GPU programming in CUDA: How to write efficient CUDA programs

Lukas Einkemmer
Department of Mathematics
University of Innsbruck

PRACE winter school, Innsbruck
Link to slides: `http://www.einkemmer.net/training.html`

Goals

Our goals in this section are

- Understand the performance characteristics of GPUs.
- Best practice for obtaining good performance.
- Commonly encountered issues that degrade performance (i.e. pitfalls).

Understanding performance limitations

Performance of a modern GPU

The theoretically achieved FLOPS are calculated as follows

$$(1.455\ \text{GHz}) \cdot (80\ \text{SM}) \cdot (64\ \text{CUDA cores}) \cdot (2\ \text{fused multiply add})$$
$$= 14.9\ \text{TFLOPS (single precision)}$$

7.45 TFLOPS (double precision).

Not the whole story

- Boost frequency might not be thermally feasible for some application.
- There is purpose built hardware for certain mathematical functions (sin, cos, ..).
- There is purpose built hardware for (small) matrix operations (Tensor Cores).

Performance of a modern GPU

Let us consider multiplying a vector by a scalar

```
__global__
void kernel(double* x, double* y, int n) {
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    if(i < n)
        y[i] = 3.0*x[i];
}

kernel<<<num_blocks, num_threads>>>(d_x, d_y, n);
```

Requires one memory read and one memory write per floating point operation.

- **Achieving 7.45 TFLOPS requires a memory transfer rate (bandwidth) of 119 TB/s**.
- State of the art hardware (V100) achieves at most 900 GB/s

Compute bound vs memory bound

A problem is **compute bound** if the performance is dictated by how many arithmetic operation the GPU can perform.

- numerical quadrature, solving dense linear system (LU), Monte Carlo methods, ...

A problem is **memory bound** if the performance is dictated by the bandwidth of main memory.

- stencil codes, solving sparse linear systems, FFT, ...
- performance measured in achieved GB/s

Understanding the characteristics of your problem is essential to guide optimization.

Compute bound vs memory bound

Count number of bytes transferred to/from memory **and number of flops** in your algorithm. We have a memory bound problem if

$$\left(\frac{\text{flop}}{\text{byte}}\right)_{\text{algorithm}} < \left(\frac{\text{flop}}{\text{byte}}\right)_{\text{hardware}} \approx 8.3 \quad \text{(V100, double precision)}.$$

For our earlier example

```
__global__ void kernel(double* x, double* y, int n) {
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    if(i < n)
        y[i] = 3.0*x[i];
}
```

we have

$$\left(\frac{\text{flop}}{\text{byte}}\right)_{\text{algorithm}} = \frac{n}{2 \cdot \text{sizeof(double)} \cdot n} \approx 0.06 \ll 8.3.$$

Compute bound vs memory bound

Scientific computing on GPUs (and on CPUs) leans heavily towards being memory bound.

- Caused by the high byte/flop ratio of GPUs.
- Important exceptions: machine learning, molecular dynamics

How many memory instructions?

```
__global__
void k_stencil(double* x, double* y, int n) {
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    if(i > 0 && i < n-1)
        y[i] = x[i+1]-x[i-1];
}
```

