Introduction to GPU performance tuning

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

Overview

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

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
  - 32 neighbor threads in team are reading data from a contiguous, aligned, region of global memory

Common GPU optimization teminology.

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 (×) manner, e.g., 3.2 ×

- 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 (×) manner, e.g., 3.2 ×

- 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) – see next slide
  - Compiler matureness (vectorization is difficult)
  - …but not by orders of magnitude(!)

Executing of the GPU

- Execution model maps code to instructions
  - SIMD – Single Instruction, Multiple Data
  - SIMT – Single Instruction, Multiple Threads
- CPU (SIMD mapping)
  - The compiler takes care of mapping code to the most efficient instructions at compile time
- GPU (SIMT mapping)
  - Kernels look like serial functions for a SINGLE thread
  - CUDA will automatically launch on MANY threads
  - Code is mapped to instructions at runtime!

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

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

```
double pi(long N)
{
   double sum = 0.0;
   double h = 1.0/N;
   #pragma omp target teams \
       num_teams(16384) \
       thread_limit(256) \
       distribute 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 target teams \
       num_teams(16384) \
       thread_limit(256) \
       distribute 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;
}
```

```
nvc++ -fast -mp=gpu pi_gpu.cpp
```

num_teams	thread_limit	nvc (s)
16384	1	10.06
16384	2	5.03
16384	4	2.52
16384	32	0.33
16384	64	0.32
16384	128	0.31
16384	256	0.31

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 / 0.31 seconds = 8.7 ×

1 GPU / best teams parallel configuration / deduct warm up

GPU performance tuning process

GPU performance tuning process

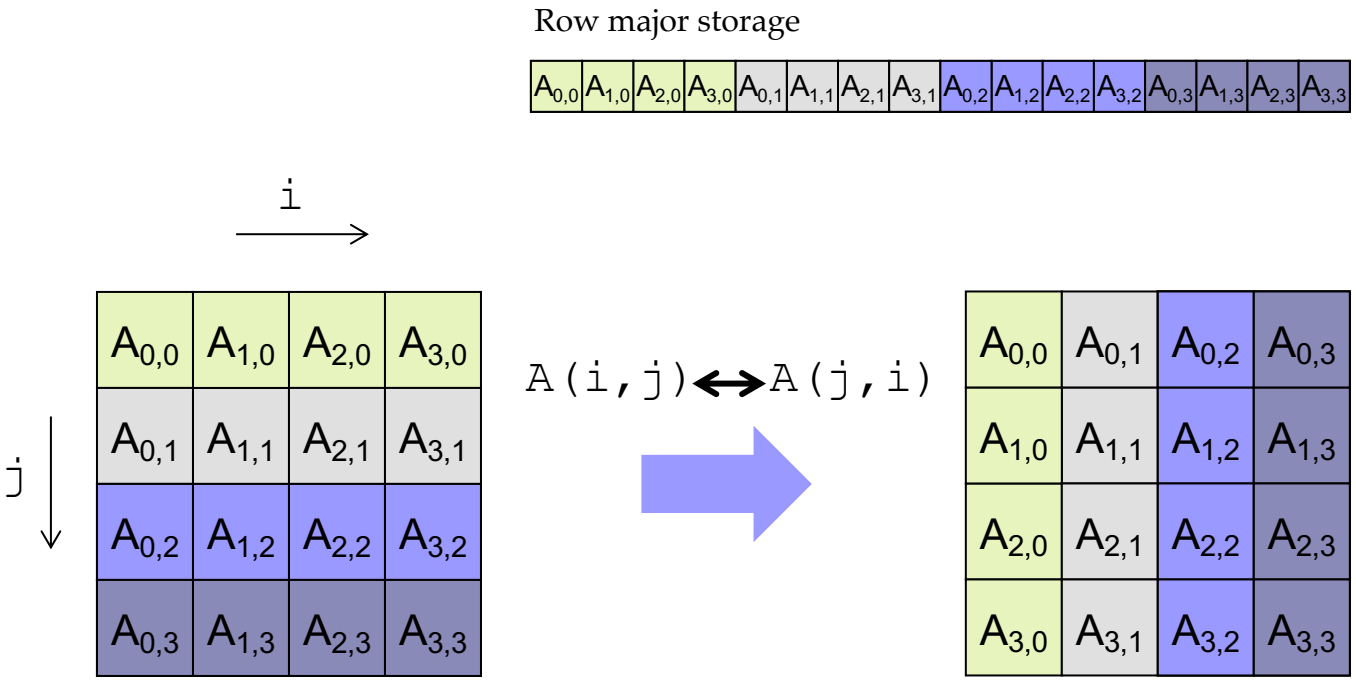
- Determine what limits the offload performance
  - Parallelism (concurrency bound)
  - Memory accesses (memory bound)
  - Floating point operations (compute bound)

