The Concepts Behind CUDA Optimization

Zafrir Patt

November 2018

1. CUDA Overview & Basic Terminology
2. CUDA Execution Model
3. CUDA Memory Model
4. Starting CUDA Optimization

CUDA Source of Information

- Work CUDA Programming Guide

  (Probably the most important and comprehensive resource for C/C++ programmers)

  https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html or the pdf version:

  https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf
- CUDA C Best practices Guide

  https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#abstract
- CUDA Experiments (CUDA Kernel-Level Experiments and CUDA Source-Level Experiments)
- CUDA Developer Zone: https://docs.nvidia.com/cuda/index.html#

  including code samples: https://developer.nvidia.com/cuda-code-samples
- Mark Harris blog: https://devblogs.nvidia.com/author/mharris/ ,

  https://devblogs.nvidia.com/even-easier-introduction-cuda/
- stackoverflow.com
- **and Lot of others . . .**
- Books archive: https://developer.nvidia.com/cuda-books-archive

1. CUDA Overview & Basic Terminology
2. CUDA Execution Model
3. CUDA Memory Model
4. Starting CUDA Optimization

CUDA Terminology 1: ILP and Pipelined Processor

- **ILP (Instruction-Level Parallelism)**

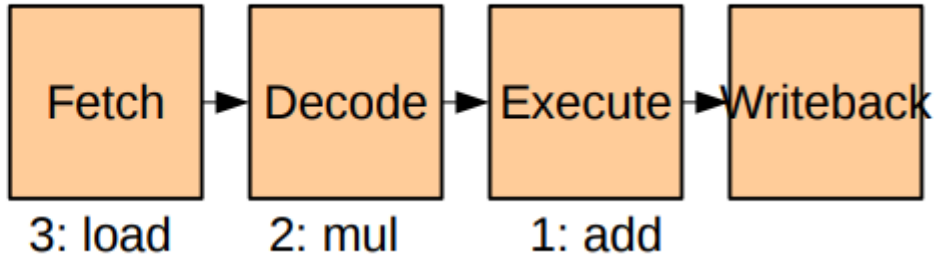
  Parallel/Overlapped execution of multiple independent instructions

- **Pipelined Processor**
  - Device that implement ILP:
    - Identify multiple independent instructions
    - Execute them in parallel/overlapped manner
  - Device that using ILP to hide instructions latency

See great presentation of Sylvain Collange at:
http://www.irisa.fr/alf/downloads/collange/cours/gpuprog_ufmg_2015/gpu_ufmg_2015_1.pdf

Program

```
1: add, r1, r3
2: mul r2, r3
3: load r3, [r1]
```



See:

https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf and CUDA Pipe Utilization

CUDA Terminology 2: Basic

- **Host**: the CPU and its memory
- **Device**: the GPU and its memory
- **SM (Streaming Multiprocessor)**: Independent Processing Unit. Each device contains several SM's
- **Compute Capability (CC)**: Define the **SM** version.

  Determine the hardware features and the available instructions. Comprises of:

  - a major revision: The core architecture
  - a minor revision: Specify an incremental improvement over the core architecture

  see https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability

  https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

  - **Does our device CC is supported by our CUDA SDK?**
    - The *Tesla* and *Fermi* architectures are no longer supported starting with CUDA 7.0 and CUDA 9.0, respectively . .
  - We should specify to which CC the nvcc will compile (can compile for multiple CC's)

Nvidia Device Architecture

Architecture	Released in	Compute Capability
Tesla	2006	1.x
Fermi	2010	2.x
Kepler	2012	3.x
Maxwell	2014	5.x
Pascal	2016	6.x
Volta	2017	7.x
Turing	2018	7.5

See https://www.nvidia.com/en-us/about-nvidia/corporate-timeline/

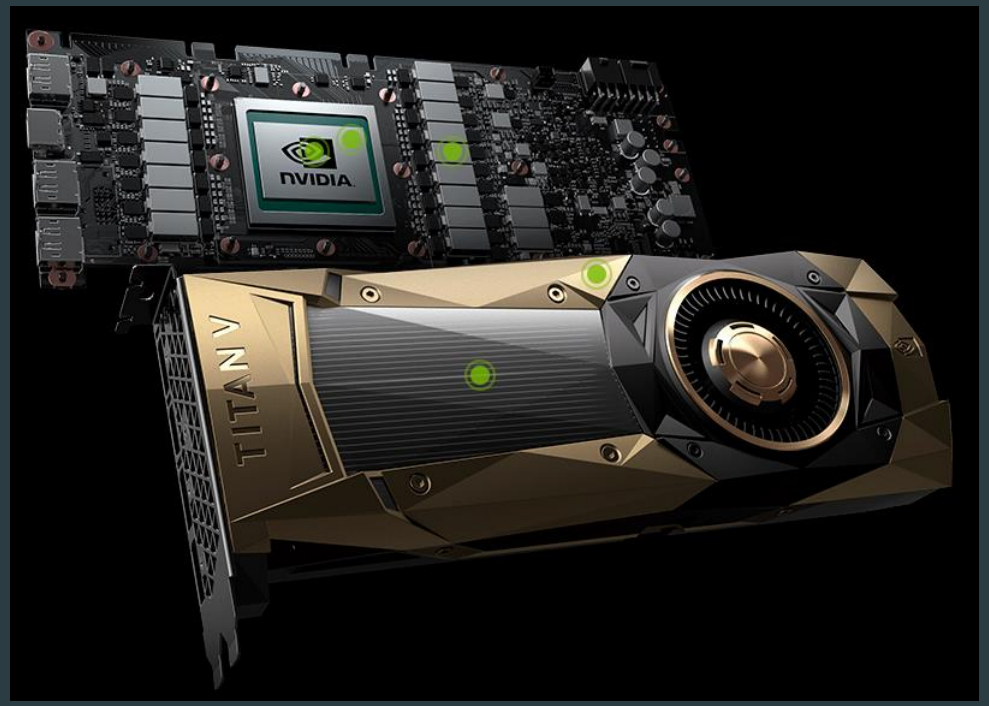
Nvidia Compute Capability

	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.5
Architecture	Kepler				Maxwell			Pascal			Volta	Turing
#fp32 cores	192				128			64	128		64	
#fp64 cores	8		64		4			32	4		32	
Special Function Units	32				32			16	32		16	
32-bit registers	64K			128K	64K			64K			64K	
Max shared Memory	48KB			112KB	64KB	96KB	64KB	64KB	96KB	64KB	96KB	64KB
Max blocks per SM	16				32			32			32	16
Max warps per SM	64				64			64			64	32
# warp schedulers	4				4			2	4		4	

