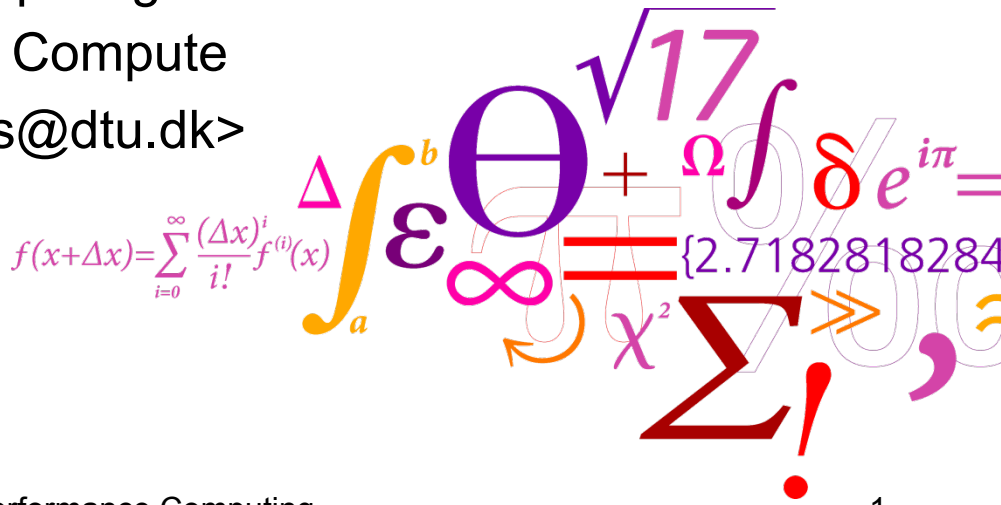
CUDA Performance Tuning Introduction

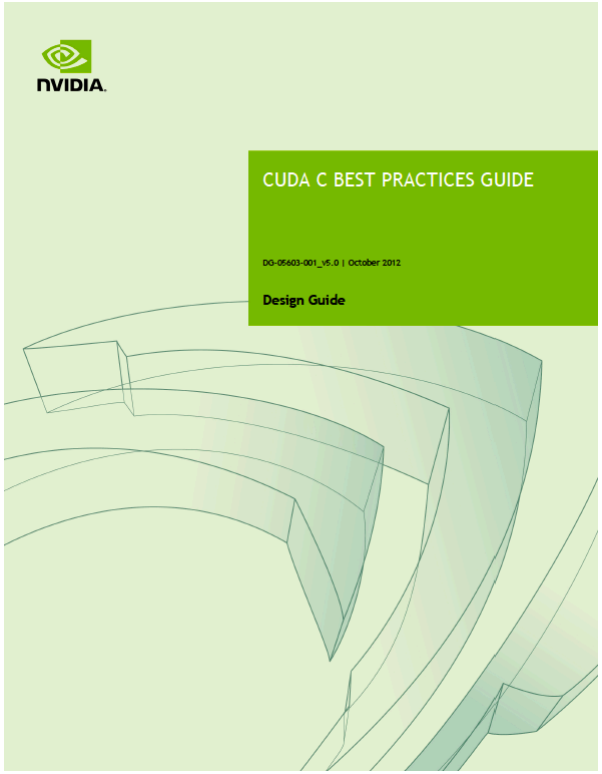

Hans Henrik Brandenborg Sørensen
DTU Computing Center
DTU Compute
<hhbs@dtu.dk>



Overview

- Recap from week 1 (now with GPUs)
  - Performance metrics
  - Measuring runtime
  - Assessing your performance
- Speed-up GPU vs. CPU (and what is ‘fair’?)
- Cuda performance tuning process
  - Transpose example
  - How many threads should I launch?
  - Latency hiding

CUDA C Best Practices Guide



https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

1. Assessing your application.
2. Heterogeneous computing.
3. Application profiling.
4. Parallelizing your application
5. Getting started.
6. Getting the right answer.
7. Optimizing CUDA applications.
8. Performance metrics.
9. Memory optimizations.
10. Execution configuration optimizations.
11. Instruction optimizations.
12. Control flow.
13. Deploying CUDA applications.
14. Understanding the programming environment.
15. Preparing for deployment.
16. Deployment infrastructure tools.