Memory access pattern

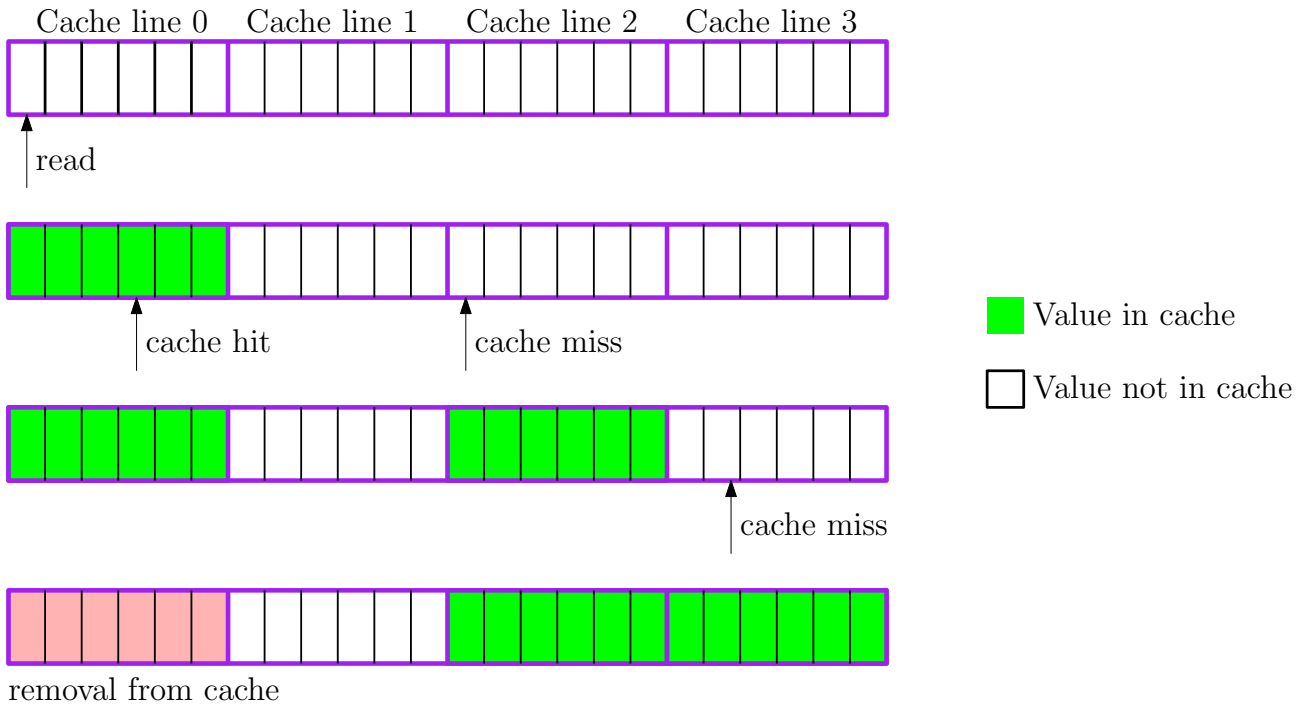
i=1 read x[0], x[2]
i=2 read x[1], x[3]
i=3 read x[2], x[4]
i=4 read x[3], x[5]

Caches

Knowledge of how caches work is important for performance.

Caches transfer data in chunks of fixed size (so-called cache lines)

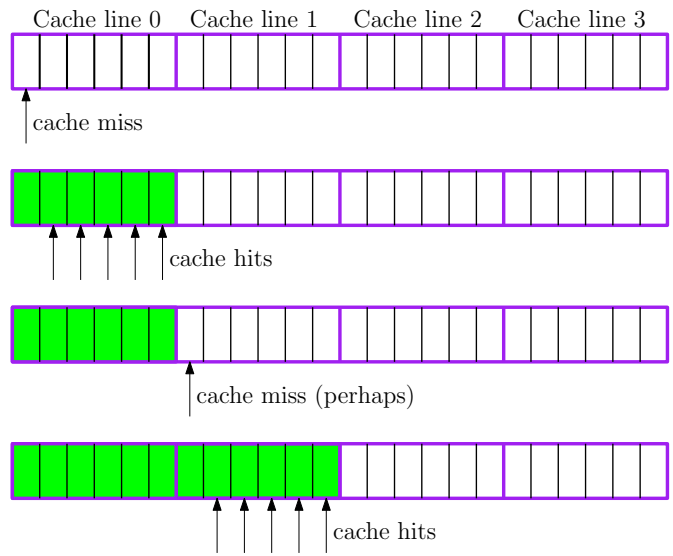
- usually 64-256 bytes in size (8-32 doubles)