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications__technical-specifications-per-compute-capability

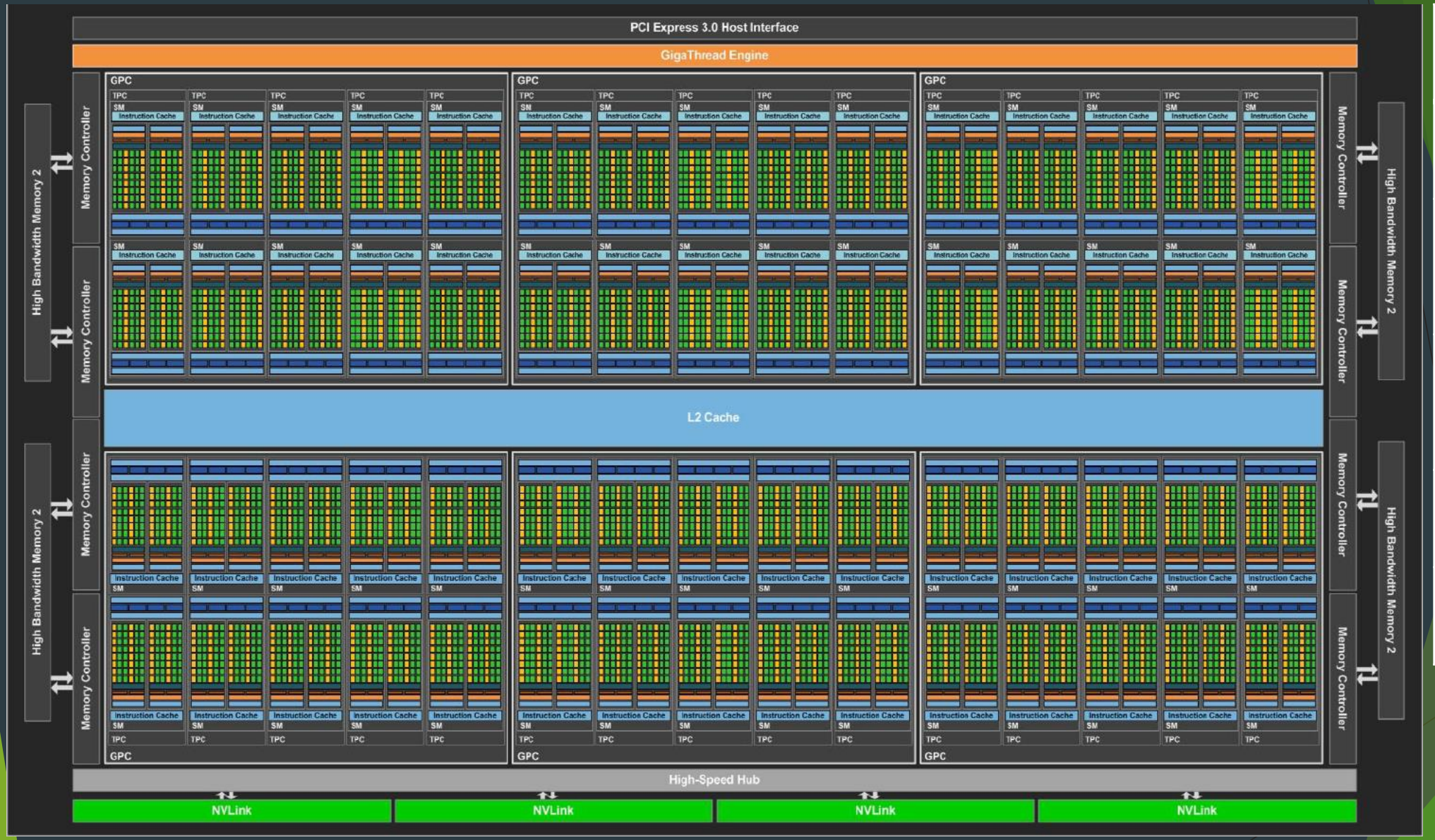
A CUDA Device Architecture

- Main components of CUDA device:
  - Several SM's (Streaming Multi-Processors)
  - GPU Global Memory (residing in the DRAM)
  - L2 cache: Shared by all SM's. Used mainly to cache the Global memory. Can also cache Local memory(*)



See:
https://developer.nvidia.com/cuda-gpus

(*): will be described in the following



PASCAL GP100	
Num SM's	56 (full: 60)
512-bit memory controllers	8
L2 cache/ memory controller	512KB
Total L2 cache	4MB
GPU memory	16GB

Pascal GP100 architecture (Figure 7) from:

https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

CC 6.0 SM is composed of

- Instructions cache
- 64 KB Shared Memory

  Fermi and Kepler had a 64 KB configurable Shared memory and L1 cache.

  From Maxwell, shared memory has a dedicated cache
- L1 / Texture cache

  L1 cache is used to cache Local memory but can also serve as a texture cache.

  (Global memory normally not cached in L1)
- Two processing blocks. Each one contains:
  - Instruction Buffer
  - Warps Scheduler
  - Dual Dispatch Unit
  - Registers
  - **Pipelined Processors Units**
    - 32 CUDA cores (FP32). can perform one single precision instruction per clock cycle
    - 16 Double Precision units (FP64)
    - 8 LDST Units (Load/Store): for Shared/global/local memory accesses
    - 8 SFU (Special Functions Units)

See: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability-6-x

Pascal SM (CC 6.0)	
FP32 Cores	64
FP64 Cores	32
registers	256KB
Shared Memory	64KB
Active Blocks	32
Active Threads	2048

Pascal GP100 SM Unit (Figure 8) from:

https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

Device attributes queries

- Getting the CUDA device properties:

  cudaDeviceProp prop;

  cudaGetDeviceProperties(&prop, deviceIndex);

- Getting specific attribute:
  - cudaDeviceGetAttribute (int* value, cudaDeviceAttr attr, int device)

    used to get an information of a specified device. If attr =
    - cudaDevAttrMaxThreadsPerBlock: Maximum number of threads per block
    - cudaDevAttrMaxSharedMemoryPerBlock: Maximum shared memory size to a thread block in bytes
    - cudaDevAttrTotalConstantMemory: Max constant memory available on device in bytes
    - cudaDevAttrGlobalL1CacheSupported: return 1 if device supports caching globals in L1 cache, 0 if not
    - cudaDevAttrLocalL1CacheSupported: return 1 if device supports caching locals in L1 cache, 0 if not
    - cudaDevAttrL2CacheSize: return Size of L2 cache in bytes. 0 if the device doesn't have L2 cache

See: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html

CUDA Terminology 3: Kernel

Kernel: function (void only) that launched, usually (*) by the host and executed asynchronously (non blocking the host) on the device

- Specified by __global__
- example:

```
template<typename T>
 __global__ void Add_kernel(int numElements, T* dst, const T* src)
{
    const int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < numElements)
        dst[idx] += src[idx];
}
```

- Launch from the host by: Add_kernel<<< gridSize, blockSize, SharedMemSize=0, Stream=0>>> (inside <<< >>> is the execution configuration)
- The kernel code run on all the kernel threads concurrently.