… or a combination

- Use appropriate performance metric for the offload
  - 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 an OpenMP offload code "step-by-step"

Transpose example (v1 seq)

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

```
// Baseline sequential GPU transpose
#pragma omp declare target(transpose)
void transpose_seq(double **A, double **At)
{
    #pragma omp target
    transpose(A, At);
}
```

Transpose example (v1 seq)

```
#define N 6912
...
   // CPU reference transpose for checking result
   transpose(A, At_CPU);

   #pragma omp target enter data \
       map(to: A[0:N][0:N]) map(alloc: At[0:N][0:N])

    // GPU sequential version
    transpose_seq(A, At);

    #pragma omp target exit data \
        map(release: A[0:N][0:N]) map(from: At[0:N][0:N])

    check(At, At_CPU); // Check result
...
```

Transpose example (v1 seq)

```
$ nvc++ -fast -Msafeptr -Minfo -mp=gpu -gpu=cc80 -acc -c -o transpose.o transpose.cpp
...
transpose_seq(double **, double **):
     14, #omp target
         14, Generating "nvkernel__Z13transpose_seqPPdS0_l_F1L14_3" GPU kernel
     19, Loop not vectorized: unprofitable for target
         Loop unrolled 4 times
```

```
& nsys profile --trace=cuda --stats=true ./transpose
...
[4/6] Executing 'gpukernsum' stats report
```

```
Time (%)  Total Time (ns)  Instances      Avg (ns)    ..    GridXYZ         BlockXYZ                        Name
--------  ---------------  ---------  ----------------  --------------  --------------  ----------------------------------------
   100.0   10,122,171,751          1  10,122,171,751.0   108     1     1   128     1     1  nvkernel__Z13transpose_seqPPdS0_l_F1L27_3
     0.0          277,248          2         138,624.0   108     1     1   128     1     1  nvkernel__Z13malloc_2d_deviiPPd_F1L45_2
```

`nsys` command line profiling tool

Version	v1 seq
Time [ms]	10122

Transpose example (v2 per col)

```
// OpenMP offload transpose using one thread per col of A
void transpose_per_col(double **A, double **At)
{
    #pragma omp target teams distribute parallel for \
        num_teams(108) thread_limit(64)
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            At[i][j] = A[j][i];
}
```

Transpose example (v2 per col)

```
$ nvc++ -fast -Msafeptr -Minfo -mp=gpu -gpu=cc80 -acc -c -o transpose.o transpose.cpp
...
transpose_per_col(double **, double **):
     52, #omp target teams distribute parallel for num_teams(108) thread_limit(64)
         52, Generating "nvkernel__Z17transpose_per_colPPdS0_l_F1L52_13" GPU kernel
         58, Loop parallelized across teams and threads(128), schedule(static)
     58, Loop not vectorized/parallelized: not countable
     59, Loop not vectorized: unprofitable for target
         Loop unrolled 4 times

& nsys profile --trace=cuda --stats=true ./transpose
...
[4/6] Executing 'gpukernsum' stats report
```

```
Time (%)  Total Time (ns)  Instances   Avg (ns)    ..    GridXYZ         BlockXYZ                          Name
--------  ---------------  ---------  -----------  --------------  --------------  ----------------------------------------------
    99.9      466,973,250        100  4,669,732.5   108    1    1    64    1    1  nvkernel__Z17transpose_per_colPPdS0_l_F1L52_13
     0.1          276,608          2    138,304.0   108    1    1   128    1    1  nvkernel__Z13malloc_2d_deviiPPd_F1L45_2
```

