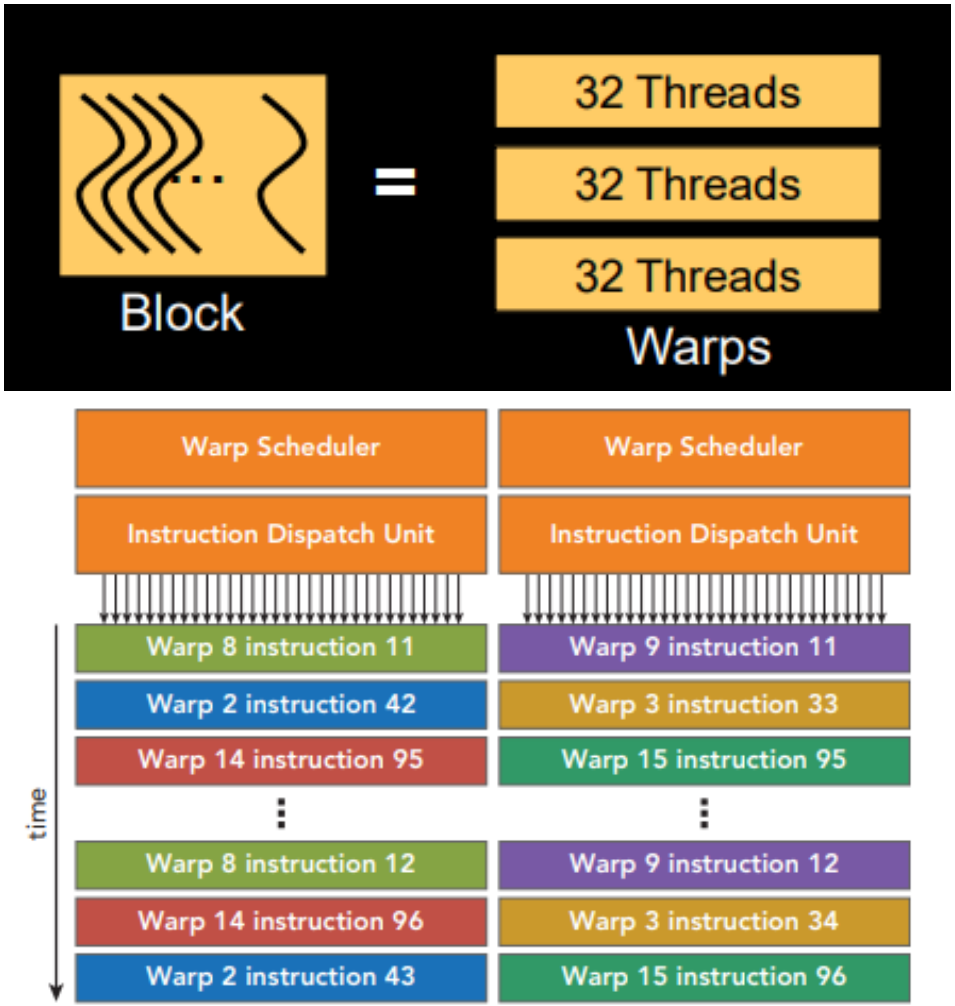
GPU Architecture

- **Execution Model** - Thread Organization
  - Threads are executed by scalar processors
  - Thread **Block**s are executed on multiprocessors (do not migrate)
  - Several concurrent thread blocks can reside on one multiprocessor
  - A kernel is launched as a **Grid** of thread blocks
  - Up to 16 kernels can execute on device at one time



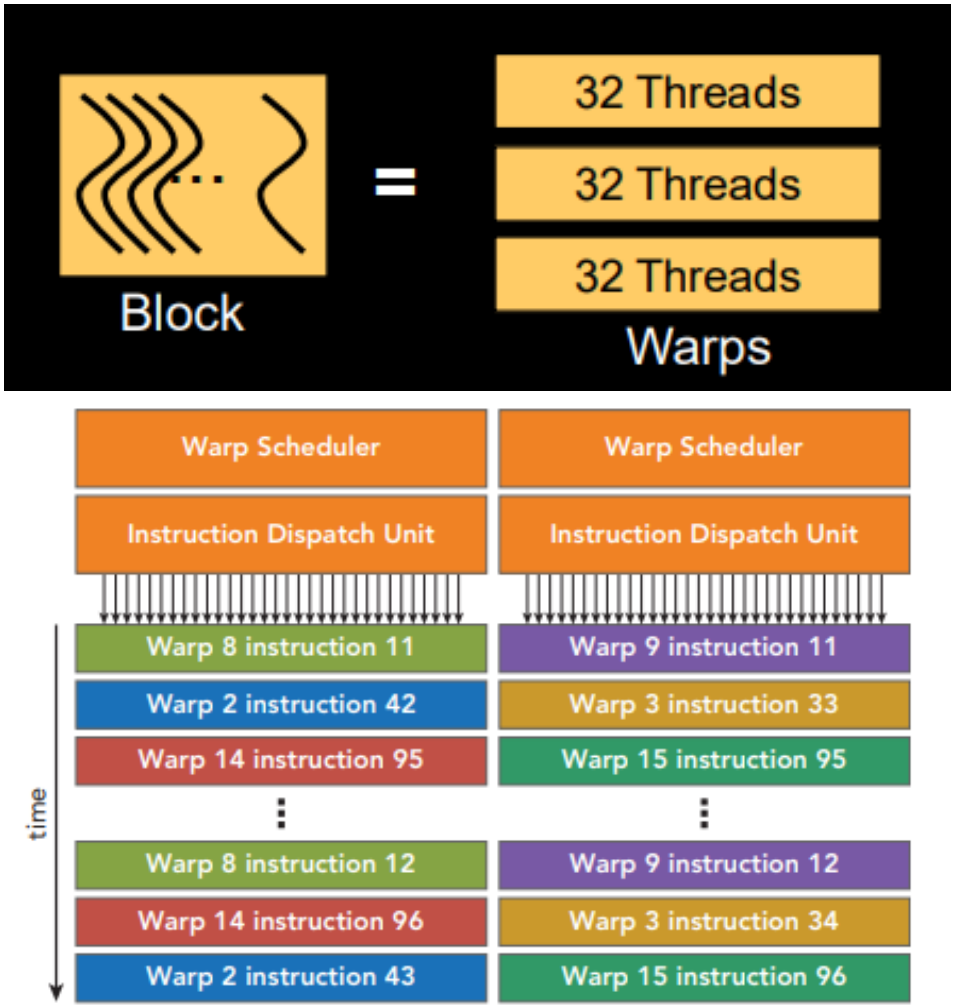
GPU Architecture

- **Execution Model - Warp** (or **work group** in AMD defined API) SIMD
  - Blocks divide into groups of 32 threads called **warps**
  - Warps are basic scheduling units (context switching is free)
  - A lot of warps can hide memory latency
  - Warps always perform the same instruction
    Each thread can execute its own code path