  Each thread has built-in variables as:
  - blockDim : (`uint3`) variable contains the size/dimension of the block
  - blockIdx : (`uint3`) variable contains the index of the current block instance within the grid
  - threadIdx: (`uint3`) variable contains the index of the current thread instance within the block

See C/C++ Support: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-cplusplus-language-support

(*): to launch kernel from another kernel see dynamic parallelism (CC 3.5 and higher) in:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism

CUDA Terminology 4: Device Function

Device Function: function that run on the device

- Specified by __device__
- Can be called from a kernel or other Device function
- Cannot be called from the host
- example:

```
template<typename T>
__device__ T MinMax(T a, T b, bool min_or_max)
{
    if (min_or_max)
        return (a < b) ? a : b ;
     else
        return (a > b) ? a : b ;
}
```

- can return value

See:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#return-type-deduction

CUDA Terminology 5: Grid/Block/Warp/Thread

- **Thread**
  - CUDA thread are extremely lightweight compared to CPU threads
  - No context switch (resources stay allocated to each thread until it completes its execution)
- **Warp**: a group of (32) consecutive threads which execute Single Instruction on Multiple-Data (SIMD) concurrently on a single SM. It is called also SIMT (Single Instruction Multiple thread).

  Each warp has its own instruction address counter and register state, so can branch and execute independently.
- **Block:** Group of (1/2/3 dimensional) threads, divided to Warps. Threads ID's 0:31 assigned to the 1st Warp. Threads ID's 32:63 assigned to the 2nd Warp and so on. Executed by a single SM.
- **Grid:** Group of (1/2/3 dimensional) thread blocks. Can be executed by all the device SM's.

So:

Kernel is executed by a **Grid**.

Grid is executed by **Blocks**.

Blocks are executed by **Warps**.

Warps are executed by **Threads**.

See:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture

CUDA Asynchronous commands

The following CUDA commands are non-blocking the host:

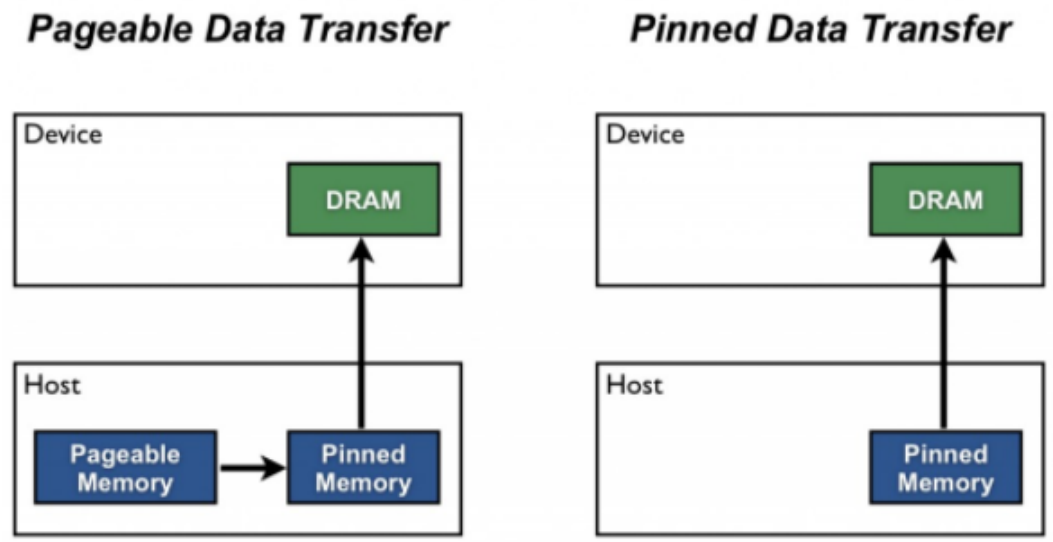
- Kernel launch
- Memory copy from/to the same device
- Memory copy/set with Async suffix (as **cudaMemcpyAsync**, **cudaMemsetAsync**, **cudaMemcpyFromSymbolAsync**, …) if the host memory is pinned (page locked)

See:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#asynchronous-concurrent-execution

Pinned Host Memory

Host (CPU) data allocations are pageable by default. The GPU cannot access data directly from pageable host memory, so when a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or "pinned", host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory, as illustrated below.



As you can see in the figure, pinned memory is used as a staging area for transfers from the device to the host. We can avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in pinned memory. Allocate pinned host memory in CUDA C/C++ using cudaMallocHost() or cudaHostAlloc(), and deallocate it with cudaFreeHost(). It is possible for pinned memory allocation to fail, so you should always check for errors. The following code excerpt demonstrates allocation of pinned memory with error checking.

```
cudaError_t status = cudaMallocHost((void**)&h_aPinned, bytes);
if (status != cudaSuccess)
  printf("Error allocating pinned host memory\n");
```

Warning:
Pinned memory should not be overused. Excessive use can reduce overall system performance!
See:
https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#pinned-memory

From Mark Harris Blog:
https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

CUDA Terminology 6: CUDA Stream

CUDA Stream: a sequence of kernels or CUDA commands (possibly issued by different host threads) that execute in order

- Streams can be used for better device utilization.

  For instance: by running concurrently
  - Kernel
  - cudaMemcpyAsync(DeviceToHost), when the host memory is paged locked
  - cudaMemcpyAsync(HostToDevice), when the host memory is paged locked

  While each of these operations is performed in a different stream (and not the **Default Stream**)
- cudaMemcpyAsync can run concurrently only in different directions
- kernels from different streams can run concurrently only if there are enough resources on the GPU, otherwise will run interleaved.

See:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams

- To enable per-thread default streams in CUDA 7 and later, you can either:
  - Compile with the nvcc command-line option: **--default-stream per-thread**
  - #define **CUDA_API_PER_THREAD_DEFAULT_STREAM** before including CUDA headers

CUDA Terminology 7: Default Stream

Default Stream: (or NULL Stream) a single and unique stream that can be used by all the host threads.

- Can be used when concurrency is not required
- Will be used by default, if no stream is specified
- Causes implicit synchronization

  i.e. When a command issued to the **Default Stream**, it will not begin until all previously issued commands in **ANY STREAM** have completed, and all the commands in **ALL STREAMS** issued after will not begin until the command in the default stream has been completed.
- In other words: two commands from any stream cannot run concurrently if the host thread issues any command to the **Default Stream** between them

See:
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams
https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/ -

Streams Synchronization

- **cudaDeviceSynchronize**()

  blocks the host thread until all preceded commands in all the streams have completed

- **cudaStreamSynchronize**(stream)

  blocks the host thread until all preceded commands in a specified stream have completed

- **cudaStreamWaitEvent**(stream, event, flags)

  All the commands issued to the stream after this call will wait until the event occurs.

- **cudaStreamQuery**(event)

  Query if all preceding commands in a stream have completed

- **cudaEventSynchronize**(event)

  blocks the host thread until a specified event has been recorded

See:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-explicit-synchronization

1. CUDA Overview & Basic Terminology
2. CUDA Execution Model
3. CUDA Memory Model
4. Starting CUDA Optimization

CUDA Terminology 8: Latency

- **Latency**: Num clocks cycles required for a Warp to complete its current instruction

  While an arithmetic operation latency is ~10-20, load/store of off-chip memory operation latency is ~100-400, depending on the device architecture (compute capability)

  For instance: according to Nvidia documentation, the typical latency for CC 3.x is about 11, which means 44 warps are needed to hide the latency

- Warp instruction

Instruction	Performed by
32-bit int/float arithmetic or logic condition	ALU's
64-bit (double precision) arithmetic	DP units
Load/Save from/to global/shared/local memory	LSDT units
Trigonometric function (sin, cos,..), log, exp, sqrt,..	SPU units

See instructions: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-ref

CUDA Terminology 9: Occupancy

- **Occupancy**: The ratio of average number of active warps on SM's to the maximum number of active warps supported by the SM
- **Theoretical Occupancy**

  The max occupancy available based on the kernel execution configuration, the resources required by the kernel and the CUDA device capabilities.
- **Achieved Occupancy**

  The average of occupancy measured during the kernel execution
- **Theoretical Occupancy (for specified CC) is controlled by three factors:**
  - Block size
  - Shared memory size (per block)
  - Registers size (per thread)
- **Higher occupancy does not always mean higher performance**

See:

https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm

Compute Theoretical Occupancy: Example

Compute Capability 5.0	
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Registers per Multiprocessor	65536
Shared Memory per Multiprocessor (bytes)	65536

- Suppose **Block Size** was selected to be: 128 (= 4 warps)

  Max num of blocks limited by block size = 64/4 = **16**

- **Shared Memory** per block = 5000 bytes

  Max num of blocks limited by shared Memory = 65536/5000 = 13.1 ➔ **13**

- **Registers** Per Thread= 48; per block= 128*48 = 6144;
  Max num of blocks limited by resisters = 65536/6144 = 10.67 ➔ **10**
  This is the limiting parameter!
  So the Theoretical Occupancy is **10** * 4 / 64 = 40 / 64 = **0.625**

- Question: What will be the Theoretical Occupancy if we will need 10000 bytes shared memory?

CUDA Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	5.0	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	

2.) Enter your resource usage:		
Threads Per Block	128	(Help)
Registers Per Thread	48	
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

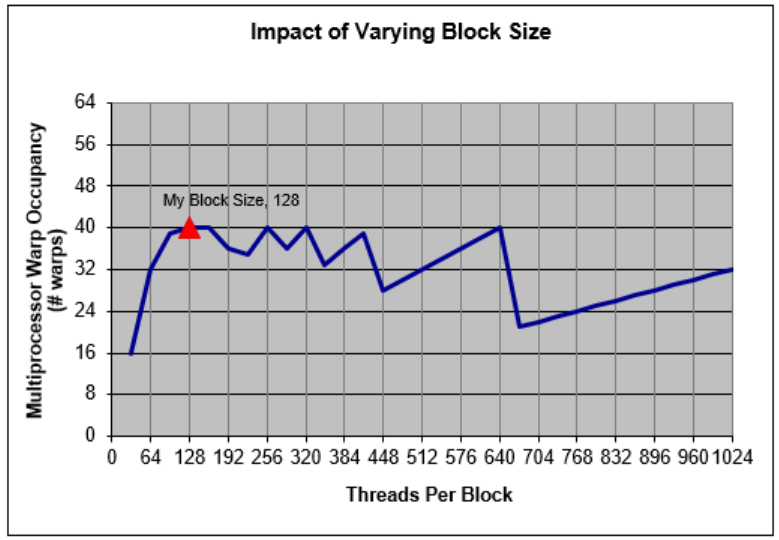
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	**1280**	(Help)
Active Warps per Multiprocessor	**40**	
Active Thread Blocks per Multiprocessor	**10**	
Occupancy of each Multiprocessor	**63%**	

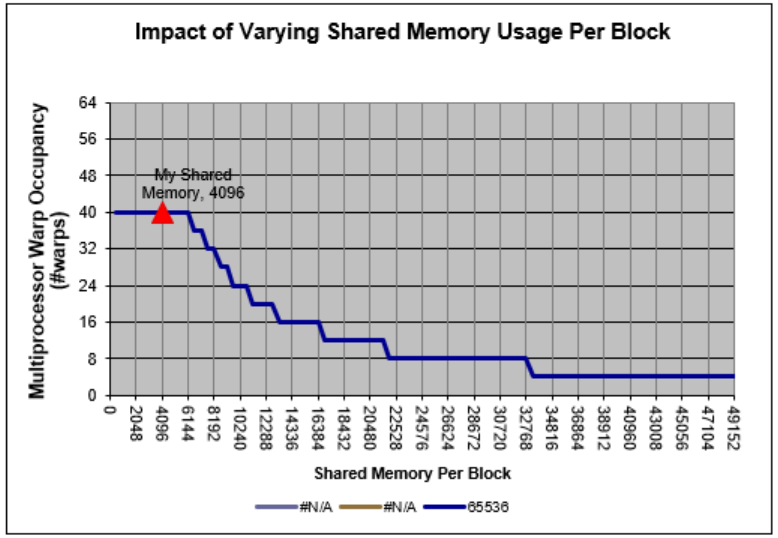
Physical Limits for GPU Compute Capability:	5.0
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4