Version	v1 seq	v2 per col
Time [ms]	10122	4.67

Transpose example (v3 per elm)

```
// OpenMP offload transpose using one thread per element
void transpose_per_elm(double **A, double **At)
{
    #pragma omp target teams distribute parallel for \
        collapse(2) num_teams(N*N/64) thread_limit(64)
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            At[i][j] = A[j][i];
}
```

Transpose example (v3 per elm)

```
$ nvc++ -fast -Msafeptr -Minfo -mp=gpu -gpu=cc80 -acc -c -o transpose.o transpose.cpp
...
transpose_per_elm(double **, double **):
     63, #omp target teams distribute parallel for num_teams(746496) thread_limit(64)
         63, Generating "nvkernel__Z17transpose_per_elmPPdS0_l_F1L63_17" GPU kernel
         72, Loop parallelized across teams and threads(128), schedule(static)
     73, Loop not vectorized/parallelized: not countable
```

```
& nsys profile --trace=cuda --stats=true ./transpose
...
[4/6] Executing 'gpukernsum' stats report
```

```
Time (%)  Total Time (ns)  Instances   Avg (ns)  ..    GridXYZ          BlockXYZ                          Name
--------  ---------------  ---------  ----------  ----------------  --------------  ----------------------------------------------
    99.7       77,821,032        100   778,210.3  746496    1    1    64    1    1  nvkernel__Z17transpose_per_elmPPdS0_l_F1L64_17
     0.3          276,768          2   138,384.0     108    1    1   128    1    1  nvkernel__Z13malloc_2d_deviiPPd_F1L45_2
```

Version	v1 seq	v2 per col	v3 per elm
Time [ms]	10122	4.67	0.78

How many threads should I run?

- Why is one thread per core not enough?
  - E.g. `num_teams(108) thread_limit(64)`

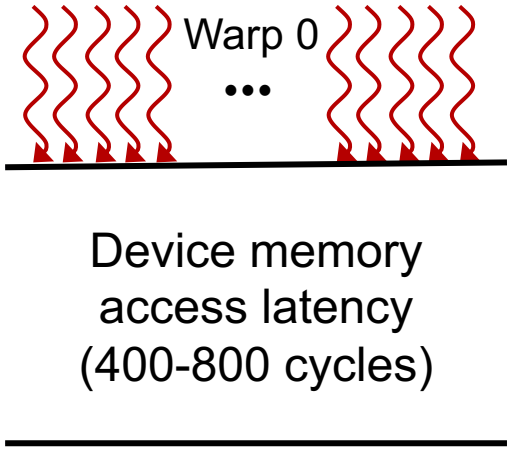
How many threads should I run?

- Why is one thread per core not enough?
  - E.g. `num_teams(108) thread_limit(64)`

How many threads should I run?

- Why is one thread per core not enough?
  - E.g. `num_teams(108) thread_limit(64)`

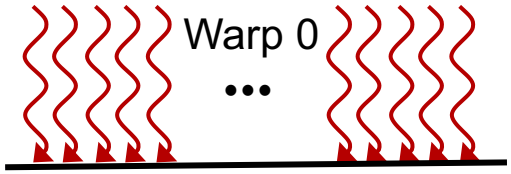
```
A[j][i];
```



How many threads should I run?