GPU Architecture

- 3 key features of GPU SIMD
  - Each thread has its own PC
  - Each thread has its own register state
  - Each thread has its own data path

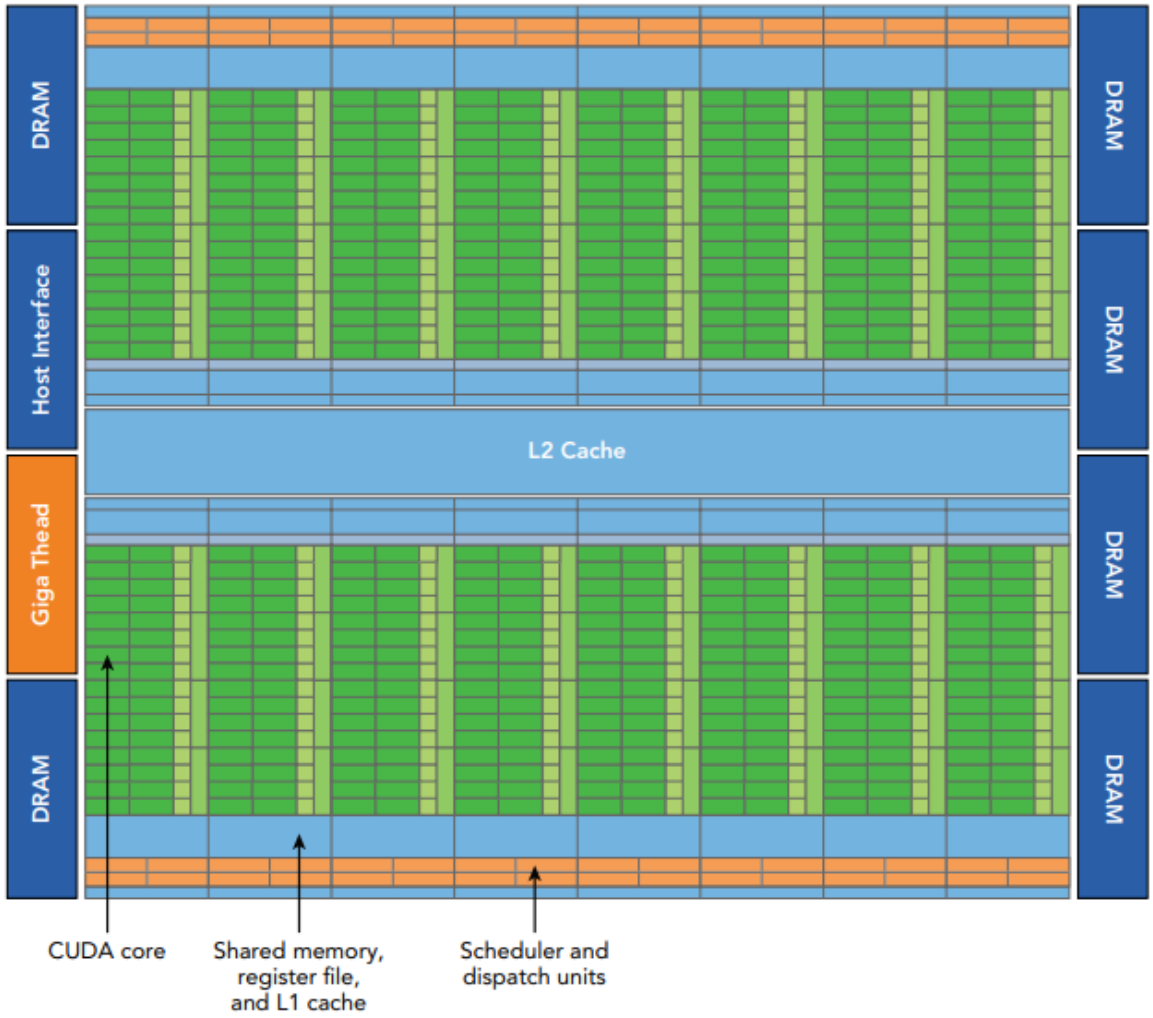


GPU Architecture

- **Execution Model** - Flow Control
  - **Basic Flow-Control Strategies**
    - Predication
    - Moving Branching up the Pipeline (e.g. z-buffer, only in rendering)
  - **Techniques to reduce flow control**
    - Static Branch Resolution (e.g. boundary conditions in simulations)
    - Precomputation (if the result of a branch was constant over a large domain of imput)

GPU Architecture

- **GPU Memory Hierarchy** - Off-Chip
  - Fermi Memory Hierarchy



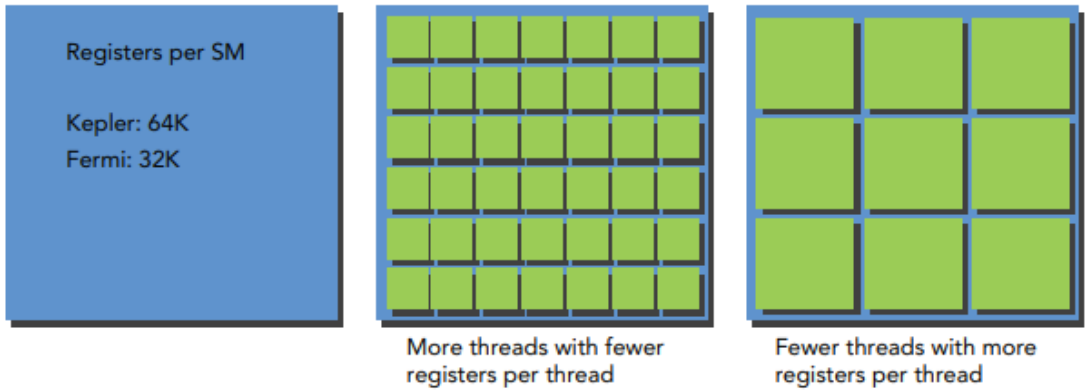
GPU Architecture

- **GPU Memory Hierarchy** - On-Chip

FIGURE 3.7

GPU Architecture

- **GPU Memory Hierarchy** - Register (R/W)
  - **Residential:** Streaming Multiprocessor (on-chip)
  - **Attribution:** 1 thread
  - **Lifetime:** thread
  - **Remark:**
    - fastest memory
    - 64K 32-bit register file per SM
    - partitioned among threads