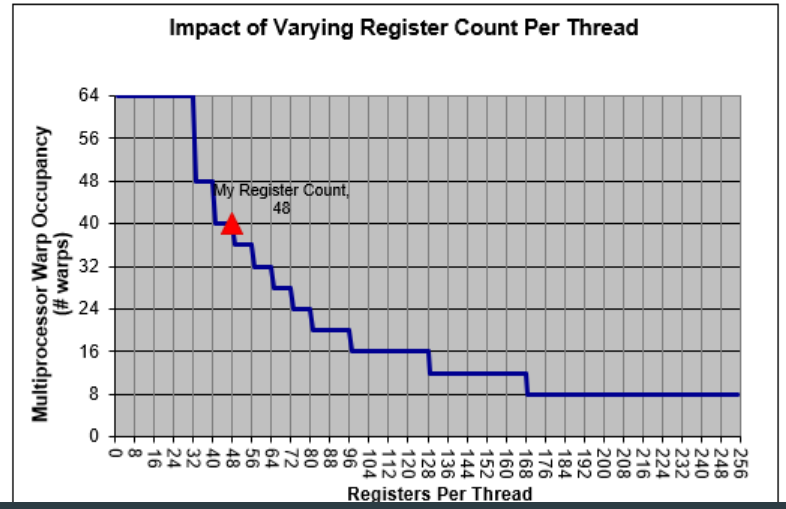
Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	4	64	16
Registers	(Warp limit per SM due to per-warp reg count)	4	40	10
Shared Memory (Bytes)		4096	49152	16

Note: SM is an abbreviation for (Streaming) Multiprocessor

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







https://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Why Achieved Occupancy < Theoretical?

- **Unbalanced workload within blocks (Significant variation in warps execution time)**

  Fewer active warps at the kernel end - "tail effect"

- **Unbalanced workload across blocks (Significant variation in blocks execution time)**

  The “tail effect” does not occur inside every block, but only at the end of the kernel.

- **Too few blocks launched**

See:
https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm

Warp Scheduler work – 1 (*)

- When a kernel is launched

  Block size and the **Grid size** : are defined by execution configuration

  Shared memory size per block : is defined by kernel code

  Registers memory size per thread : is defined by kernel code

- The Compute Work Distributor will assign a blocks to a SM's as long as the SM has sufficient resources for each block as: Shared memory, registers and warps. Each Block assigned to one SM and executed only on it.
- Unassigned blocks will wait until one block terminate execution and exit

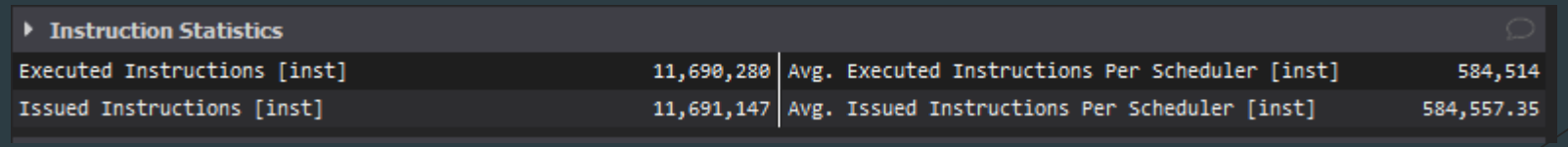
See:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture

(*) The Volta architecture introduces *Independent Thread Scheduling* among threads in a warp

Warp Scheduler work - 2

- When block terminate execution, the resource manager releases it resources (shared memory)
- When block is assigned to SM, **all** its Warps become active, i.e. assigned registers memory and added to a Warp Scheduler.
- At every clock cycle, each warp scheduler selects one warp from the eligible warps (Selected Warp) and issues to it the next one or two instructions.
- A warp scheduler might issue an instruction multiple times to complete all 32 warp threads execution. The two primary reasons for this difference between **Instructions Issued** and **Instructions Executed** are:
  - address divergence and bank conflicts on memory operations
  - assembly instructions that can only be issued for a half-warp per cycle and thus need to be issued twice. (Double floating-point instructions for example)



Instruction Statistics			
Executed Instructions [inst]	11,690,280	Avg. Executed Instructions Per Scheduler [inst]	584,514
Issued Instructions [inst]	11,691,147	Avg. Issued Instructions Per Scheduler [inst]	584,557.35

See:

https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/instructionstatistics.htm

Warp Scheduler work – 3

- At every instruction issue time, each Warp Scheduler issues one instruction for one of its assigned warps that is ready to execute (if any) by a prioritized scheduling policy as: (see the reference below)
  - Loosely- Round-Robin (LRR)
  - Greedy-then-Oldest (GTO)
  - prefetch-aware
  - memory-aware
- The most common reason a warp is not ready to execute its next instruction is that the instruction's input operands are not available yet. Other reason in waiting for synchronization point.
- SM is fully utilized when all warp schedulers always have some instruction to issue for some warp at every clock cycle - The latency is fully hidden.

See: Barrier-Aware Warp Scheduling for Throughput Processors from may, 2016

CUDA Terminology 10: Warps in Warp Scheduler

- **Eligible Warp:** a warp that is ready for execution. i.e.:
  - The next instruction was fetched and all its arguments are ready
  - Resources are ready: fp32 cores/ fp64 cores/ ldst unit /special function units
- **Stalled Warp:** a warp that is not ready for execution the next instruction
  - Pipeline Busy — The compute resources required by the instruction are not yet available
  - Instruction Fetch — The next assembly instruction has not yet been fetched
  - Memory Throttle — A large number of pending memory operations prevent further forward progress
  - Memory Dependency — LD/ST units are not available or fully utilized, or too many requests already issued
  - Execution Dependency — An input required by the instruction is not yet available
  - Synchronization — The warp is blocked at a _syncthreads()

  Notice that even the previous instruction didn't completed yet, it doesn't stall the warp.

  The warp will be stalled only when attempting to access an argument not ready yet.

