SCALABLE COMPUTING LABORATORY

VANDERBILT UNIVERSITY

SC3260 / SC5260

GPU Performance

Lecture by: Ana Gainaru

Today

Focus on performance

- Performance optimizations for GPU
- Performance difference between CPU and GPU
- Submitting jobs on ACCRE
- Post-running analysis (Homework 2)

Performance Considerations

Main bottlenecks

- Global memory access
- CPU-GPU data transfers

- Memory access
  - Latency hiding
  - Memory coalescing
  - Data reuse
  - Shared memory usage
- SIMD Utilization
- Atomic operations
- Data transfers between CPU and GPU
  - Overlap of communication and computation

Multiprocessor occupancy

- Defined as number of warps running concurrently on multiprocessor divided by max warps that can run concurrently
  - Max warps differs across architectures
  - On C1060: max of 32 concurrent warps
  - On Kepler K20: max of 64 concurrent warps

Multiprocessor occupancy

- Defined as number of warps running concurrently on multiprocessor divided by max warps that can run concurrently
  - Max warps differs across architectures
  - On C1060: max of 32 concurrent warps
  - On Kepler K20: max of 64 concurrent warps

Can be limited by register and shared memory usage

- Registers and shared memory are shared among all active warps on multiprocessor
- Higher register/shared memory usage in each thread limits number of simultaneous warps

Latency hiding

Higher occupancy is often a goal in GPU optimization

- Greater occupancy can hide instruction latency
  - Read-after-write register latency
  - Latency in reading data from global memory
  - While threads in one warp waiting for data, another warp can run on multiprocessor

Recommendation from NVIDIA

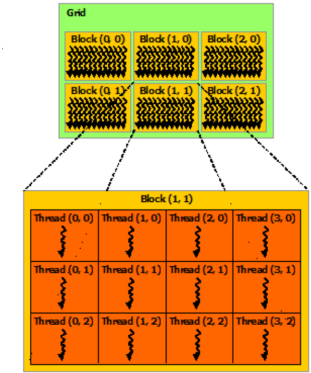
Run at least 192 threads (6 warps) per GPU
Threads do not have to belong to the same thread block

Multiprocessor Occupancy Factors

Maximizing occupancy does not always result in best performance

Factors

- Thread block dimensions
- Register usage per thread
- Shared memory usage per thread block
- Target GPU architecture



VANDERBILT UNIVERSITY

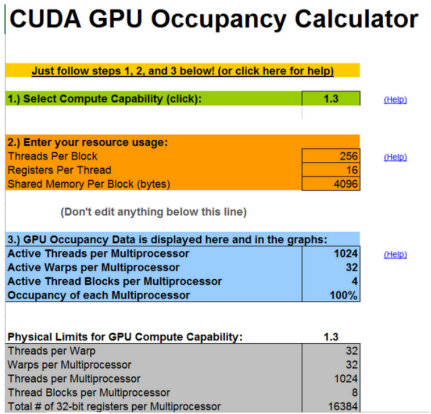
CUDA Occupancy Calculator

- Provided by NVIDIA to compute occupancy of CUDA kernel
- User enters GPU compute capability and resource usage
- Occupancy calculator computes occupancy
- Shows impact of...
  - Adjusting thread block size
  - Adjusting register usage
  - Adjusting shared memory usage

- **Can be used as a tool to tweak CUDA program to improve multiprocessor occupancy**

CUDA Occupancy Calculator

Available at https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html



CUDA GPU Occupancy Calculator

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

1.) Select Compute Capability (click):	1.3	(Help)

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	16	
Shared Memory Per Block (bytes)	4096	

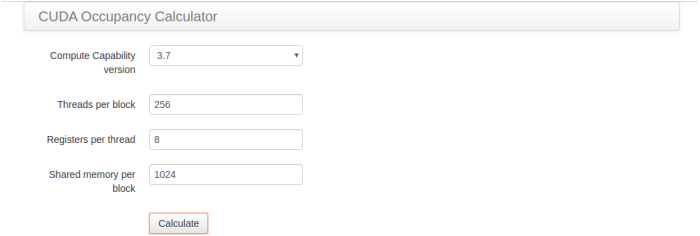
(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1024	(Help)
Active Warps per Multiprocessor	32	
Active Thread Blocks per Multiprocessor	4	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	1.3
Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384