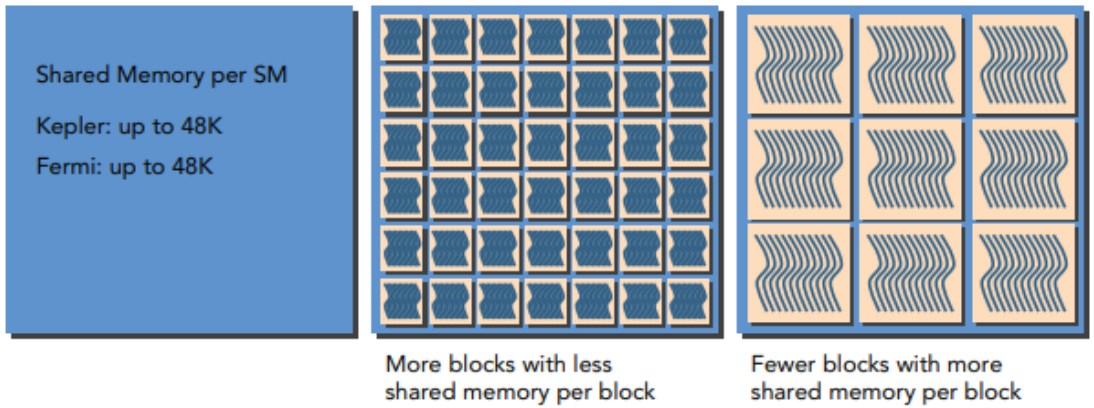


GPU Architecture

- **GPU Memory Hierarchy** - Local Memory (R/W, cached)
  - **Residential:** Off-chip
  - **Attribution:** 1 thread
  - **Lifetime:** thread
  - Variables in a kernel that are eligible for registers but cannot fit into the register space allocated for that kernel will spill into local memory.
  - Variables that the compiler is likely to place in local memory are:
    1. Local arrays referenced with indices whose values cannot be determined at compile-time.
    2. Large local structures or arrays that would consume too much register space.
    3. Any variable that does not fit within the kernel register limit.
  - **Remark:**
    - fast memory
    - 512 KB per thread

GPU Architecture

- **GPU Memory Hierarchy** - Shared Memory (R/W)
  - **Residential:** Streaming Multiprocessor (on-chip)
  - **Attribution:** All threads in one block
  - **Lifetime:** block
  - **Remark:**
    - fast memory
    - 64-100 KB per SM (align to device *computation capability*)
    - partitioned among threads
    - (in CUDA) Variables decorated with the following attribute in a kernel are stored in shared memory: *__shared__* (usually use with *__syncthreads()*)



GPU Architecture

- **GPU Memory Hierarchy** - Global Memory (R/W, cached)
  - **Residential:** Off-chip
  - **Attribution:** All threads + host
  - **Lifetime:** Host allocation
  - **Remark:**
    - Largest but also has the highest latency
    - (in CUDA) statically declared with *__device__*; or dynamically allocated with *__cudaMalloc()*

GPU Architecture

- **GPU Memory Hierarchy** - Texture and Constant Memory (R, cached)
  - **Residential:** Off-chip
  - **Attribution:** All threads + host
  - **Lifetime:** Host allocation
  - **Remark:**
    - 64KB + 128KB
    - Faster than Global memory
    - (in CUDA) statically declared with *__constant__*; or dynamically allocated with *cudaMemcpyToSymbol()*

GPU Programming

- **Graphic Rendering**: OpenGL, Vulkan
- **Compute**: CUDA (NV Only), OpenCL, OpenGL Compute Shader

Figure: Real Time Rendering

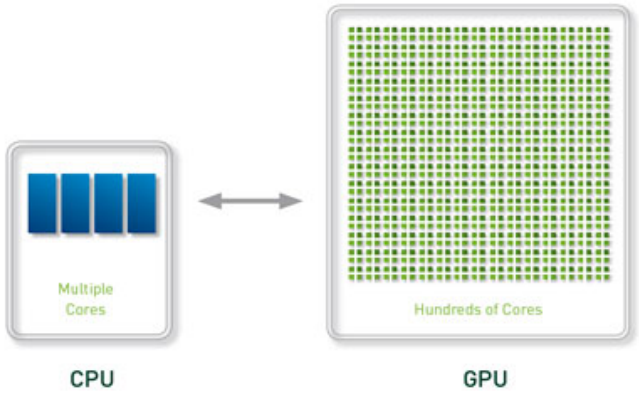


Figure: General Purpose Computing

CUDA Programming

- **Environment Setup**
  - *If you have an NVIDIA GPU:*
    - Install NVIDIA driver and get official CUDA Toolkit from: `https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html`
  - *Otherwise*
    - Follow this link to get a GPU emulator: `https://github.com/CUHKSZ-HPC/Seminars/blob/main/2020-10-25-GPUArchnProgramming/GPUEmulator.md`
  - After installation, you can try to compile and execute the demo programs to verify that the installation is successful

CUDA Programming

- **Host function and Kernel Function**
  - Function Type Qualifiers

QUALIFIERS	EXECUTION	CALLABLE	NOTES
`__global__`	Executed on the device	Callable from the host Callable from the device for devices of compute capability 3	Must have a `void` return type
`__device__`	Executed on the device	Callable from the device only	
`__host__`	Executed on the host	Callable from the host only	Can be omitted

  - Host code example

```
void sumArraysOnHost(float *A, float *B, float *C, const int N) {
   for (int i = 0; i < N; i++)
      C[i] = A[i] + B[i];
}
```