GPU performance metrics

Performance tuning terminology

- Execution time [seconds]
  - Time to run the application (wall or cpu/gpu)
- Performance [Gflops]
  - How many floating point operations per second
- Latency [cycles or seconds]
  - Time from initiating a memory access or other action until the result is available
- Bandwidth [Gb/s]
  - The rate at which data can be transferred
- Blocking [blocksize]
  - Dividing matrices into tiles to fit memory hierarchy

You know these from week 1+2 of this course!

Performance tuning terminology

- Throughput [#/s or Gb/s]
  - Sustained rate for instructions executed or data reads + writes achieved in practice
- Occupancy [%]
  - Ratio of active warps to max possible active warps
- Instruction level parallelism (ILP) [#]
  - How many independent instructions can be executed (=pipelining)
- Thread level parallelism (TLP) [#]
  - How many independent threads can be launched
- Coalescing
  - All threads in a warp are reading data from a contiguous, aligned, region of global memory

Common GPU optimization teminology.

Measuring runtime for kernels

- Using CPU timers
  - Run your kernel several times and compute average
  - Remember that kernel launches are non-blocking
    - Add `cudaDeviceSynchronize()`
  - GPUs have a “wake-up” time to create CUDA context

Measuring runtime for kernels

- Using CPU timers
  - Run your kernel several times and compute average
  - Remember that kernel launches are non-blocking
    - Add `cudaDeviceSynchronize()`
  - GPUs have a “wake-up” time to create CUDA context
- E.g., use the OpenMP timer

```
#include <omp.h>

double time = omp_get_wtime();

kernel<<<dimGrid, dimBlock>>>();
cudaDeviceSynchronize();

double elapsed = omp_get_wtime() - time;
```

Measuring runtime for kernels

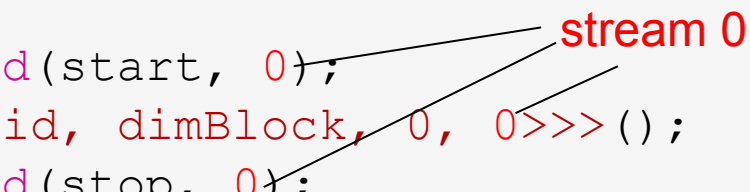
- Using CUDA GPU timers

```
cudaEvent_t start, stop;
float elapsed;

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

cudaEventRecord(start, 0);
kernel<<<dimGrid, dimBlock, 0, 0>>>();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsed, start, stop);

cudaEventDestroy(start); cudaEventDestroy(stop);
```



Events should be used, e.g., if you want separate runtimes of several kernels running simultaneously

Which performance metric to use?

- Compute bound
  - Limited by # flops * time per flop

Gflops = # floating point operations / 10^9 / runtime

Which performance metric to use?

- Compute bound
  - Limited by # flops * time per flop

Gflops = # floating point operations / 10^9 / runtime

- Memory bound
  - Limited by # bytes moved / bandwidth

Bandwidth = (Bytes_read + Bytes_written) / 10^9 / runtime

How to assess your performance?

- Compute bound
  - Compare with the theoretical peak performance
    - Run device query to find specs and calculate, e.g.

SP: 6912 cores * 1.41 GHz * 2 flops per core = 19492 Gflops

DP: (1/2) * 19492 Gflops = 9746 Gflops

How to assess your performance?

- Compute bound
  - Compare with the theoretical peak performance
    - Run device query to find specs and calculate, e.g.

SP: 6912 cores * 1.41 GHz * 2 flops per core = 19492 Gflops

DP: (1/2) * 19492 Gflops = 9746 Gflops

- Memory bound
  - Compare with the theoretical peak bandwidth
    - Run device query to find specs and calculate, e.g.

Peak bandwidth = 1.215 GHz*(5120 / 8) bytes*2 = 1555 GB/s

How to assess your performance?

- Compute bound
  - Compare with the theoretical peak performance
    - Run device query to find specs and calculate, e.g.

SP: 6912 cores = 19492 Gflops

DP: = 9746 Gflops

40-60%: okay
60-75%: good
>75%: excellent

- Memory b
  - Compare with the theoretical peak bandwidth
    - Run device query to find specs and calculate, e.g.

