Introduction to GPUs in HPC
CSCS Summer School 2020

Ben Cumming, CSCS

July 8, 2020

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Introduction

Course Overview

Over these two days we will cover a range of topics:

- Learn about the GPU memory model;
- Implement parallel CUDA kernels for simple linear algebra;
- Learn how to scale our parallel kernels to utilize all resources on the GPU;
- Learn about thread cooperation and synchronization;
- Learn about concurrent task-based parallelism;
- Learn how to profile GPU applications.
- Port the miniapp to the GPU.

Course Overview

We focus on HPC and modern GPU architectures, specifically:

- HPC development for P100 GPUs on Piz Daint;
- Using CUDA toolkit version 10 and above;
- Some are available on P100 and later GPUs.
  - e.g. double precision atomics.
- Likewise, we won't be covering some features that are available on the latest "Volta" GPUs.

Course Overview

There aren't many prerequisites for the course:

- No GPU or graphics experience required.
- I assume C++11 knowledge.
- The generic GPU programming concepts from CUDA are useful for when:
  - Developing with OpenACC, OpenCL and GPU-ready libraries.
  - Using ML frameworks that use GPU for compute.

Why GPUs?

There is a trend towards more parallelism "on node"

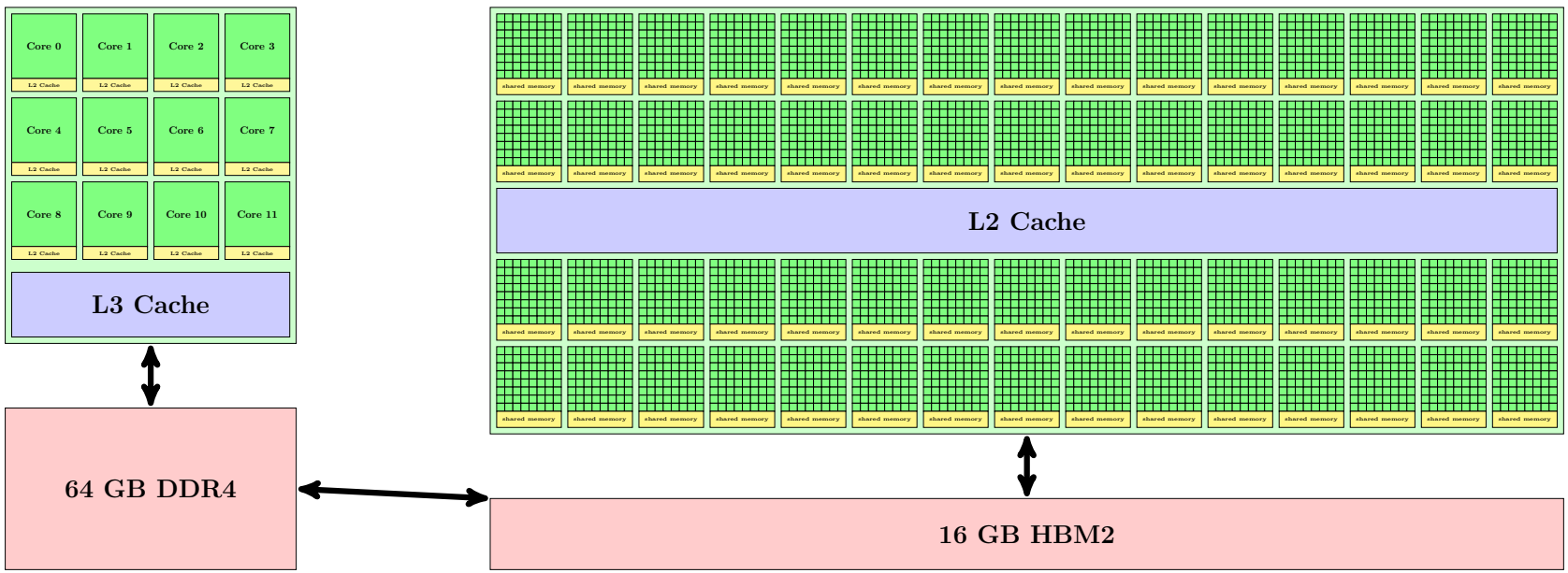
Multi-core CPUs get more cores and wider vector lanes:

- 24-core×SMT 4×SIMD 128: IBM Power9 (2017).
- 28-core×SMT 2×SIMD 512: Intel Xeon (2020);
- 48-core×SMT 1×SIMD 512: Fujitsu ARM A64FX (2020).