  - Device code example

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C) {
   int i = threadIdx.x;
   C[i] = A[i] + B[i];
}
```

  - Launch kernel

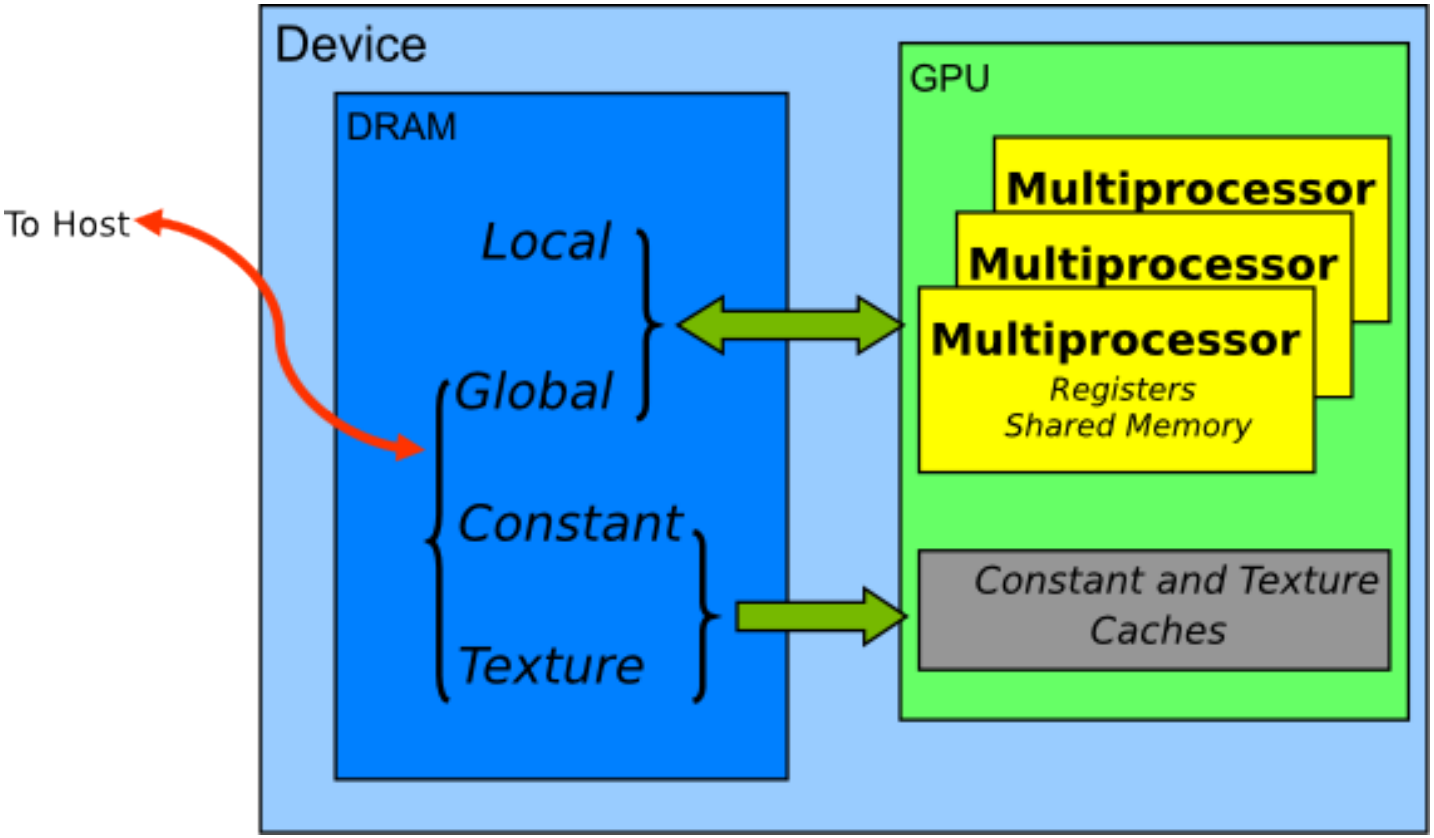
```
sumArraysOnGPU<<<1,32>>>(float *A, float *B, float *C);
```

CUDA Programming

- **Example 1: Get Device Info from CUDA API**
  `https://github.com/CUHKSZ-HPC/Seminars/tree/main/2020-10-25-GPUArchnProgramming/1_Get_Device_Info`
- **Example 2: Hello World!**
  `https://github.com/CUHKSZ-HPC/Seminars/tree/main/2020-10-25-GPUArchnProgramming/2_Hello_World`

CUDA Programming

- **Memory Management**



CUDA Programming

- **Memory Management**
  - memory allocation

```
cudaError_t cudaMalloc(void **devPtr, size_t count);
```

  - set memory

```
cudaError_t cudaMemset(void *devPtr, int value, size_t count);
```

  - memory deallocation

```
cudaError_t cudaFree(void *devPtr);
```

  - memory transfer

```
cudaError_t cudaMemcpy(void *dst, const void *src, size_t count,
                       enum cudaMemcpyKind kind);
```

  - type of *cudaMemcpy*

```
cudaMemcpyHostToHost
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice
```

CUDA Programming

- **Example 3: Simple Memory Allocation**
  `https://github.com/CUHKSZ-HPC/Seminars/tree/main/2020-10-25-GPUArchnProgramming/3_Memory_Management`
- **Example 4: Sum Arrays on GPU**
  `https://github.com/CUHKSZ-HPC/Seminars/blob/main/2020-10-25-GPUArchnProgramming/4_Sum_Array/SumArray.cu`
- **Example 5: Sum Arrays with Zero-Copy**
  `https://github.com/CUHKSZ-HPC/Seminars/blob/main/2020-10-25-GPUArchnProgramming/4_Sum_Array/SumArray0CP.cu`

GPU Program Optimization Techniques

- **Benchmarking Methods**
  - **1. Timer**
    - CPU Timer
    - CUDA Native Event Timer

```
cudaEvent_t start, stop;
float time;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord( start, 0 );
kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y,
                           NUM_REPS);
cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );

cudaEventElapsedTime( &time, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

GPU Program Optimization Techniques

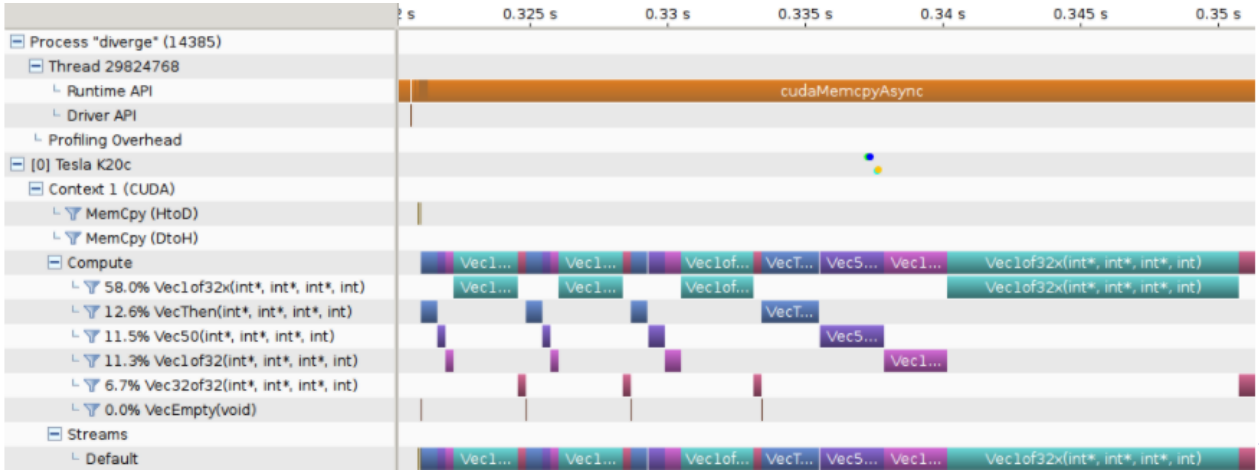
- **2. NV Profiler**
  `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`
  - *nvprof* command line tool

```
$ nvprof matrixMul
[Matrix Multiply Using CUDA] - Starting...
==27694== NVPROF is profiling process 27694, command: matrixMul
GPU Device 0: "GeForce GT 640M LE" with compute capability 3.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 35.35 GFlop/s, Time= 3.708 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: OK

Note: For peak performance, please refer to the matrixMulCUBLAS example.
==27694== Profiling application: matrixMul
==27694== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.94%  1.11524s       301  3.7051ms  3.6928ms  3.7174ms  void matrixMulCUDA<int=32>(float*, float*, float*,
int, int)
  0.04%  406.30us         2  203.15us  136.13us  270.18us  [CUDA memcpy HtoD]
  0.02%  248.29us         1  248.29us  248.29us  248.29us  [CUDA memcpy DtoH]
```