Peak bandwidth = 1.215 GHz*(5120 / 8) bytes*2 = 1555 GB/s

How to assess CPU performance?

- Find the CPU specs online:
  https://www.intel.com/content/www/us/en/products/processors/xeon/scalable/gold-processors/gold-6226r.html
  https://en.wikichip.org/wiki/intel/xeon_gold/6226r

How to assess CPU performance?

- Find the CPU specs online:
  https://www.intel.com/content/www/us/en/products/processors/xeon/scalable/gold-processors/gold-6226r.html
  https://en.wikichip.org/wiki/intel/xeon_gold/6226r
- Compute bound

SP: 16 cores * 2.8 GHz (AVX-512) * 16 (AVX-512) * 2 (FMA units) * 2 flops per core = 2866 Gflops

DP: (1/2) * 2866 Gflops = 1433 Gflops

- Memory bound

Peak bandwidth = 2.933 GHz * (64 / 8) bytes * 6 = 141 GB/s

Speed-up

Speed-up (now with GPUs)

- GPU definition of speed-up is traditionally

$$\texttt{Speedup} = \frac{\texttt{CPUtime}[\mathrm{s}]}{\texttt{GPUtime}[\mathrm{s}]}$$

  and usually written in times ($\times$) manner, e.g., 3.2$\times$

- Useful for indicating performance without telling what the performance actually is(!)

Speed-up (now with GPUs)

- GPU definition of speed-up is traditionally

$$\texttt{Speedup} = \frac{\texttt{CPUtime}[\mathrm{s}]}{\texttt{GPUtime}[\mathrm{s}]}$$

  and usually written in times ($\times$) manner, e.g., 3.2$\times$

- Useful for indicating performance without telling what the performance actually is(!)

But be fair when comparing to CPU times – speed-ups of 100×-1000× (see `http://www.nvidia.com/object/cuda_showcase_html.html`) are unrealistic when the hardware specs are taken into account

Speed-up (now with GPUs)

- Expected speed-up on our nodes
  - Compute bound: 9746 / 1433 = 6.8x
  - Memory bound: 1555 / 141 = 11.0x

Speed-up (now with GPUs)

- Expected speed-up on our nodes
  - Compute bound: 9746 / 1433 = 6.8x
  - Memory bound: 1555 / 141 = 11.0x
- Better speed-ups may be seen in practice
  - Execution model (SIMD / SIMT)
  - Compiler matureness (vectorization is difficult)
  - …but not by orders of magnitude(!)

Example of speed-up calculation

$$\pi = \int_0^1 \frac{4}{1+x^2}\mathrm{d}x \approx \frac{1}{N}\sum_{i=1}^{N} \frac{4}{1+\left(\frac{i-0.5}{N}\right)^2}.$$

```
double pi(long N)
{
    double sum = 0.0;
    double h = 1.0/N;
    #pragma omp parallel for \
        reduction(+: sum)
    for(long i=1; i<=N; i++) {
       double x = h*(i-0.5);
       sum += 4.0/(1.0+x*x);
    }
    return h*sum;
}
```

Example of speed-up calculation

$$\pi = \int_0^1 \frac{4}{1+x^2}\mathrm{d}x \approx \frac{1}{N}\sum_{i=1}^{N} \frac{4}{1+\left(\frac{i-0.5}{N}\right)^2}.$$


```
double pi(long N)
{
    double sum = 0.0;
    double h = 1.0/N;
    #pragma omp parallel for \
        reduction(+: sum)
    for(long i=1; i<=N; i++) {
        double x = h*(i-0.5);
        sum += 4.0/(1.0+x*x);
    }
    return h*sum;
}
```

```
icc -O3 -fopenmp pi_cpu.cpp
```