See: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/issueefficiency.htm#IssueStallReasons

CUDA Terminology 11: Warps in Warp Scheduler

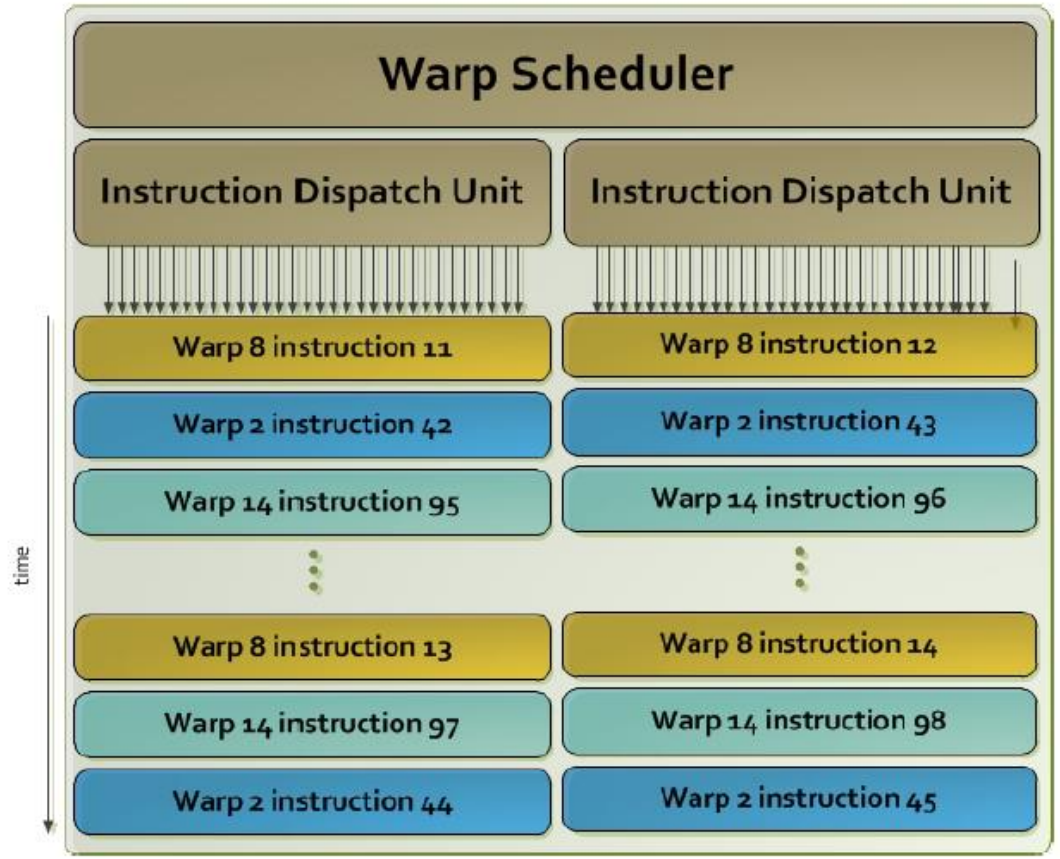
- **Active Warp:** A warp is defined as active when it is assigned to a warp scheduler and its threads start executing until all its threads end executing

  Num Active Warps = Num Eligible Warps + Num Stalled Warps

- **Selected Warp:** The warp selected to execute the next instruction in the current warp scheduler. The number of selected warps at any cycle in SM is <= the number of warp schedulers in the SM.

- **Warp Divergence:** Warp threads may diverge under conditional branch, so execute different paths. In this case the warp execute each branch path while disabling the threads that are not taking part in that path.

See:
https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/issueefficiency.htm



Each Kepler SMX contains 4 Warp Schedulers, each with dual Instruction Dispatch Units. A single Warp Scheduler Unit is shown above.

We also looked for opportunities to optimize the power in the SMX warp scheduler logic. For example, both Kepler and Fermi schedulers contain similar hardware units to handle the scheduling function, including:

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

Pascal GP100 architecture (page 10) from:
https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Full Utilization = Full Hiding Latency

CUDA Device is an Hiding Latency Machine

See:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#multiprocessor-level

https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.pdf

1. CUDA Overview & Basic Terminology
2. CUDA Execution Model
3. CUDA Memory Model
4. Starting CUDA Optimization

CUDA Device Memory Types

Table 1. Salient Features of Device Memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached in L1 and L2 by default on devices of compute capability 2.x; cached only in L2 by default on devices of higher compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

†† Cached in L1 and L2 by default on devices of compute capability 2.x and 3.x; devices of compute capability 5.x cache locals only in L2.

See https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#device-memory-spaces

CUDA Terminology 12: Global Memory

- **Global Memory**: Is a memory that resides in DRAM (Device Random Access Memory).
- has a high latency access - 100s of clock cycles. Much slower than shared memory.
- Global memory and is accessed via 32/64/128-byte memory transactions.
- When a warp executes an instruction that accesses global memory (LD or ST instructions), it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread.
  If word size is 1/2/4 bytes the memory transaction will be 32/64/128 bytes correspondingly.
- A L1 cache line is 128 bytes and maps to a 128 byte aligned segment in device memory.
  Memory accesses cached by L1 and L2 are serviced with 128-byte memory transactions
- A L2 cache line is 32 bytes and maps to a 32 byte aligned segment in device memory.
  Memory accesses cached by L2 only are serviced with 32-byte memory transactions.
- Global memory accesses can be configured at compile time to be cached in L1 and L2 caches (**Cached Loads**) or in L2 only (**Uncached Loads**).

See:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses

https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/sourcelevel/memorytransactions.htm

https://docs.nvidia.com/gameworks/index.html#developertools/desktop/nsight/analysis/report/cudaexperiments/kernellevel/memorystatisticsglobal.htm

CUDA Terminology 13: Coalesced Access

- **Global Memory Coalesced Access**

  Occurs when all the threads in a warp execute a load instruction such that they access aligned and consecutive memory locations.

- When the warp access to Global Memory is coalesced, the number of memory transaction is minimized

See:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses

CUDA Terminology 14: Constant Memory

- The constant memory space resides in device memory and is cached in the constant cache.
- Constant memory can be access from all kernels as read-only data
- Must be set from the host prior to launch kernel
- Constant memory size is 64KB for all Compute Capabilities (1.x to 7.x)

  while the cache working set is 8KB (4KB only for CC 6.0)
- Since the constant memory size is much smaller than the global memory and it is cached, the average latency is much smaller than global memory access.