CUDA Occupancy Calculator

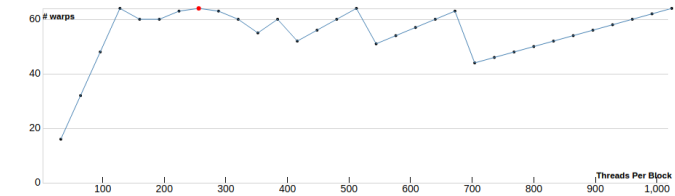
Or here https://xmartlabs.github.io/cuda-calculator/



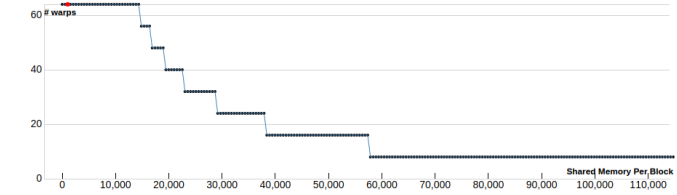
VANDERBILT UNIVERSITY

CUDA Occupancy Calculator

Impact of Varying Block Size



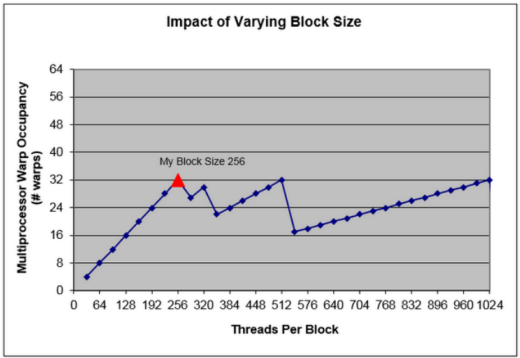
Impact of Varying Shared Memory Usage Per Block



VANDERBILT UNIVERSITY

CUDA Occupancy Calculator

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.



CUDA Occupancy Considerations

All things equal, same program with higher occupancy should run faster

However, may be necessary to sacrifice register / shared memory usage to increase occupancy

- May increase necessary memory transfers from global memory
- **May slow program more than reduced occupancy**

Thread Block Dimensions

- CUDA threads grouped together in thread block structure
  - Run on same multiprocessor
  - Have access to common pool of fast shared memory
  - Can synchronize between threads in same thread block
- On a large number of GPUs, maximum of 8 active thread blocks / multiprocessor
- Max thread block size is 512

Optimizing Thread Block Dims

On a GPU

- Up to 1024 threads can execute concurrently
- Up to 8 thread blocks can be active

To allow full occupancy, thread block size must be at least 128

Optimizing Thread Block Dims

On a GPU

- Up to 1024 threads can execute concurrently
- Up to 8 thread blocks can be active

To allow full occupancy, thread block size must be at least 128

Unless shared memory requirement forces lower block size

- Use multiple of warp size (32)
- Different thread block dimensions work best for different programs
- Experiment and see what works best
  - Common thread block sizes: 128, 192, 256, 384, 512
  - If 2D thread block, common dimensions are 32X4, 32X6, 32X8, 32X12, 32X18

Global memory load/store pattern

- Global memory latency is over 400 clock cycles
- Minimize accesses as much as possible
- Best to arrange accesses in **coalesced manner**

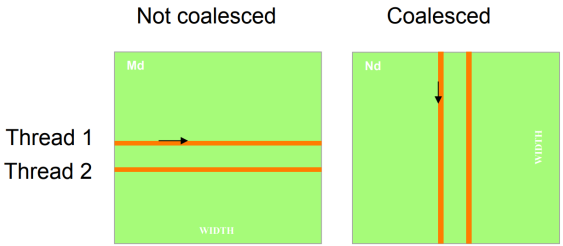
Global memory load/store pattern

- Global memory latency is over 400 clock cycles
- Minimize accesses as much as possible
- Best to arrange accesses in **coalesced manner**

- Global memory accesses are per half-warp (16 threads)
- If data perfectly aligned and thread memory accesses contiguous, data for threads in half-warp retrieved in one chunk
- If data accesses not in same array chunk, need additional memory accesses
  - Anywhere between 1-16 memory accesses necessary to retrieve data from global memory