OMP_NUM_THREADS	-O3 (s)
1	97.7
2	50.3
4	26.2
8	13.2
16	6.6
32	3.4

Example of speed-up calculation

$$\pi = \int_0^1 \frac{4}{1+x^2}\mathrm{d}x \approx \frac{1}{N}\sum_{i=1}^{N} \frac{4}{1+\left(\frac{i-0.5}{N}\right)^2}.$$


```
double pi(long N)
{
    double sum = 0.0;
    double h = 1.0/N;
    #pragma omp parallel for \
        reduction(+: sum)
    for(long i=1; i<=N; i++) {
        double x = h*(i-0.5);
        sum += 4.0/(1.0+x*x);
    }
    return h*sum;
}
```

```
icc -O3 -fopenmp pi_cpu.cpp
icc -xcore-avx512 -O3 …
```

OMP_NUM_THREADS	-O3 (s)	avx (s)
1	97.7	29.1
2	50.3	14.9
4	26.2	8.0
8	13.2	4.4
16	6.6	2.7
32	3.4	1.4

Example of speed-up calculation

$$\pi = \int_0^1 \frac{4}{1+x^2}\mathrm{d}x \approx \frac{1}{N}\sum_{i=1}^{N}\frac{4}{1+\left(\frac{i-0.5}{N}\right)^2}.$$


```
double pi(long N)
```

remark #15305: vectorization support: vector length **2**
remark #15399: vectorization support: unroll factor set to 4
remark #15300: LOOP WAS VECTORIZED
remark #15486: divides: 1

```
    #pragma omp parallel for \
```

remark #15305: vectorization support: vector length **8**
remark #15399: vectorization support: unroll factor set to 4
remark #15300: LOOP WAS VECTORIZED
remark #15486: divides: 1

```
        sum += 4.0/(1.0+x*x);
    }
    return h*sum;
}
```

```
icc -O3 -fopenmp pi_cpu.cpp
icc -xcore-avx512 -O3 …
```

OMP_NUM_THREADS	-O3 (s)	avx (s)
1	97.7	29.1
2	50.3	14.9
4	26.2	8.0
8	13.2	4.4
16	6.6	2.7
32	3.4	1.4

Example of speed-up calculation

$$\pi = \int_0^1 \frac{4}{1+x^2}\mathrm{d}x \approx \frac{1}{N}\sum_{i=1}^{N} \frac{4}{1+\left(\frac{i-0.5}{N}\right)^2}.$$

```
__global__
void pi(long N, double *res)
{
    long idx = threadIdx.x+
        blockDim.x*blockIdx.x;
    double sum = 0.0;
    double h = 1.0/N;
    for(long i=idx+1; i<=N;
     i+=blockDim.x*gridDim.x){
        double x = h*(i-0.5);
        sum += 4.0/(1.0+x*x);
    }
    res[idx] = h*sum;
}
```

Example of speed-up calculation

$$\pi = \int_0^1 \frac{4}{1+x^2}\mathrm{d}x \approx \frac{1}{N}\sum_{i=1}^{N}\frac{4}{1+\left(\frac{i-0.5}{N}\right)^2}.$$

```
__global__
void pi(long N, double *res)
{
    long idx = threadIdx.x+
        blockDim.x*blockIdx.x;
    double sum = 0.0;
    double h = 1.0/N;
    for(long i=idx+1; i<=N;
     i+=blockDim.x*gridDim.x){
        double x = h*(i-0.5);
        sum += 4.0/(1.0+x*x);
    }
    res[idx] = h*sum;
}
```

```
nvcc -O3 -lgomp pi_gpu.cu
```

blockDim	gridDim	nvcc (s)
1	2048	10.00
2	2048	5.06
16	2048	0.62
32	2048	0.31
64	2048	0.29
128	2048	0.26
256	2048	0.25

Example of speed-up calculation

$$\pi = \int_0^1 \frac{4}{1+x^2}\mathrm{d}x \approx \frac{1}{N}\sum_{i=1}^{N} \frac{4}{1+\left(\frac{i-0.5}{N}\right)^2}.$$