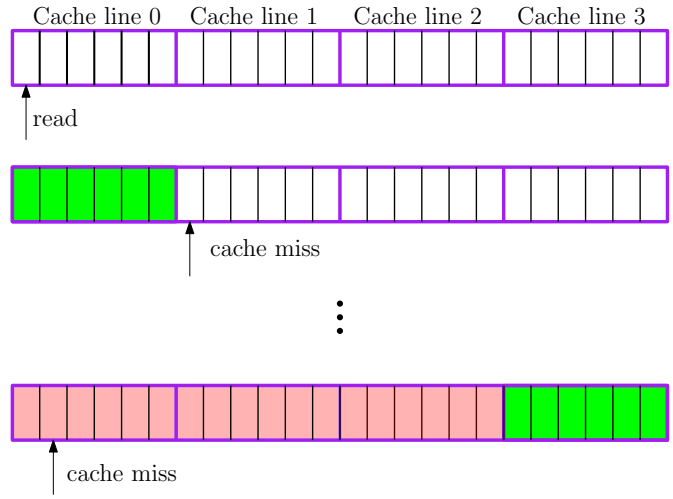
First read of any byte in a cache line transfers the entire cache line.

Memory access pattern

```
// Access with stride 1
__global__
void k_copy(double* x,
        double* y, int n) {
    int i = threadIdx.x
          + blockDim.x*blockIdx.x;
    if(i < n)
        y[i] = x[i];
}
```




```
// Access with larger stride
__global__
void k_copy(double* x,
        double* y, int n) {
    int i = blockIdx.x
          + gridDim.x*threadIdx.x;
    if(i < n)
        y[i] = x[i];
}
```



Caches

Often assuming that everything is cached perfectly is a good starting point.

- In the implementation we have to work for this!

There are **important differences** between **CPU and GPU caches**.

In CUDA caches can be **explicitely controlled.**

- Programmer has full control over what goes into the cache.
- We consider this **shared memory** later.

GPU caches are smaller and more simplistic.

- V100 has 6MB L2 cache (per device) and 128 kB L1 cache (per SM).
- CPUs use very sophisticated prefetching strategies.

Memory & data transfer

Avoid memory transfer between device and host

PCIe bandwidth is orders of magnitude slower than device memory.

Recommendation: Avoid memory transfer between device and host, if possible.

Recommendation: Copy your initial data to the device. Run your entire simulation on the device. Only copy data back to the host if needed for output.
To get good performance we have to live on the GPU.

Recommendation: Run only parts of your algorithm on the host for which the data transfer overhead is small.

RAM is not fast at random access

We are supposed to have **random access memory** (RAM).

```
cudaMemcpy(d_y, d_x, sizeof(double)*n,
    cudaMemcpyDeviceToDevice);
```

800 GB/s

```
__global__
void k_transpose(double* x, double* y) {
    int i = threadIdx.x;
    int j = blockIdx.x;

    y[i + blockDim.x*j] = x[j + gridDim.x*i];
}
```

200 GB/s

```
__global__
void k_random(double* x, double* y) {
    long n = blockDim.x*gridDim.x;
    y[rand() % n] = x[rand() % n];
}
```

20 GB/s

Favorable memory access pattern

Recommendation: Use contiguous memory access, if possible.

Recommendation: Well understood optimizations are available for strided memory access (i.e. **cache blocking**).

Recommendation: A problem with truly random (i.e. unpredictable) access might be better suited for the CPU.

- There is still a significant performance penalty for random memory access on the CPU.

Recommendation: Data structures can have a big impact on how memory is accessed.

- Array of structs vs structs of arrays.

Latency vs bandwidth

The time it takes to

- read n bytes from memory
- transfer n bytes from host to device

can be modeled by

$$\text{time to completion} = \text{latency} + \frac{n}{\text{bandwidth}}.$$

There are fundamental physical limits that prevent reduction in latency.

Waiting for one byte to arrive from memory could have been used to read 250 **kB.**

Latency

As a rule of thumb we pay the following penalty (in clock cycles)

Operation	CPU	GPU (V100)
arithmetics	1	4-8
Shared memory	–	20
L1 hit	1-10	30
LL hit	30	190
memory	200	400
PCIe	–	700
disk	10^5	10^5

Exact numbers depend on the specific architecture.