  - *Visual Profiler* GUI



GPU Program Optimization Techniques

- **Investigate the bottleneck of your program**
  - **Type 1: Memory Accessing Bandwidth**
    Symptom: memory bound; profiled memory throughput is much lower than the peak
    - Improve access pattern
    - Reduce redundant access
    - Use constant or texture memory
  - **Type 2: Latency**
    Symptom: both the memory and instruction throughputs are far from peak
    - Adjust resource usage to increase concurrency
  - **Type 3: Instruction Throughput**
    - Reduce # of instructions
    - Use instructions that have higher throughput
    - Avoid bad flow control

GPU Program Optimization Techniques

- **Coalesced Memory Access**
  - **Global Memory Latency: 400-800 cycles**
  - Coalesced memory access or memory coalescing refers to combining multiple memory accesses into a single transaction.
  - On the GPUs, every successive 128 bytes (32 single precision words) memory can be accessed by a warp (32 consecutive threads) in a single transaction.

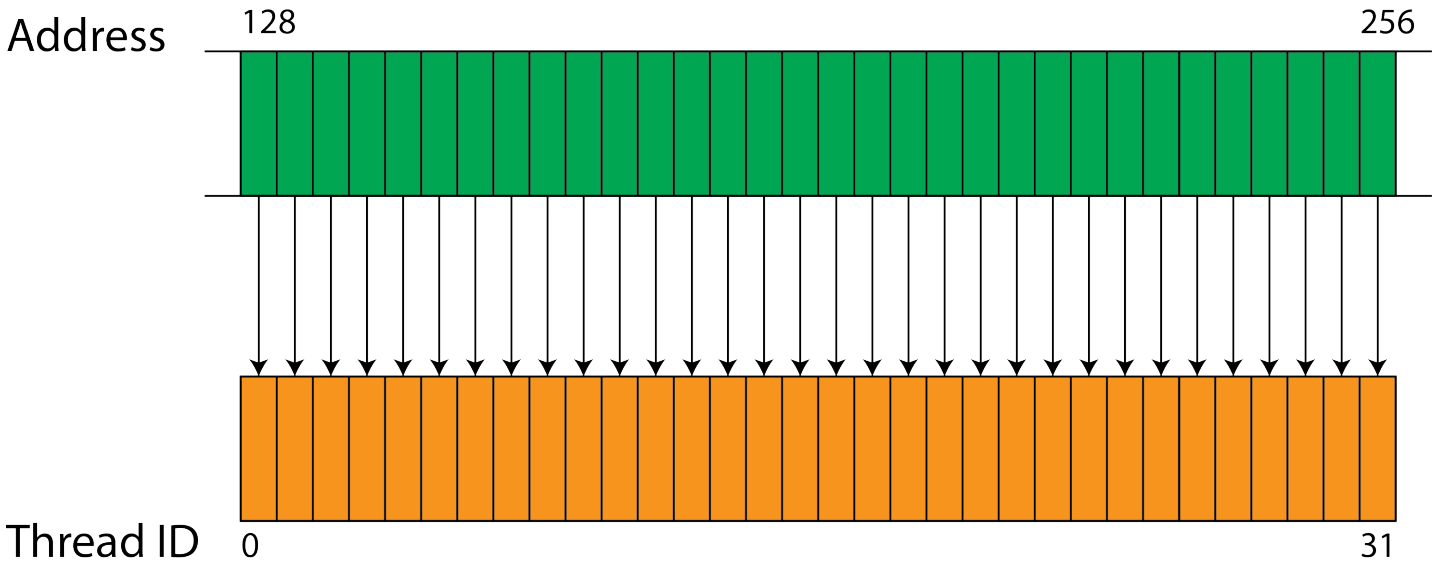


Figure: Aligned and Consecutive Access

GPU Program Optimization Techniques

- **Coalesced Memory Access**
  - **What condition will result in uncoalesced load?**
    - **Memory is not sequential**
    - **Memory access is sparse**
    - **Misaligned memory access**