See:

https://docs.nvidia.com/cuda/cda-c-programming-guide/index.html#device-memory-accesses

CUDA Terminology 15: Local Memory & Register Spilling

- Local Memory scope is local to the thread, as registers
- The local memory space resides in device memory, so have the same high latency and low bandwidth as global memory
- Automatic variables that the compiler is likely to place in local memory are:
  - Arrays not indexed with constant quantities
  - Large structures or arrays
  - Any variable if the kernel uses more registers than available (**Register Spilling**).
- Local memory is organized such that consecutive 32-bit words are accessed by consecutive thread IDs, so accesses are therefore fully coalesced as long as all threads in a warp access the same relative address

See:

https://docs.nvidia.com/cuda/cda-c-programming-guide/index.html#device-memory-accesses

CUDA Terminology 16: Shared Memory and Bank Conflict

Bank	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
row0: 32-bit address 0 - 31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
row1: 32-bit address 32 - 63	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
row2: 32-bit address 64 - 95	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
row3: 32-bit address 96 - 128	96	97	98	99																												

- Shared memory is divided into equally-sized memory modules, called banks.
  There are 32 memory banks in all advanced devices (Compute capability 2.x to 6.x)
- Successive 32-bit words are assigned to successive banks. Each bank can service one address per clock cycle.
- Memory read/write made of n addresses in n distinct banks can be serviced simultaneously
- If two different addresses of shared memory request fall in the same memory bank,
  there is a bank conflict and the access will be serialized.
- Shared memory is ~ as fast as registers as long as there are no bank conflicts

Shared Memory Bank Conflict: Example

- A [32 x 32] image square tile will be defined as a static shared memory
  __shared__ float tile[32][32];
- We shall mark each word in the shared memory with the bank index it get a service from
- If consecutive threads will access consecutive words in a shared memory row, it will be served concurrently
- But what will happened if consecutive threads will access consecutive words in a shared memory column?

Matrix Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
row0: bank number For each word	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
row1: bank number For each word	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
row2: bank number For each word	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
row3: bank number For each word	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Next rows. . .																																

Handling Shared Memory Bank Conflict: 1 Padding

- A [32 x 32] image tile will be defined as a block static shared memory

  __shared__ float tile[32][32 + 1];

Tile Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
row0: bank number For each word	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0
row1: bank number For each word	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1
row2: bank number For each word	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2
row3: bank number For each word	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3

Using Dynamic Shared Memory

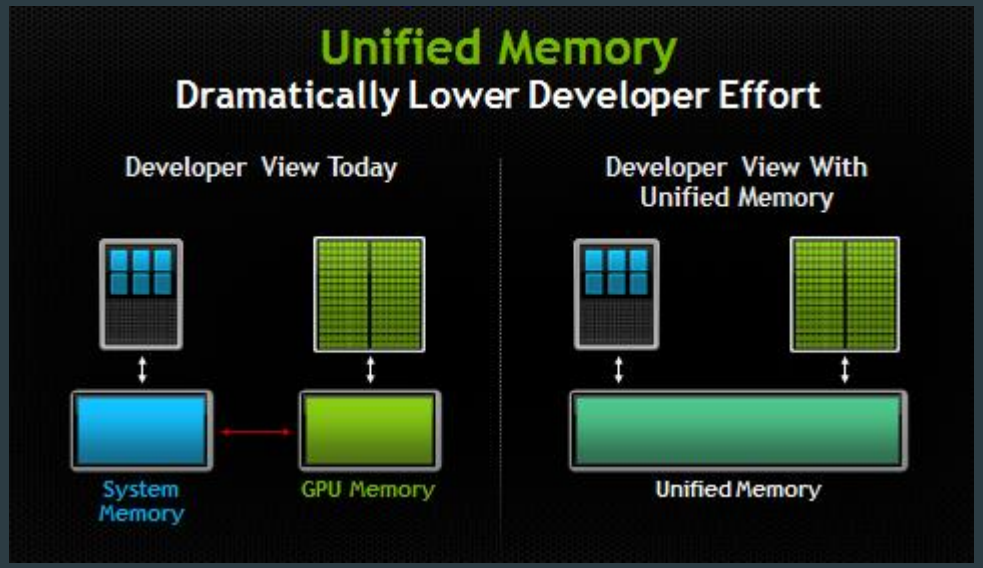
```
template<typename T>
__global__ void Somekernel(. . .)
{
   extern __shared__ T sharedMemory[];
   // 1. Copy from Global memory to Shared Memory
   . . . . . .
   __syncthreads();
   // 2. Processing, using the Shared Memory
   . . . . . .
   // 3. Copy from the Shared Memory to the Global Memory
   . . . . . .
}

Somekernel<float><<< gridSize, blockSize, sharedMemSize, stream>>>(. . .);
```

CUDA Terminology 17: Uniform Memory

- Managed memory that is shared between the CPU and GPU, Introduced in CUDA 6.0 (2013)
- Automatically migrates allocated data in between host and device,

  So, cudaMemcpy (DeviceToHost and HostToDevice) are not necessary
- Biggest advantage: Code Simplification



See:

https://devblogs.nvidia.com/unified-memory-in-cuda-6/

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd

1. CUDA Overview & Basic Terminology
2. CUDA Execution Model
3. CUDA Memory Model
4. Starting CUDA Optimization

Staring CUDA Optimization (1)

- Prefer use of Shared Memory over Global Memory wherever applicable
- Try to avoid Global memory uncoalesced access
- Try to avoid Shared Memory bank conflicts
- Use Constant Memory whenever global memory used as read-only inside kernel
- Avoid unbalanced workload (between warps, between blocks)
- Try to schedule as long as possible kernels queue w/o host sync in between
- If the device is shared between applications, don't use the default stream
- Try to achieve the highest occupancy available as long as you don't hit any resource bound limit
- For frequent H2D and D2H cudaMemcpy, consider use of Host pinned memory

Staring CUDA Optimization (2)

- Minimize using of barriers inside device code.
- Minimize or avoid using cudaDeviceSynchronize (heavy hammer)
- Consider use of #pragma unroll before loops with size known at compile time
- Declare read-only (and unaliased) parameters as const (and __restrict__)
- Block Size of 128 or 256 is usually the best choice. Avoid too big block size
- Analyze your code with Nvidia Tools
  - nvprof, nvvp: Command Line Profiler and Visual Profiler

    (Will be Deprecated in future release)
  - Nsight: The most powerful tool for profiling and debugging CUDA

    Can be used for various profiling issues as:
    - Finding the most common reasons for stalled warps (if there are too many)
    - Finding the level of resources utilization
    - . . . . See: https://www.nvidia.com/object/nsight.html

NVIDIA Nsight

Activity Type — Profile CUDA Application with Nsight Compute

- ○ **Trace Application**
  Collects events from the target application. The analysis session and data collection are stopped when the launched application exits.
- ○ **Trace Process Tree**
  Collects events from the target application and all native child processes of the target application. The analysis session and data collection are not stopped when the launched application exits. The session and data collection must be stopped manually.
- ○ **Profile CUDA Application**
  Collects counters, statistics and derived values for given CUDA kernel launches.
- ○ **Profile CUDA Process Tree**
  Collects counters, statistics and derived values for given CUDA kernel launches from the target application and all native child processes of the target application. The analysis session and data collection are not stopped when the launched application exits. The session and data collection must be stopped manually.
- ◉ **Profile CUDA Application with Nsight Compute**
  Collects counters, statistics and derived values for given CUDA kernel launches using the command line profiler.