Z. Jia et al, arXiv:1804.06826.

Copy data from host to device

Situation 1:

```
cudaMemcpy(d_x, x, sizeof(double)*n,
    cudaMemcpyHostToDevice);
```

10 GB/s (100 MB)

Situation 2:

```
long offset = n/num;
for(int i=0;i<num;i++)
    cudaMemcpy(d_x+i*offset,
    x+i*offset,
    sizeof(double)*n/num,
    cudaMemcpyHostToDevice);
```

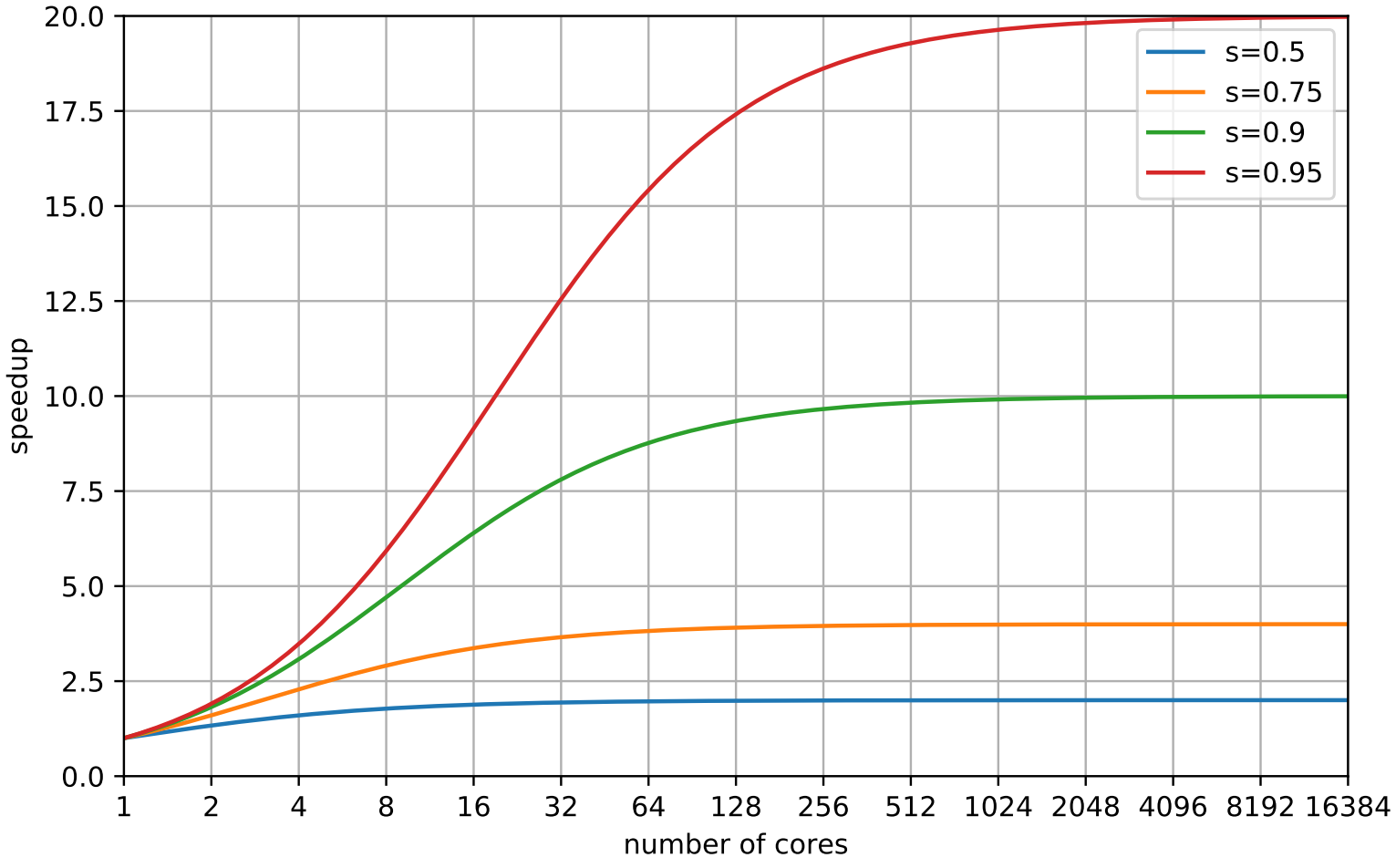
5 GB/s (100 $\times$ 1MB)
370 MB/s (1k $\times$ 100kB)
36 MB/s (10k $\times$ 10kB)