1 CPU / # threads = # cores / `avx512`

Speed-up = 2.7 seconds / 0.25 seconds = 10.8×

1 GPU / best launch configuration / deduct warm up

CUDA performance tuning process

CUDA performance tuning process

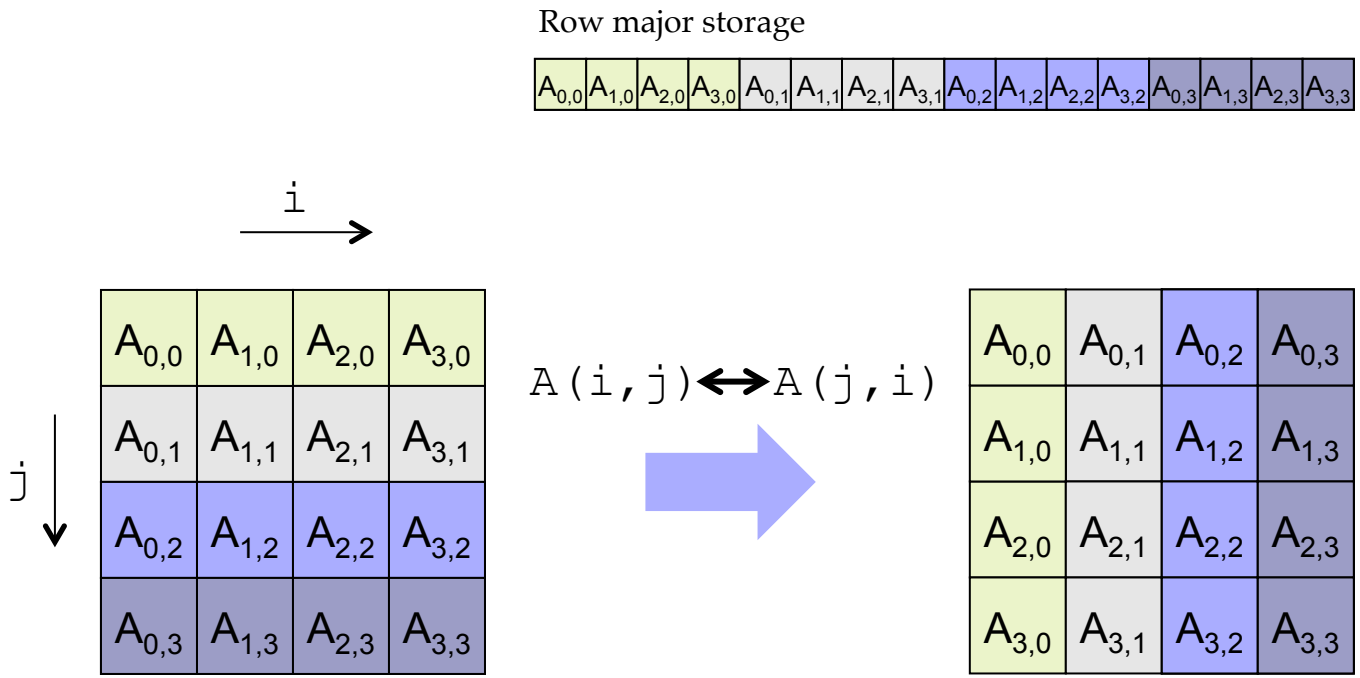
- Determine what limits kernel performance
  - Parallelism (concurrency bound)
  - Memory accesses (memory bound)
  - Floating point operations (compute bound)

… or a combination

- Use appropriate performance metric for the kernel
  - Or use speed-up between kernel modifications
- Address the limiters in the order of importance
  - Determine how close you are to the theoretical peaks
  - Analyze
  - Apply optimizations

… and iterate with small steps

Transpose example



- We will use this example to illustrate the process of performance tuning a CUDA kernel “step-by-step”

Transpose example (v1 serial)

```
// Reference serial CPU transpose
__host__ __device__
void transpose(double *A, double *At)
{
    for(int j=0; j < N; j++)
        for(int i=0; i < N; i++)
            At[j + i*N] = A[i + j*N]; // At(j,i)=A(i,j)
}
```

```
// Kernel to be launched on a single thread
__global__
void transpose_serial(double *A, double *At)
{
     transpose(A, At);
}
```

Transpose example (v1 serial)

```
#define N 2880
...
    // CPU reference transpose for checking result
    transpose(h_A, h_At_CPU);

    // Transfer matrix to device
    cudaMemcpy(d_A, h_A, A_size, cudaMemcpyHostToDevice);

    transpose_serial<<<1, 1>>>(d_A, d_At);
    cudaDeviceSynchronize();

    // Transfer result to host
    cudaMemcpy(h_At, d_At, A_size, cudaMemcpyDeviceToHost);
...
```