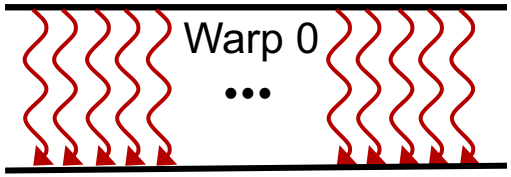
- Why is one thread per core not enough?
  - E.g. `num_teams(108) thread_limit(64)`

```
A[j][i];
```



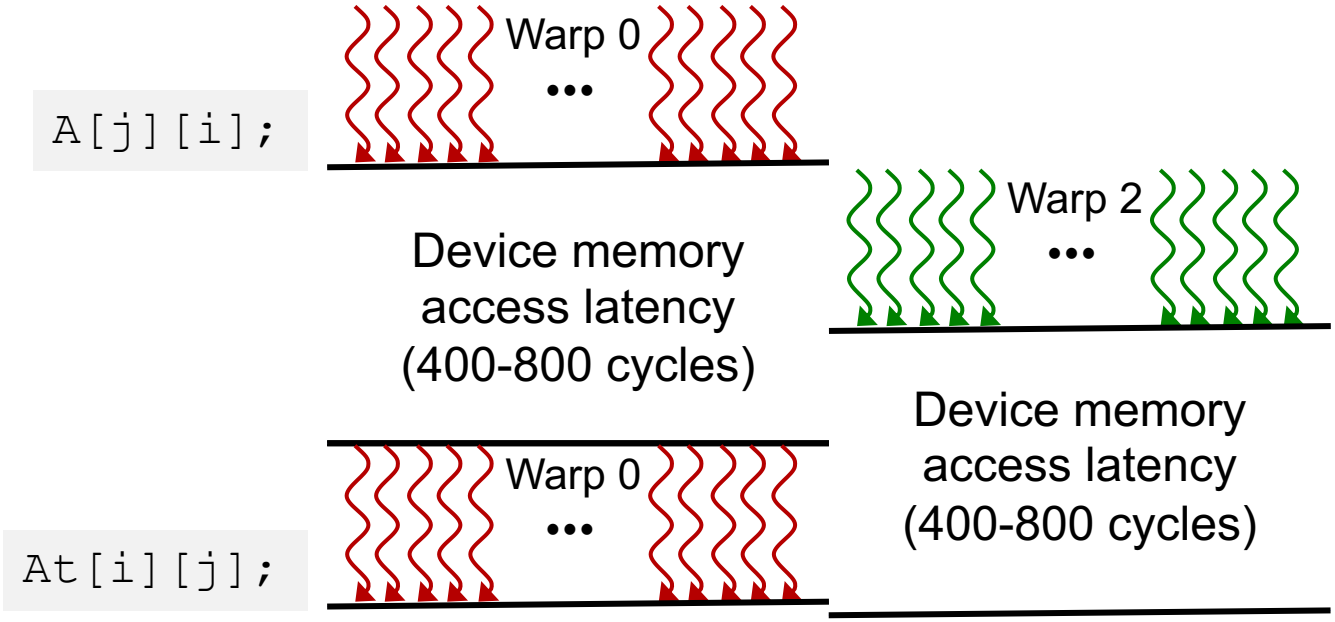
Device memory
access latency
(400-800 cycles)

```
At[i][j];
```