Recommendation: Combine small data transfers into bigger ones to hide the latency of each operation.

Massive parallelism

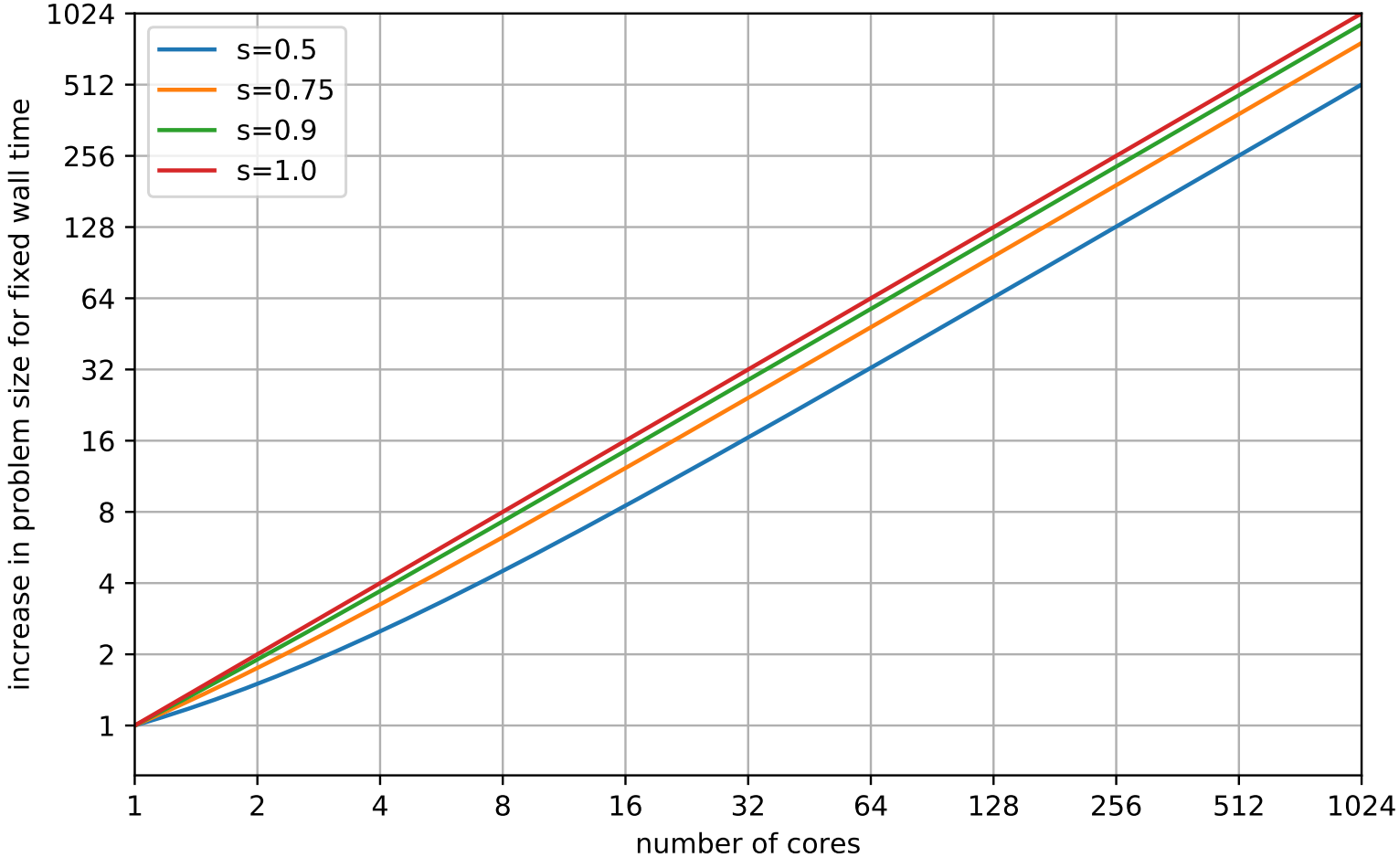
Amdahl's law

Let us assume the **sequential execution time is** T. The fraction of the program that is (perfectly) **parallelizable is denoted by** $0 \leq s \leq 1$.



Gustafson's law

Let us assume that the **total work of the sequential program is** W and takes time T to execute. The fraction of the program that is (perfectly) parallelizable is denoted by $0 \leq s \leq 1$.



Amdahl vs Gustafson

Amdahl is the pessimist.

- Even a small sequential part severely limits the speedup.
- Assumes the problem size is constant.
- Amdahl's law talks about **strong scaling** (the more difficult problem).

Gustafson is the optimist.

- An arbitrary increase in the work for a given time can be achieved for any s.
- Gustafson's law is about **weak scaling** (the easier problem).

Summary

Amdahl and Gustafson's law hold for any parallel system.