Many-core Accelerators with many highly-specialized cores and high-bandwidth memory:

- NVIDIA P100 GPUs with 3584 cores (2016);
- NVIDIA V100 GPUs with 5120 cores (2017);
- NVIDIA A100 GPUs with 8192 cores (2020).

A Piz Daint node



. . . that is a lot of parallelism!

MPI and the free lunch

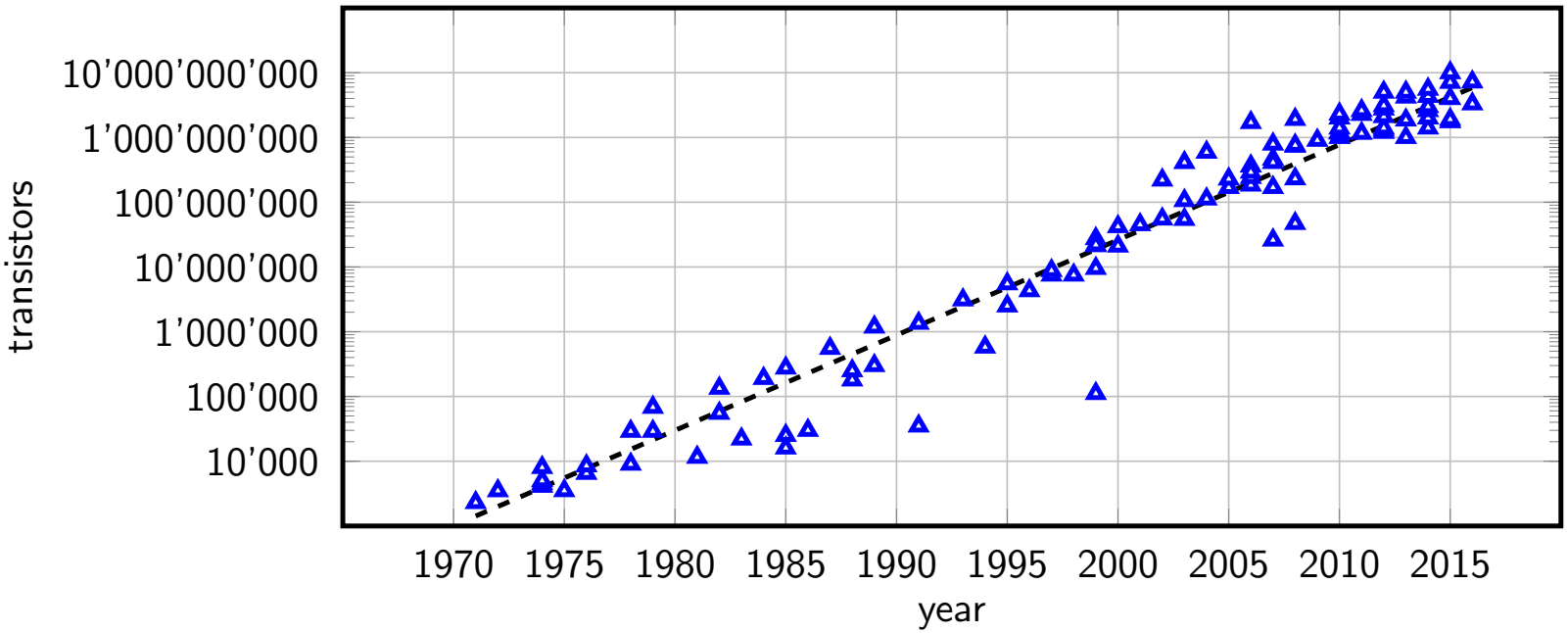
HPC applications were ported to use the message passing library MPI in the late 90s and early 2000s at great cost and effort