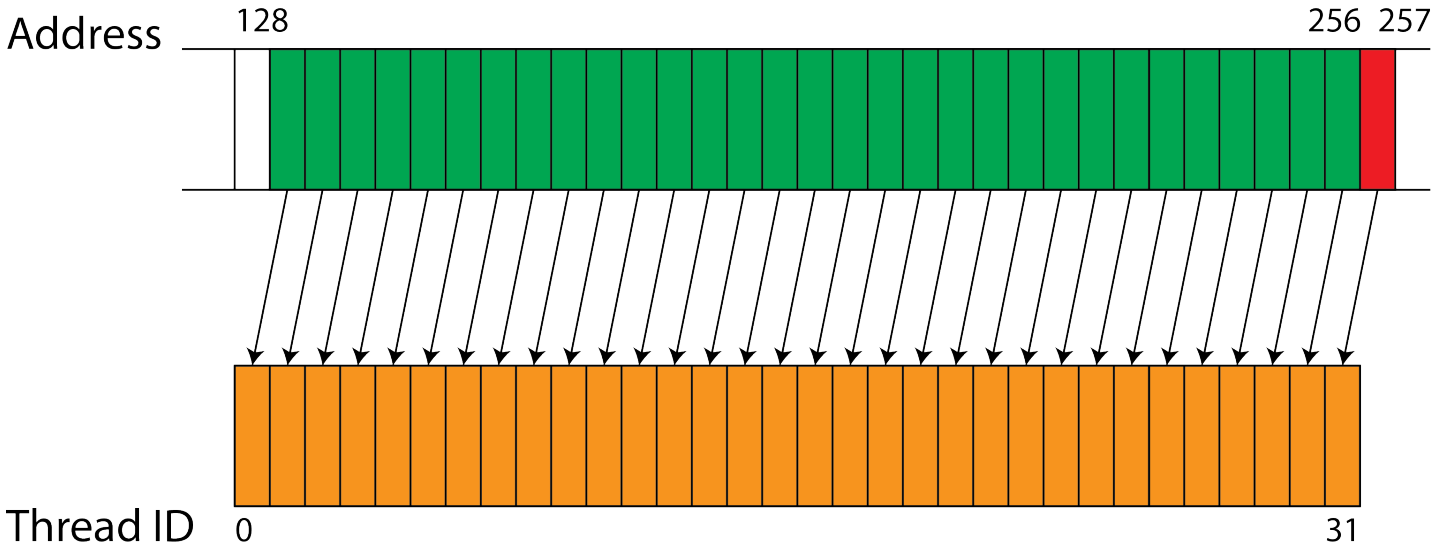


Figure: Misaligned Access

- In the case above, we need 2 transactions to load 128 byte memory

GPU Program Optimization Techniques

- **Coalesced Memory Access**
  - Good Coalesced Memory Access:
    *shmem[threadIdx.x]=gmem[blockIdx.x*blockDim.x + threadIdx.x];*
  - Not Coalesced Memory Access:
    stride=4;
    *shmem[threadIdx.x]=gmem[stride*blockIdx.x*blockDim.x+threadIdx.x*stride];*

GPU Program Optimization Techniques

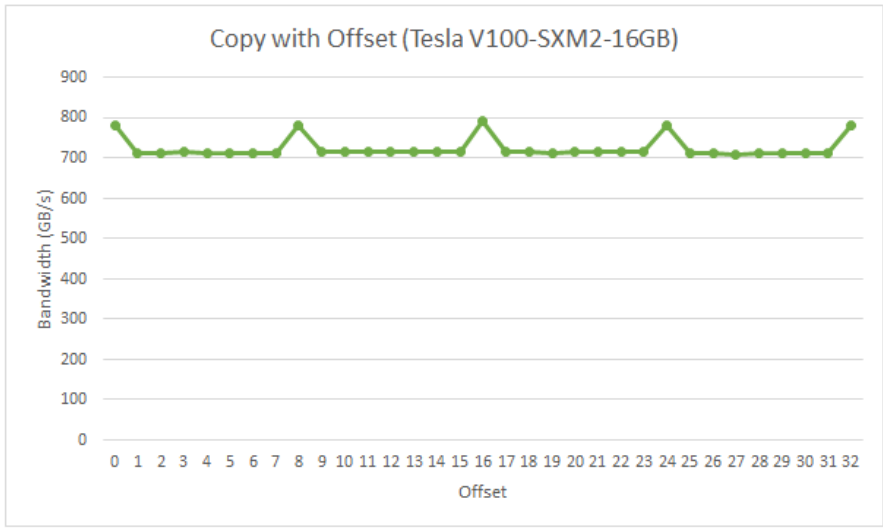
- **Coalesced Memory Access**
  - *Example from CUDA C Best Practice Guide*