Nsight Compute Profiler Settings

Kernels to Profile:

☐ After skipping *No* kernels, profile *All* kernels.

☐ Apply Rules

Select All Clear All

- ☑ Compute Workload Analysis
- ☑ Instruction Statistics
- ☑ Launch Statistics
- ☑ Memory Workload Analysis
- ☑ Occupancy
- ☑ Scheduler Statistics
- ☑ Source Counters
- ☑ GPU Speed Of Light
- ☑ Warp State Statistics

Resulting Commands:

Page: Details Launch: 0 - 122 - NaiveTranspose Add Baseline Apply Rules Copy as Image

Current 122 - NaiveTranspose (6000128, 1, 1) Time: 2.93 msecond Cycles: 2,702,311 Regs: 15 GPU: GeForce GTX 950A SM Frequency: 922.02 Mhz CC: 5.0

Recommendations

Bottleneck High-level bottleneck detection — Apply

Compute Workload Analysis

Executed Ipc Elapsed [inst/cycle]	1.05	SM Busy [%]	24.63
Executed Ipc Active [inst/cycle]	1.06	Issue Slots Busy [%]	17.61
Issued Ipc Active [inst/cycle]	1.06		

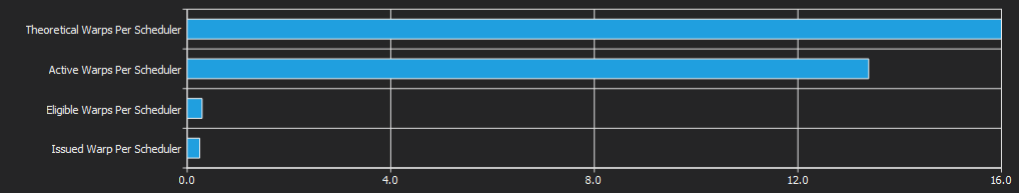
Memory Workload Analysis

Memory Throughput [Gbyte/second]	3.95	Mem Busy [%]	31.33
L1 Hit Rate [%]	0	Max Bandwidth [%]	30.37
L2 Hit Rate [%]	96.94	Mem Pipes Busy [%]	2.78

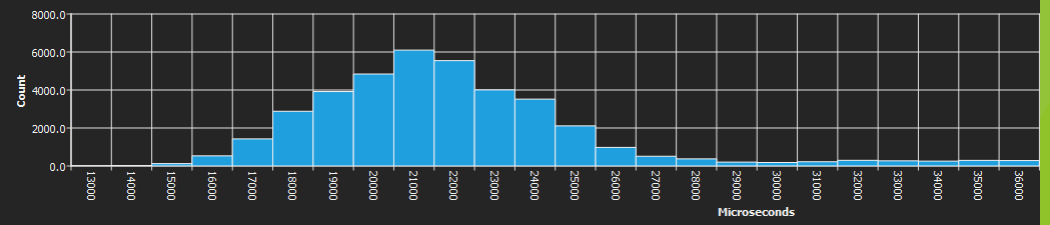
Scheduler Statistics

Active Warps Per Scheduler [warp/cycle]	13.39	Instructions Per Active Issue Slot [inst/issue]	1.07
Eligible Warps Per Scheduler [warp/cycle]	0.29	No Eligible [%]	75.31
Issued Warp Per Scheduler [issue/cycle]	0.25	One or More Eligible [%]	24.69

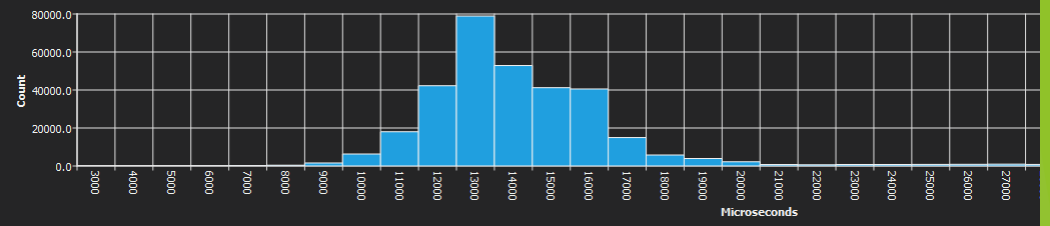
Warps Per Scheduler



Block Durations



Warp Durations



Recommendations

Launch Configuration Kernel launch configuration analysis

Occupancy

Theoretical Occupancy [%]	100	Block Limit Registers [register]	
Theoretical Active Warps per SM [warp/cycle]	64	Block Limit Local Mem [byte]	⚠
Achieved Occupancy [%]	84.79	Block Limit Warps [warp]	
Achieved Active Warps Per SM [warp/cycle]	54.26	Block Limit SM [block]	

Examine the assembly code

Use cuobjdump utility (in every CUDA toolkit)

https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#cuobjdump

- See what's going on behind the scenes
- Verify whether Local Memory is used within our kernel code
- **Get PTX code: cuobjdump myBin.exe[.dll] -ptx > myBin.ptx**
  - Convert host binary file to a virtual machine (or pseudo-assembly) language for Parallel Thread Execution (PTX)
  - Provide a machine-independent ISA (Instruction Set architecture) for multiple GPU generations as:
    - ld.global / st.global : are global memory load/store commands
    - ld.local / st.local : are local memory load/store commands
    - ld.shared/ st.shared : are shared memory load/store commands
    - ld.const : are constant memory load commands
    - ld.param : are load kernel or function parameters

  See:

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

  https://docs.nvidia.com/cuda/pdf/ptx_isa_6.0.pdf
- **Get Machine assembly code: cuobjdump myBin.exe[.dll] -sass**

  see https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-ref