- They are valid on the CPU as well as on the GPU.

But the **amount of parallelism required to fully exploit GPUs is significantly larger**.

- 160 cores (5120 CUDA cores) vs 32 cores.

GPUs usually require a larger problem size to be effective.

Number of threads & number of blocks

Number of threads & number of blocks

To launch a kernel we have to specify

- number of threads per block (integer or dim3);
- number of blocks per grid (integer or dim3).

Using dim3 makes it easier to access 2d/3d structures.

All threads in a block are executed on the same streaming multiprocessor.

- These threads share L1 cache/shared memory.

On the V100 each of the (80) SM can process 64 threads in parallel.

- But we can launch up to 1024 threads per block.
- Each SM can host 2048 threads.

Oversubscription

Oversubscription is launching more threads than the hardware can support.

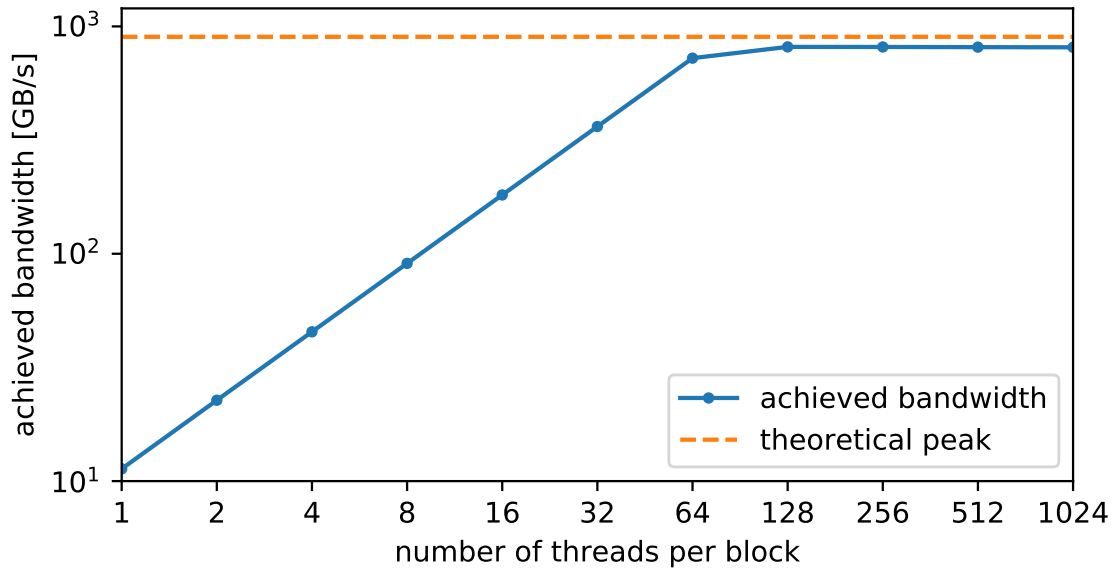
Oversubscription is a big no-go on CPU hardware.