Transpose example (v1 serial)

```
$ nvprof --print-gpu-summary ./transpose_gpu
==30836== Profiling application: ./transpose_gpu
==30836== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 98.10%  2.52254s         1  2.52254s  2.52254s  2.52254s  transpose_serial(double*, double*)
  1.08%  27.837ms         1  27.837ms  27.837ms  27.837ms  [CUDA memcpy DtoH]
  0.80%  20.670ms         1  20.670ms  20.670ms  20.670ms  [CUDA memcpy HtoD]
  0.01%  331.49us         1  331.49us  331.49us  331.49us  [CUDA memset]
$
```

Version	**v1 serial**
Time [ms]	2523

Transpose example (v2 per row)

```
// Kernel to be launched with one thread per row of A
__global__
void transpose_thread_per_row(double *A, double *At)
{
   // Thread index gives row of A
   int j = blockIdx.x * blockDim.x + threadIdx.x;

   for(int i=0; i < N; i++)
      At[j + i*N] = A[i + j*N]; // At(j,i)=A(i,j)
}
```

```
...
   transpose_per_row<<<15, 192>>>(d_A, d_At);
...
```

Transpose example (v2 per row)

```
$ nvprof --print-gpu-summary ./transpose_gpu
==32362== Profiling application: ./transpose_gpu
==32362== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 90.04%  438.38ms       100  4.3838ms  4.2700ms  4.7287ms  transpose_per_row(double*, double*)
  5.68%  27.679ms         1  27.679ms  27.679ms  27.679ms  [CUDA memcpy DtoH]
  4.21%  20.499ms         1  20.499ms  20.499ms  20.499ms  [CUDA memcpy HtoD]
  0.07%  330.34us         1  330.34us  330.34us  330.34us  [CUDA memset]
$
```

Version	v1 serial	v2 per row
Time [ms]	2523	4.38

Transpose example (v3 per elm)

```
// Kernel to be launched with one thread per element of A
__global__
void transpose_per_elm(double *A, double *At)
{
    // 2D thread indices defining row and col of element
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    At[j + i*N] = A[i + j*N]; // At(j,i)=A(i,j)
}
```

```
#define K 16
...
    transpose_per_elm<<<dim3(N/K, N/K), dim3(K,K)>>>(d_A,
d_At);
...
```

Transpose example (v3 per elm)

```
$ nvprof --print-gpu-summary ./transpose_gpu
==3324== Profiling application: ./transpose_gpu
==3324== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 74.88%  153.66ms       100  1.5366ms  1.5283ms  1.5410ms  transpose_per_elm(double*, double*)
 13.60%  27.914ms         1  27.914ms  27.914ms  27.914ms  [CUDA memcpy DtoH]
 11.36%  23.310ms         1  23.310ms  23.310ms  23.310ms  [CUDA memcpy HtoD]
  0.16%  331.20us         1  331.20us  331.20us  331.20us  [CUDA memset]
$
```

Version	v1 serial	v2 per row	v3 per elm
Time [ms]	2523	4.38	1.54

How many threads should I run?

- Why is one thread per CUDA core not enough?
  - E.g. `kernel<<<15, 192>>>(...);`

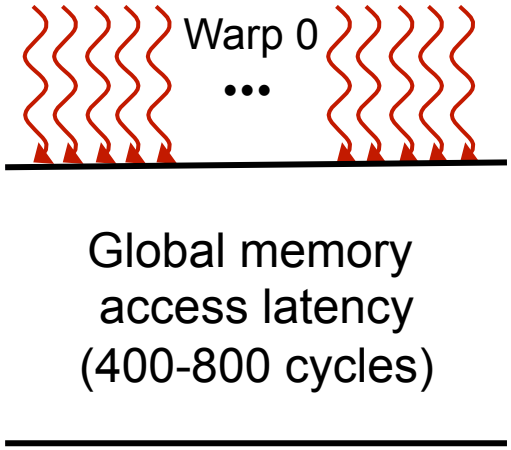
How many threads should I run?

- Why is one thread per CUDA core not enough?
  - E.g. `kernel<<<15, 192>>>(...);`

How many threads should I run?