Memory Coalescing

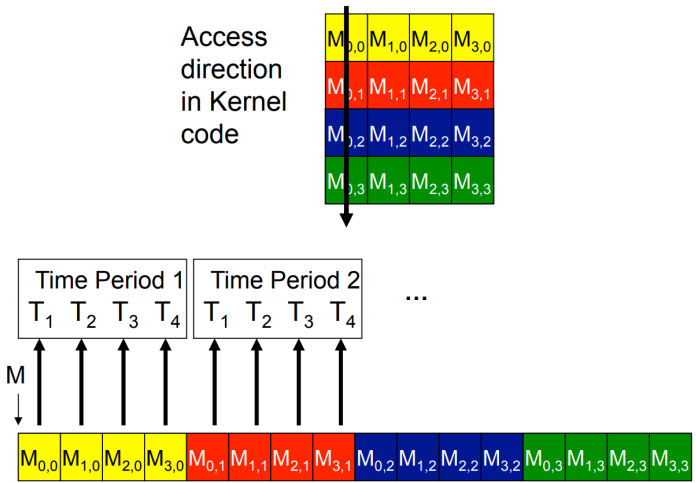
When accessing global memory, **peak bandwidth utilization occurs when all threads in a warp access the same cache line**



Source: Hwu & Kirk

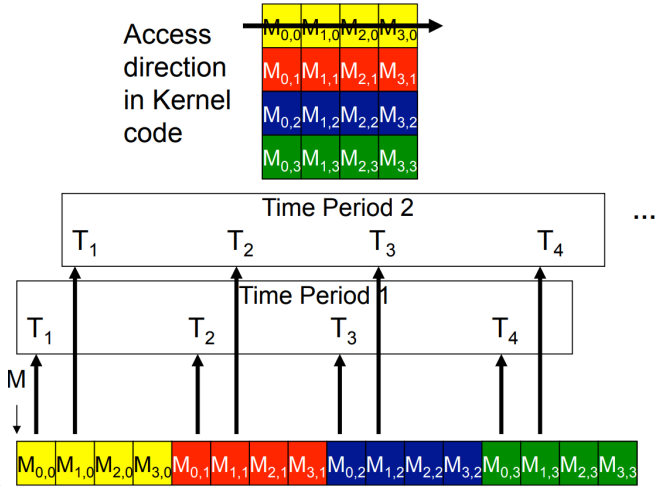
VANDERBILT UNIVERSITY

Coalesced accesses



Source: Hwu & Kirk

Uncoalesced accesses



Source: Hwu & Kirk

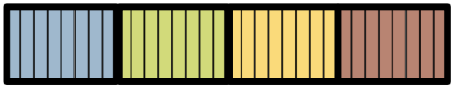
VANDERBILT UNIVERSITY

Memory Coalescing

AoS vs. SoA

Structure of Arrays (SoA)

```
struct foo{
 float a[8];
 float b[8];
 float c[8];
 int d[8];
} A;
```



Array of Structures (AoS)

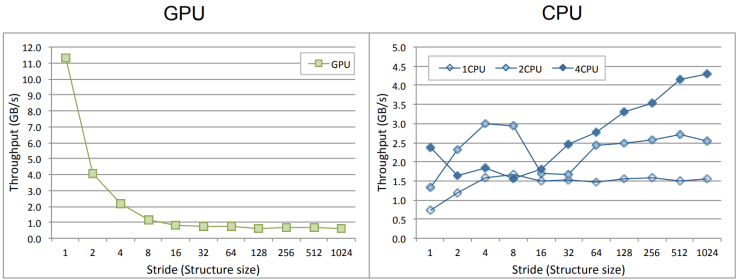
```
struct foo{
 float a;
 float b;
 float c;
 int d;
} A[8];
```