GPUs are very fast at switching out threads.

- **Oversubscription as a way to hide latency.**

Oversubscription

```
__global__ void k_mult(double* x, double* y, int n) {
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    if(i < n) y[i] = 3.0*x[i];
}
k_copy<<<n/threads_per_block, threads_per_block>>>(x, y, n);
```



Recommendation: Use at least 128 threads per block.

Recommendation: Some problems require more parallelism than CUDA cores to obtain optimal performance.

Branches and SIMD

Avoid branch divergence

There is no explicit vectorization in CUDA, but **warps are still executed in lockstep**.

- Each 32 threads in a block from a warp.

The code

```
if(threadIdx.x % 2 == 0)
    // do something
else
    // do something else
```

is ececuted as follows

```
// do something (all thrads for which threadIdx.x % 2 == 0)
// do something (all threads for which threadIdx.x % 2 == 1)
```

Branches that diverge within a warp are serialized.

- Essentially SIMD behavior.

Measure performance

Measure performance

Modern hardware is complicated.

- Measuring performance is often key.

In principle you can use your CPU timer, but there are dragons.

WRONG!

```
double a = clock();
kernel<<<num_blocks,num_threads>>>(...);
cout << (clock()-a)/CLOCKS_PER_SEC << " s" << endl;
```

Kernel launches are asynchronous. Stops the time until control is returned to the CPU.

Correct.

```
double a = clock();
kernel<<<num_blocks,num_threads>>>(...);
cudaDeviceSynchronize();
cout << (clock()-a)/CLOCKS_PER_SEC << " s" << endl;
```

Note: `clock` reports the CPU time **not** the wall time.

Measure performance

CUDA provides facilities to measure time on the GPU.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);
// do stuff on the GPU
cudaEventRecord(stop, 0);
// kernel might still be running

cudaEventSynchronize(stop);
float time;
cudaEventElapsedTime(&time, start, stop);
cout << time*1e-3 << " s" << endl;

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

Exercise

The provided program (`exercise-efficiency.cu`) solves the heat equation.

The main part of the code is repeated matrix-vector multiplication.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_2 & a_1 & a_2 & 0 & 0 & \dots & 0 \\ 0 & b_2 & b_1 & b_2 & 0 & \dots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & 0 & 0 & c_3 & c_1 & c_2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ u_{n-1} \end{bmatrix}$$

Note: matrix is stored in memory as $[a_1, a_2, a_2, b_1, b_2, b_2, ...]$.

Exercise

There are a number of performance problems. Not all of them equally consequential.

- Compare the performance of the original code to the theoretical peak performance of the algorithm.
- Find the (many) performance issues and fix them.
- Time the different parts of your code to see where the bottleneck is.

Check that your code still produces the correct result.

Solution

Do not call `cudaMemcpy` for each row of the matrix.

- Create the matrix on the CPU and then copy it to the GPU with one call to `cudaMemcpy`.
- Or better, create the matrix directly on the GPU.
- Or even better, eliminate the matrix altogether.

Number of threads per block is too low (only 64).

Interchange pointers instead of copying `d_out` to `d_in`.

- cudaMemcpy call is superfluous.

Solution

Initialize the vector on the GPU

- No need to initialize the vector on the CPU and copy it to the GPU.
- GPU is better at computing sin than the CPU.
- Relevant in practice?

Problem is still too small to obtain peak performance.

- Increase the problem size (n).