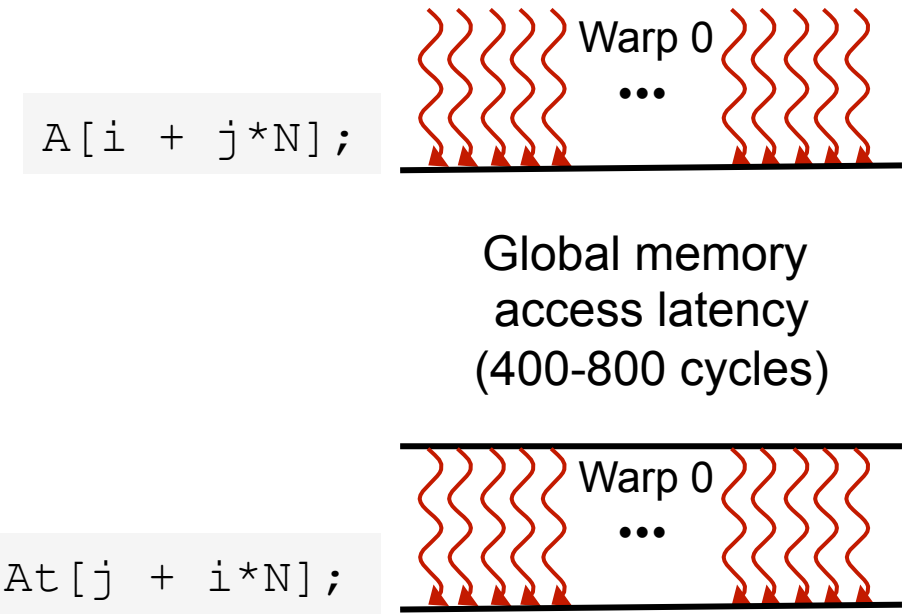
- Why is one thread per CUDA core not enough?
  - E.g. `kernel<<<15, 192>>>(...);`

```
A[i + j*N];
```



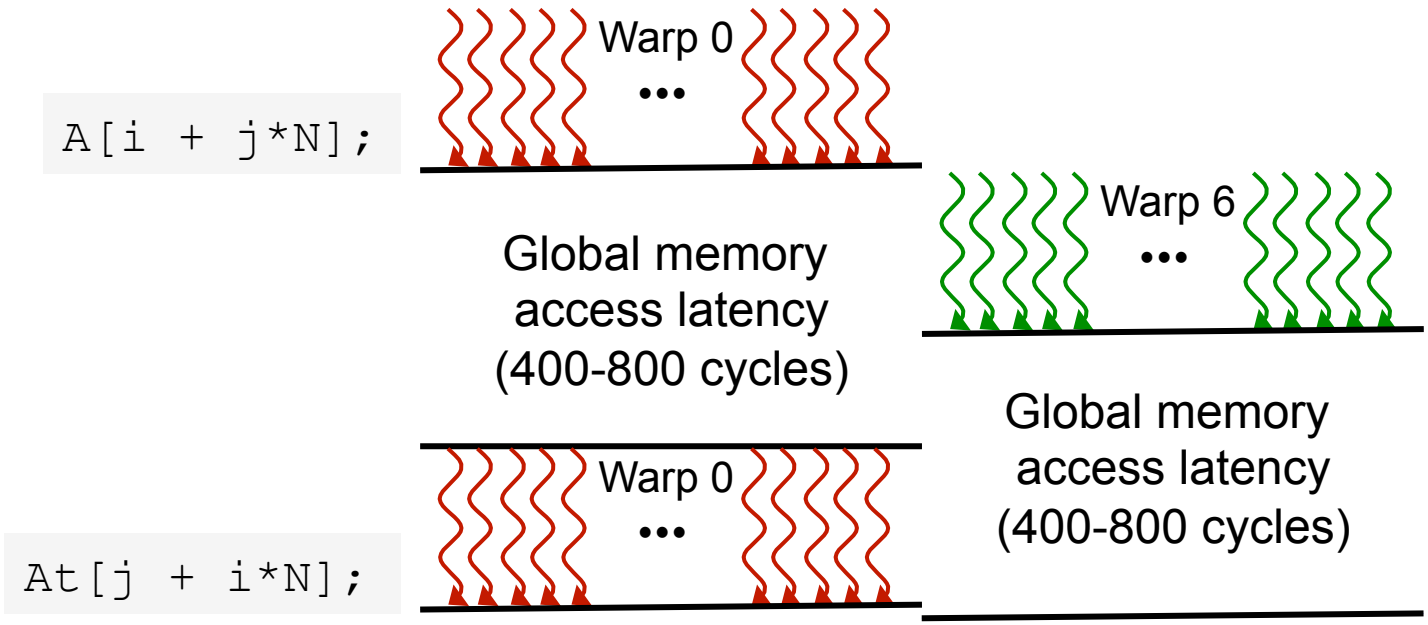
How many threads should I run?