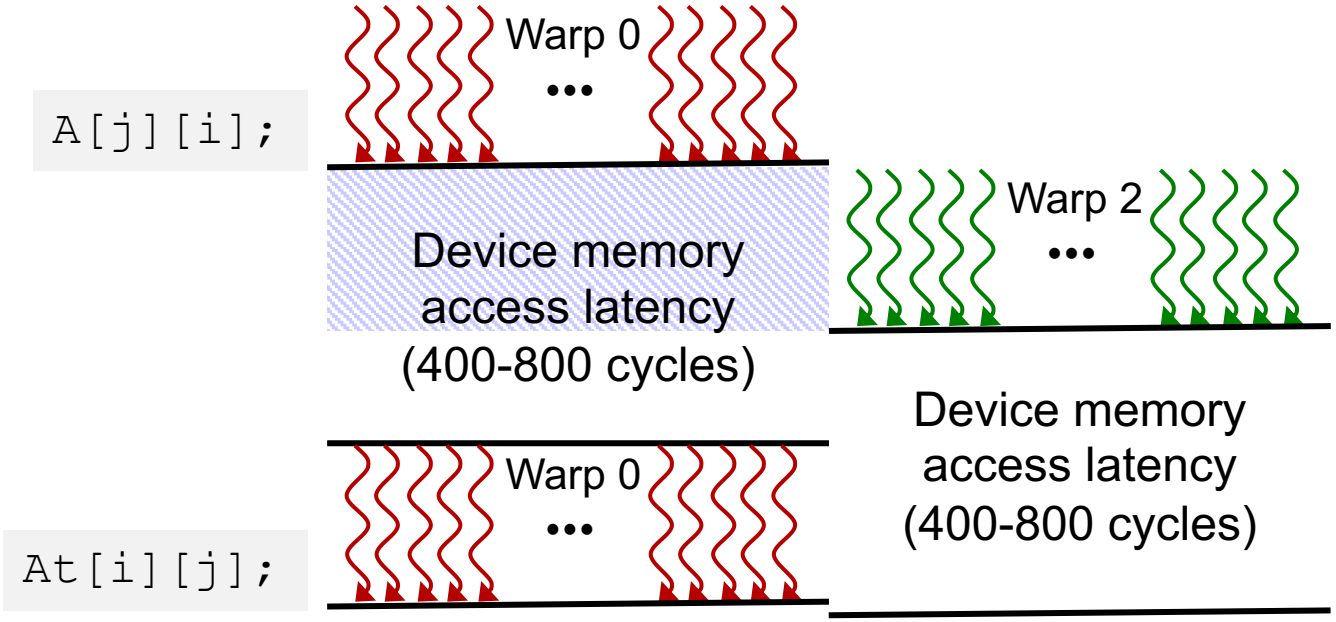
How many threads should I run?

- Why is one thread per core not enough?
  - E.g. `num_teams(108) thread_limit(64)`



How many threads should I run?

- Why is one thread per core not enough?
  - E.g. `num_teams(108) thread_limit(64)`



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

Transpose example (v4 warp)

```
// OpenMP offload transpose using one warp per col
void transpose_warp_per_col(double **A, double **At)
{
    #pragma omp target teams loop \
        num_teams(N) thread_limit(32)
    for(int i = 0; i < N; i++) {
        #pragma omp loop bind(parallel)
        for(int j = 0; j < N; j++)
            At[i][j] = A[j][i];
    }
}
```

Transpose example (v4 warp)

```
$ nvc++ -fast -Msafeptr -Minfo -mp=gpu -gpu=cc80 -acc -c -o transpose.o transpose.cpp
...
transpose_warp_per_col(double **, double **, long):
     80, #omp target teams loop num_teams(6912) thread_limit(32)
         80, Generating "nvkernel__Z22transpose_warp_per_colPPdS0_l_F1L80_19" GPU kernel
             Generating NVIDIA GPU code
           86, Loop parallelized across teams(6912) /* blockIdx.x */
           88, Loop parallelized across threads(32) /* threadIdx.x */
         80, Generating Multicore code
           86, Loop parallelized across threads
     88, Loop is parallelizable
         Loop not vectorized: unprofitable for target
         Loop unrolled 4 times

& nsys profile --trace=cuda --stats=true ./transpose
...
[4/6] Executing 'gpukernsum' stats report
```

Time (%)	Total Time (ns)	Instances	Avg (ns)	..	GridXYZ			BlockXYZ			Name
99.7	67,950,144	100	679,501.4		6912	1	1	32	1	1	nvkernel__Z22transpose_warp_per_colPPdS0_l_F1L8
0.3	277,184	2	138,592.0		108	1	1	128	1	1	nvkernel__Z13malloc_2d_deviiPPd_F1L45_2

Version	v1 seq	v2 per col	v3 per elm	v4 warp
Time [ms]	10122	4.67	0.78	0.68

End of lecture