Source: https://safari.ethz.ch/architecture/fall2017

VANDERBILT UNIVERSITY

Memory Coalescing

Linear and strided accesses



AMD Kaveri A10-7850K

Source: https://safari.ethz.ch/architecture/fall2017

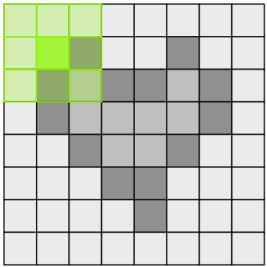
VANDERBILT UNIVERSITY

Register Usage

- Registers allow quick access to data on GPU
- Limited number of registers on each GPU multiprocessor
  - Example Tesla: 16,384 registers per multiprocessor
- Limited number of registers / thread allowed
  - Tesla: up to 127 registers allowed per thread Fermi / Kepler GK104: up to 63 registers allowed per thread
  - Kepler GK110: up to 255 registers allowed per thread
- **Greater register usage limits number of concurrent threads / multiprocessor**
  - Optimization trade-off
  - Can limit register use per thread NVCC flag `-maxrregcount=N`
    - Can cause spillover to local memory

Data Reuse

Same memory locations accessed by neighboring threads



```
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
  }
}
```

VANDERBILT UNIVERSITY

Shared Memory

Can be as fast as registers for data access

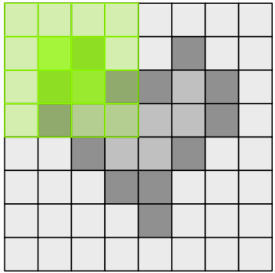
- Can be used as user-managed cache
- Shared between all threads within a thread block
  - Can be used for communication between threads
- Tesla: 16KB shared memory/multiprocessor
- **Can be limiting factor for multiprocessor occupancy**

Shared Memory

Can be as fast as registers for data access

- Can be used as user-managed cache
- Shared between all threads within a thread block
  - Can be used for communication between threads
- Tesla: 16KB shared memory/multiprocessor
- **Can be limiting factor for multiprocessor occupancy**
- Can be statically declared in CUDA kernel
  - Specified using `__shared__` keyword
  - Space for entire thread block given in declaration

Shared memory tiling



```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
...
// Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
  }
}
```

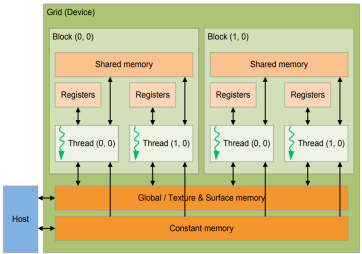
Bank conflicts are only possible within a warp

- No bank conflicts between different warps

- If strided accesses are needed, some optimization techniques can help
  - Padding
  - Hash functions

Constant Memory

- Potentially faster for readonly data
  - Best when all threads in warp are reading same data simultaneously
  - Allows for single memory read with data then broadcast to remaining threads in half-warp
  - Constant memory also cached
- Cached constant memory can be as fast as registers
- Constant memory is transferred to GPU from host using `cudaMemcpyToSymbol` command



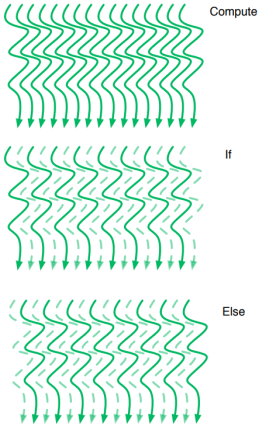

```
//declare constant memory
__constant__ float cst_ptr[size];

//copy data from host to constant memory
cudaMemcpyToSymbol(cst_ptr, host_ptr, data_size);
```

VANDERBILT UNIVERSITY

SIMD Utilization

Intra-warp divergence

```
Compute(threadIdx.x);

if (threadIdx.x % 2 == 0){
    Do_this(threadIdx.x);
}
else{
    Do_that(threadIdx.x);
}
```



VANDERBILT UNIVERSITY

Branches Within Kernels

GPU architecture dedicates most transistors to computation

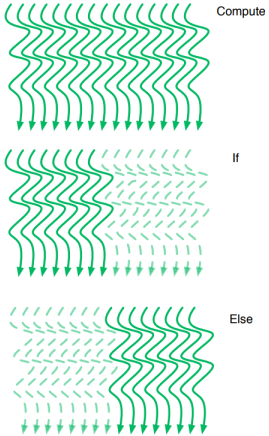
- Not much focus on branch prediction / recovery
- Tesla: If divergence in warp, all threads need to step through both branches
- Thread divergence in warp can hurt performance