A copy kernel that illustrates misaligned accesses

```
__global__ void offsetCopy(float *odata, float* idata, int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

In A copy kernel that illustrates misaligned accesses, data is copied from the input array `idata` to the output array, both of which exist in global memory. The kernel is executed within a loop in host code that varies the parameter `offset` from 0 to 32. (e.g. Figure 4 corresponds to this misalignments) The effective bandwidth for the copy with various offsets on an NVIDIA Tesla V100 (compute capability 7.0) is shown in Figure 5.

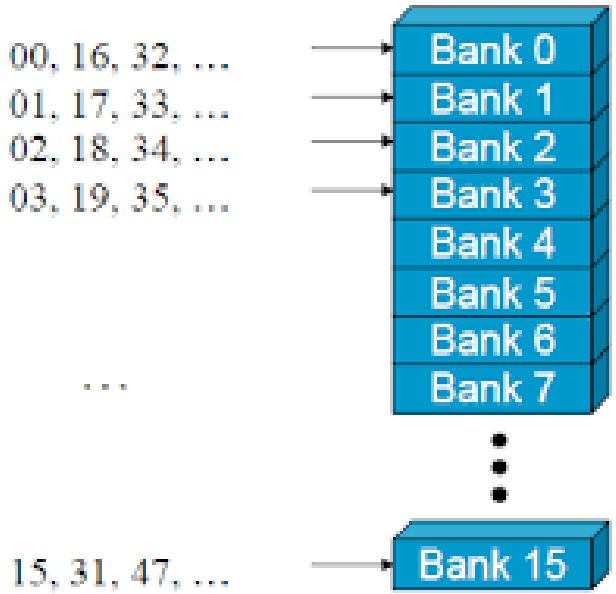
Figure 5. Performance of offsetCopy kernel



For the NVIDIA Tesla V100, global memory accesses with no offset or with offsets that are multiples of 8 words result in four 32-byte transactions. The achieved bandwidth is approximately 790 GB/s. Otherwise, five 32-byte segments are loaded per warp, and we would expect approximately $4/5^{th}$ of the memory throughput achieved with no offsets.

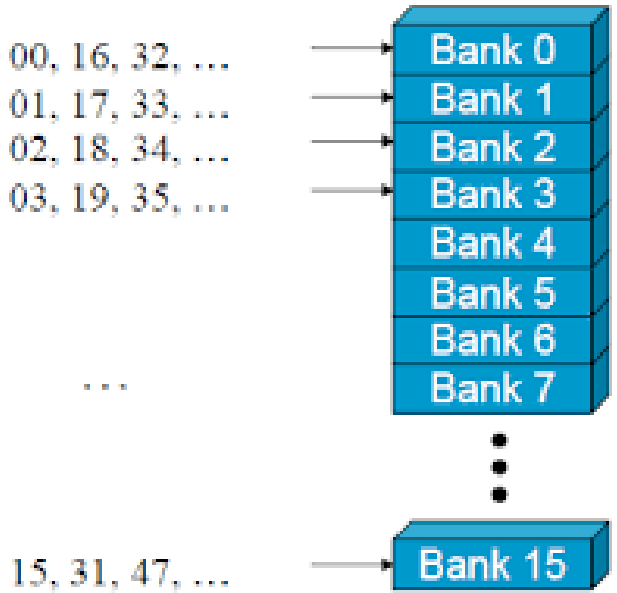
GPU Program Optimization Techniques

- **Bank Conflicts**
  - To achieve high memory bandwidth for concurrent accesses, shared memory is divided into equally sized memory modules, called banks that can be accessed simultaneously.
  - Therefore, any memory load or store of **n** addresses that spans n distinct memory banks can be serviced simultaneously.
  - **Special Cases**
    - If all threads in a warp **access the same word**, one broadcast, **no conflict**
    - If reading **continuous byte/double**, **no conflict**



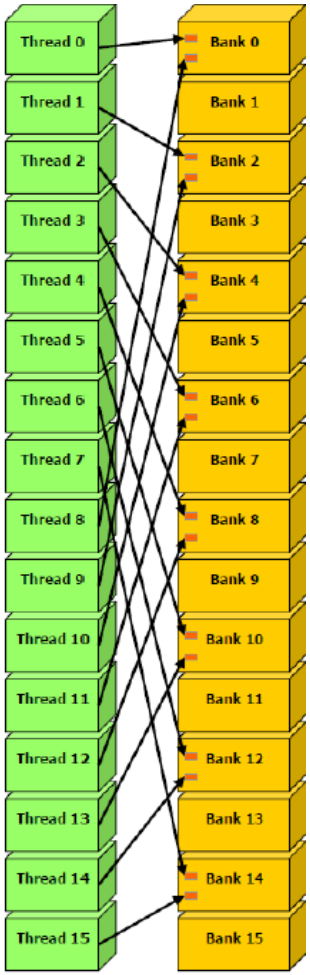
GPU Program Optimization Techniques

- **Bank Conflicts**
  - **No Bank Conflict**
    - Assume that we have an array on shared memory
      __shared__ int data[128]
    - Access pattern:
      int number = data[base + tid];



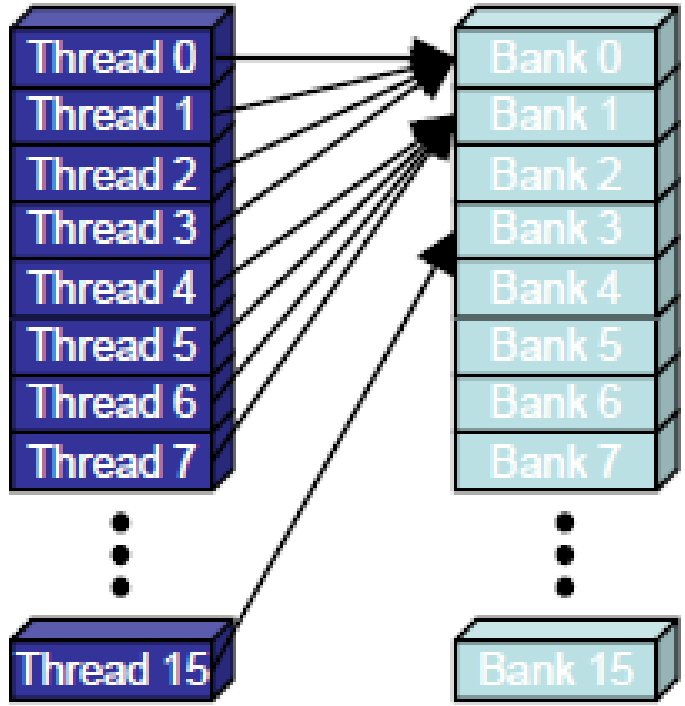
GPU Program Optimization Techniques

- **Bank Conflicts**
  - **2-Way Bank Conflict**
    - Assume that we have an array on shared memory
      __shared__ int data[128]
    - Access pattern:
      *int number = data[base + 2 * tid];*
  - **Only 1/2 shared memory throughput!!!**



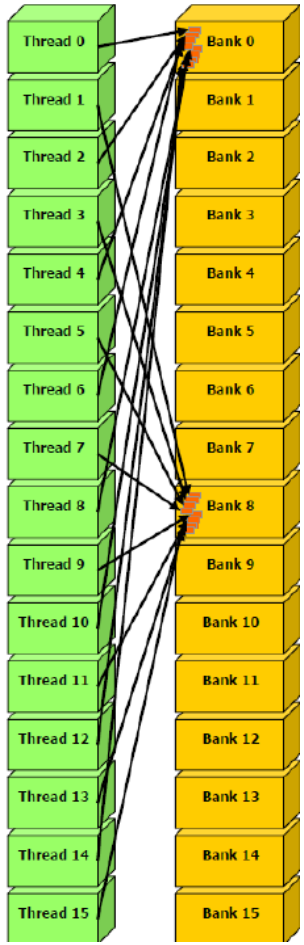
GPU Program Optimization Techniques

- **Bank Conflicts**
  - **4-Way Bank Conflict**
    - Assume that we have an array on shared memory
      __shared__ int data[128]
    - Access pattern:
      *int number = data[base + 4 * tid];*
  - **Only 1/4 shared memory throughput!!!**



GPU Program Optimization Techniques

- **Bank Conflicts**
  - **8-Way Bank Conflict**
    - Assume that we have an array on shared memory
      __shared__ int data[128]
    - Access pattern:
      *int number = data[base + 8 * tid];*
      or,
      int number = data[base];
  - **Only 1/8 shared memory throughput!!!**



GPU Program Optimization Techniques

- **Use Texture or Constant Memory**

```
__global__ void exampleKernel(int N,
    double * readOnlyArray,
    double * inputData,
    double *  outputData
    ){
    // kernel body
}
```