- Why is one thread per CUDA core not enough?
  - E.g. `kernel<<<15, 192>>>(...);`



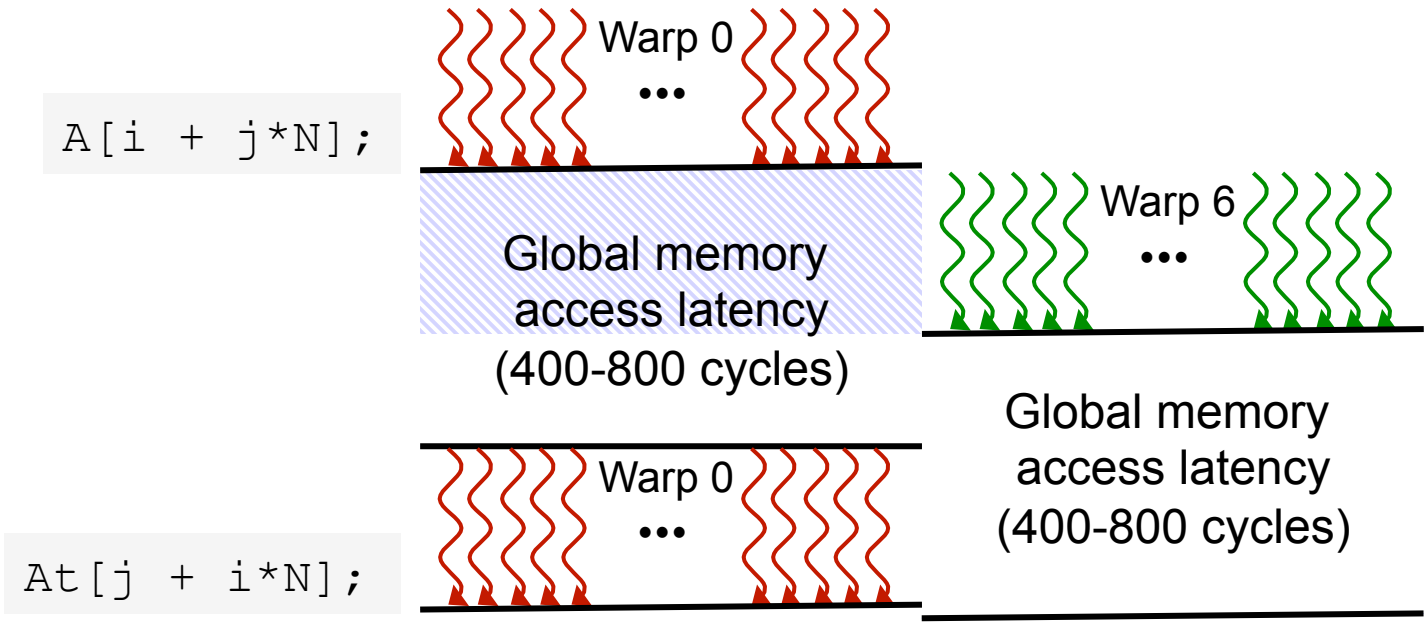
How many threads should I run?

- Why is one thread per CUDA core not enough?
  - E.g. `kernel<<<15, 192>>>(...);`



How many threads should I run?

- Why is one thread per CUDA core not enough?
  - E.g. `kernel<<<15, 192>>>(...);`



- Reason: We can hide memory latency by having idle warps to schedule while waiting for data

End of lecture