- **If branching within thread, want all threads in warp to branch in same direction if possible**

SIMD Utilization

Intra-warp divergence

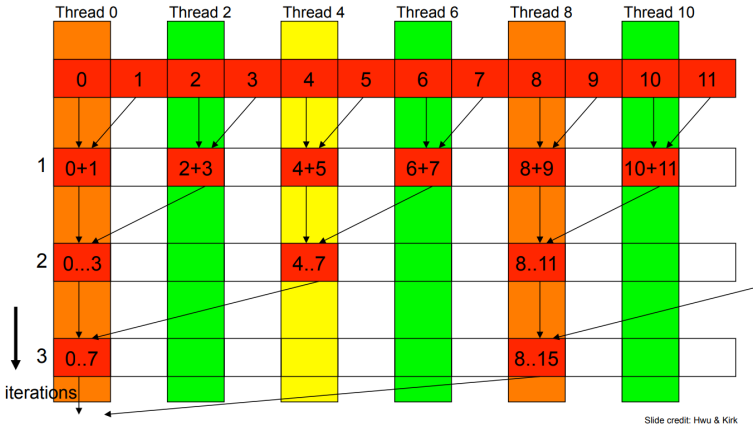
```
Compute(threadIdx.x);

if (threadIdx.x < 32){
    Do_this(threadIdx.x * 2);
}
else{
    Do_that((threadIdx.x % 32) * 2 + 1);
}
```



VANDERBILT UNIVERSITY

Vector Reduction

Naive mapping



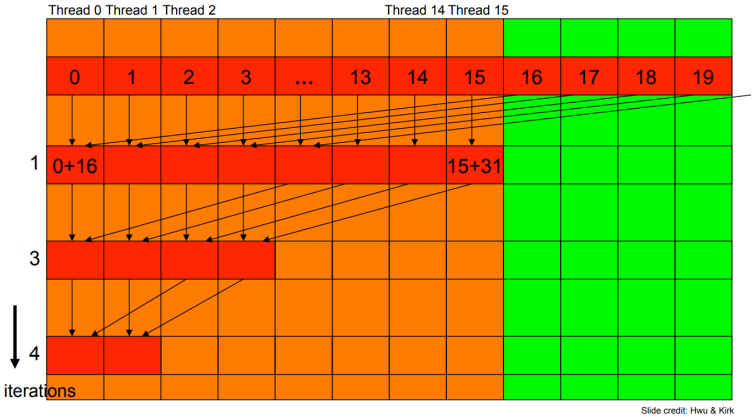
Slide credit: Hwu & Kirk

Vector Reduction

Naive mapping

```
__shared__ float partialSum[];

unsigned int t = threadIdx.x;

for (int stride = 1; stride < blockDim.x; stride *= 2) {
   __syncthreads();

   if (t % (2*stride) == 0)
         partialSum[t] += partialSum[t + stride];
}
```

Divergence-free mapping



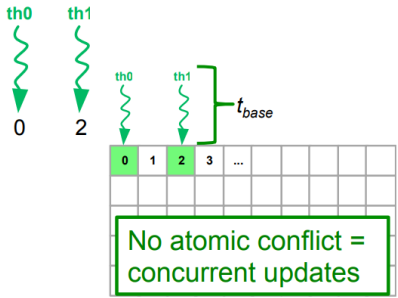
Divergence-free mapping

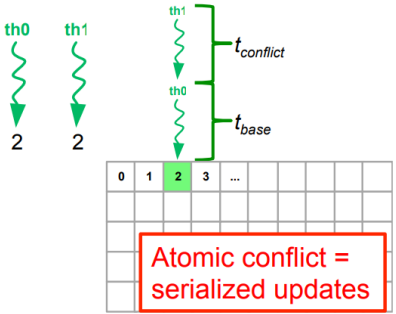
```
__shared__ float partialSum[];

unsigned int t = threadIdx.x;

for (int stride = blockDim.x; stride > 1; stride >> 1){
   __syncthreads();

   if (t < stride)
         partialSum[t] += partialSum[t + stride];
}
```