```
__global__ void exampleKernel(int N,
    const double * readOnlyArray,
    const double * inputData,
    double *  outputData
    ){
  // kernel body
}
```

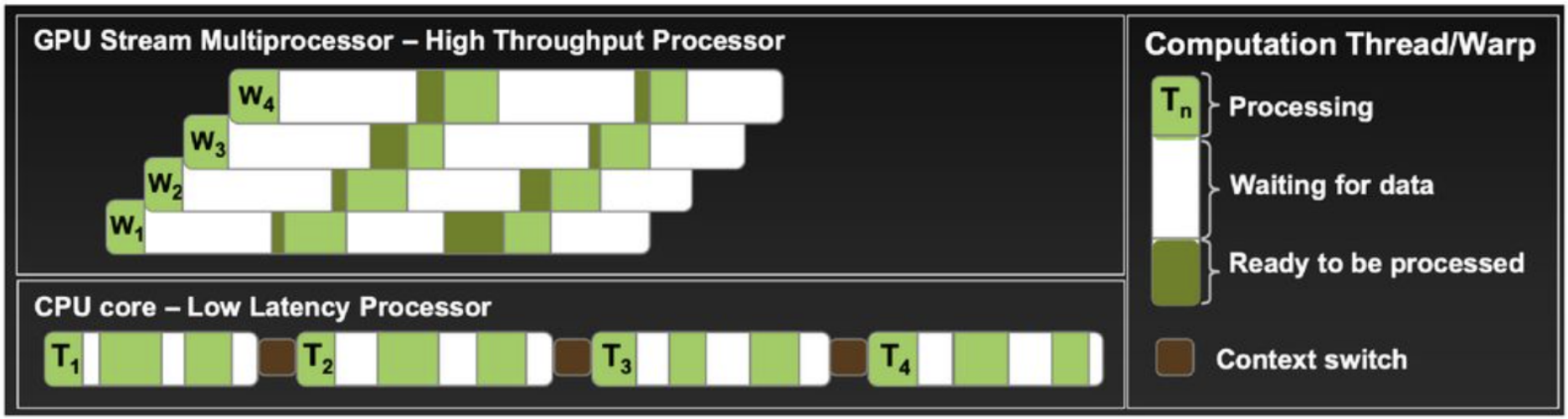
texture memory is almost as fast as constant memory, but it has a more complicated indexing method in order to align its unique prefetch mechanism

- **Conclusion on Memory Optimization**
  - Not-mentioned optimizations:
    - Array of Structure $\rightarrow$ Structure of Array
    - Padding
    - Change parallelism scheme: 1-thread-per-task $\rightarrow$ 1-warp-per-task
    - Use Memcpy instead of Zero-Copy
  - Utilize SHMEM to avoid GMEM access
  - GMEM Metric: achieve 70-80% of theoretical bandwidth is VERY GOOD

GPU Program Optimization Techniques

- **Latency Issues** - Where the latency comes from?
  - Instruction issue
  - Thread blocking
  - Memory accessing
    - GMEM: 400-800 cycles
    - Arithmetic: 18-22 cycles

- **General Solution:** **Launch more threads to hide the latency**



GPU Program Optimization Techniques

- **How to set Grid and Block size?**
  - **Rule 1**: # of Blocks >> # of Streaming Multiprocessors
  - **Rule 2**: Block size should be a multiple of 32 (suggestion: 128 or 256)
  - **Rule 3**: Do more experiments to set the optimize size!
- Overall, we need the occupancy to be high enough to hide latency
  - Assume GMEM takes 400 cycles, we need 200 arithmetic instructions to hide the latency
  - Assume code has 8 independent arithmetic instructions for every one global memory access, then around 26 warps would be enough (54% occupancy, given 48 active warps is maximum)

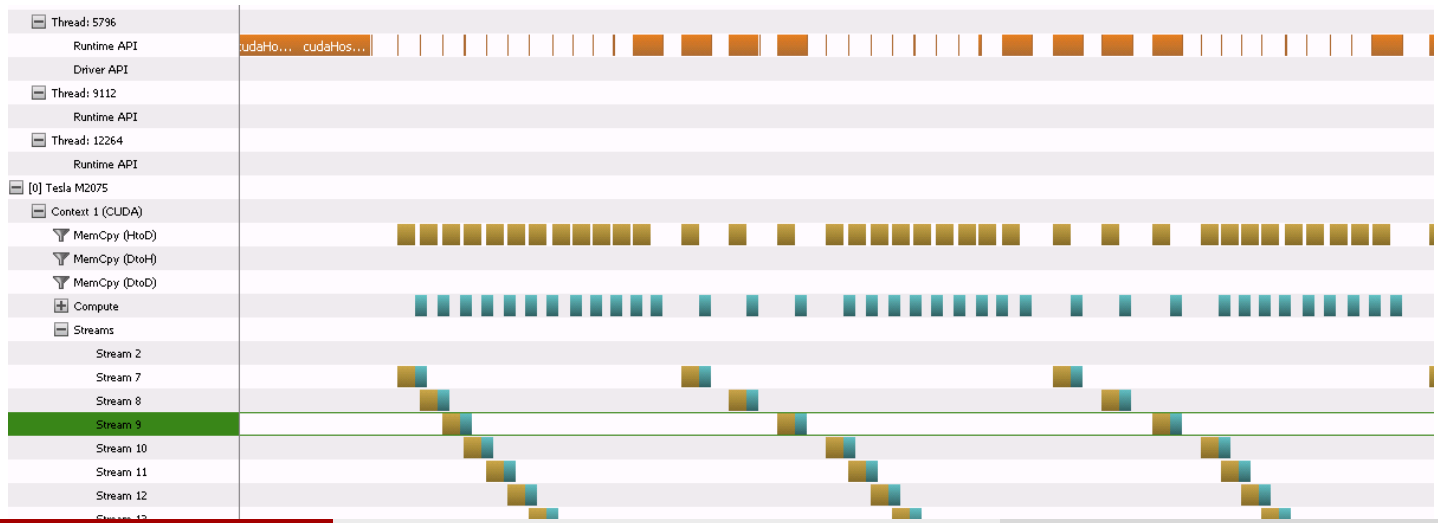
GPU Program Optimization Techniques

- **SM Resource Limitation**
  - Shared memory is partitioned among blocks
  - Registers are partitioned among threads: ≤ 63
  - Thread block slots: ≤ 8
  - Thread slots: ≤ 1536
  - **More Information on:** `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities`

- Any of those can be the limiting factor on how many threads can be launched at the same time on a SM

GPU Program Optimization Techniques

- **Instruction Level Optimization**
  - **Techniques on GPU is almost identical to those on CPU**
    - Use high throughput instructions (e.g. use shift instead of modulo and divide)
    - Reduce wasted instructions (branch divergence, bank conflict, etc.)
    - Avoid diverging with a warp
    - Exploit instruction level parallelism (data dependency, unroll loop, etc.)
    - (*) Use IEEE Fast-Math API (higher throughput but lower accuracy)
    - More other tricks, not going to discuss on today ...
- Other misc: Use STREAM COPY to pipelining CPU-GPU data transfer (not going to discuss on today) ...



Userful Resources

- **CUDA C Best Practice Guide**
  https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
- **GPU Gems**
  https://developer.nvidia.com/gpugems/gpugems/
- **CUDA C Programming Guide**
  https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- **Professional CUDA C Programming**
  http://www.hds.bme.hu/~fhegedus/C++/Professional%20CUDA%20C%20Programming.pdf