Atomic Operations

Atomic conflicts

- Intra-warp **conflict degree** from 1 to 32



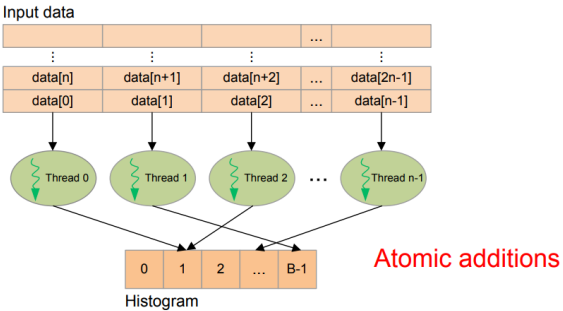


VANDERBILT UNIVERSITY

Histogram Calculation

Histograms count the number of data instances in disjoint categories (bins)

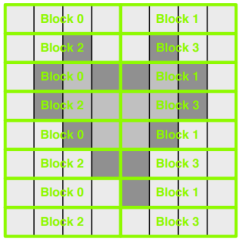
```
for (each pixel i in image I){
        Pixel = I[i]                        // Read pixel
        Pixel_new = Computation(Pixel)      // Optional computation
        Histogram[Pixel_new]++              // Vote in histogram bin
}
```

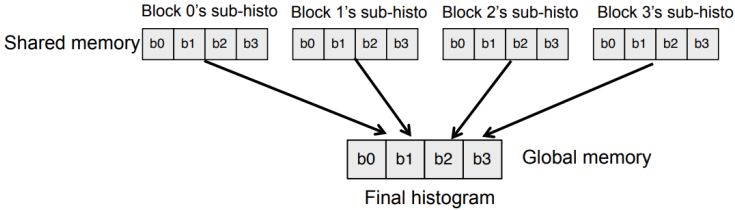


VANDERBILT UNIVERSITY

Histogram Calculation

Privatization: Per-block sub-histograms in shared memory





VANDERBILT UNIVERSITY

Conclusions

- Many factors in GPU optimizations
- Optimization may help one program and hurt another program

- Necessary to experiment among possible optimizations

Homework 2 requires you to play with all these parameters

ACCRE has 27 compute nodes equipped with Nvidia GPU cards for general-purpose GPU computing

- The nodes are divided into two partitions depending on the type of GPU available on the node

Partition	pascal	maxwell
number of nodes	15	12
GPU	4 x Nvidia Pascal	4 x Nvidia Maxwell
CPU cores	8 (Intel Xeon E5-2623 v4)	12 (Intel Xeon E5-2620 v3)
CUDA cores (per GPU)	3584	3072
host memory	128 GB	128 GB
GPU memory	12 GB	12 GB
network	56 Gbps RoCE	56 Gbps RoCE
gres	1 GPU + 2 CPUs	1 GPU + 3 CPUs

Running on ACCRE

Several versions of the Nvidia CUDA API are available on the cluster

```
module load CUDA/8.0.61
```

Running on ACCRE

Several versions of the Nvidia CUDA API are available on the cluster

```
module load CUDA/8.0.61
```

Below is an example SLURM script header to request 2 Pascal GPUs and 4 CPU cores on a single node on

```
#!/bin/bash
#SBATCH --partition=pascal
#SBATCH --gres=gpu:2
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --mem=20G
#SBATCH --time=2:00:00
#SBATCH --output=gpu_job.log
```

GPU partitions are intended for GPU jobs only, so do not run purely CPU programs

Summary

Performance optimization methods for GPU are different than CPU

- Performance considerations
  - Memory access
  - Latency hiding: #threads
  - Memory coalescing
  - Data reuse: shared memory
  - SIMD utilization
  - Atomic operations

Further Readings

Books for C CUDA

- *Programming Massively Parallel Processors*, David B. KIRK et Wen-mei W. HWU, Morgan Kaufmann, 2010
- *The CUDA handbook*, Nicholas WILT, Addison-Wesley, 2013
- https://docs.nvidia.com/cuda/

Python CUDA

- Nvidia's CUDA parallel computation API from Python
  https://mathema.tician.de/software/pycuda/
- Example of 2D FFT using PyFFT and PyCUDA
  https://wiki.tiker.net/PyCuda/Examples/2DFFT