- Individual nodes with one or two CPUs
- Break problem into chunks/sub-domains
- Explicit message passing between sub-domains

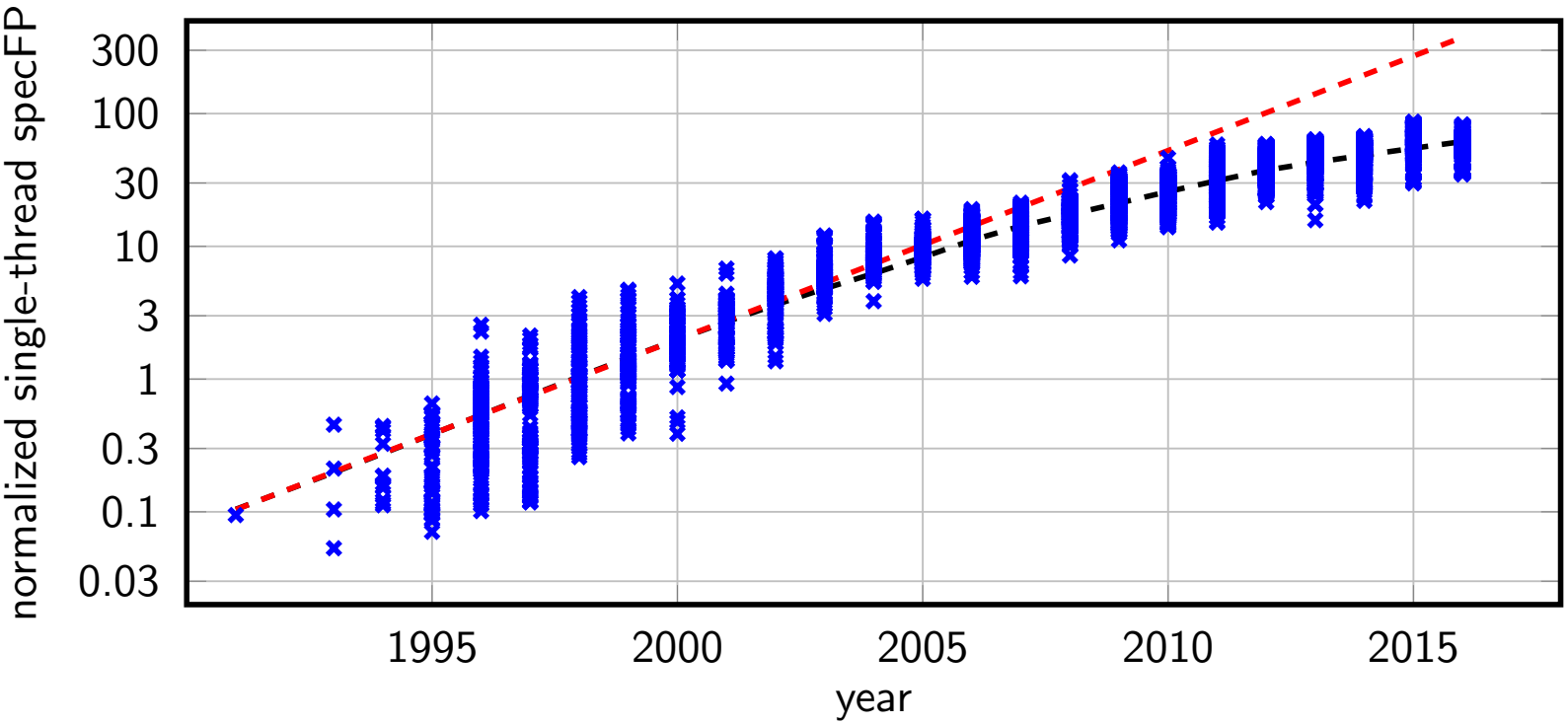
The "free lunch" was the regular speedup in codes as CPU clock frequencies increased and as the number of nodes in systems increased

- With little/no effort, each new generation of processor bought significant speedups.

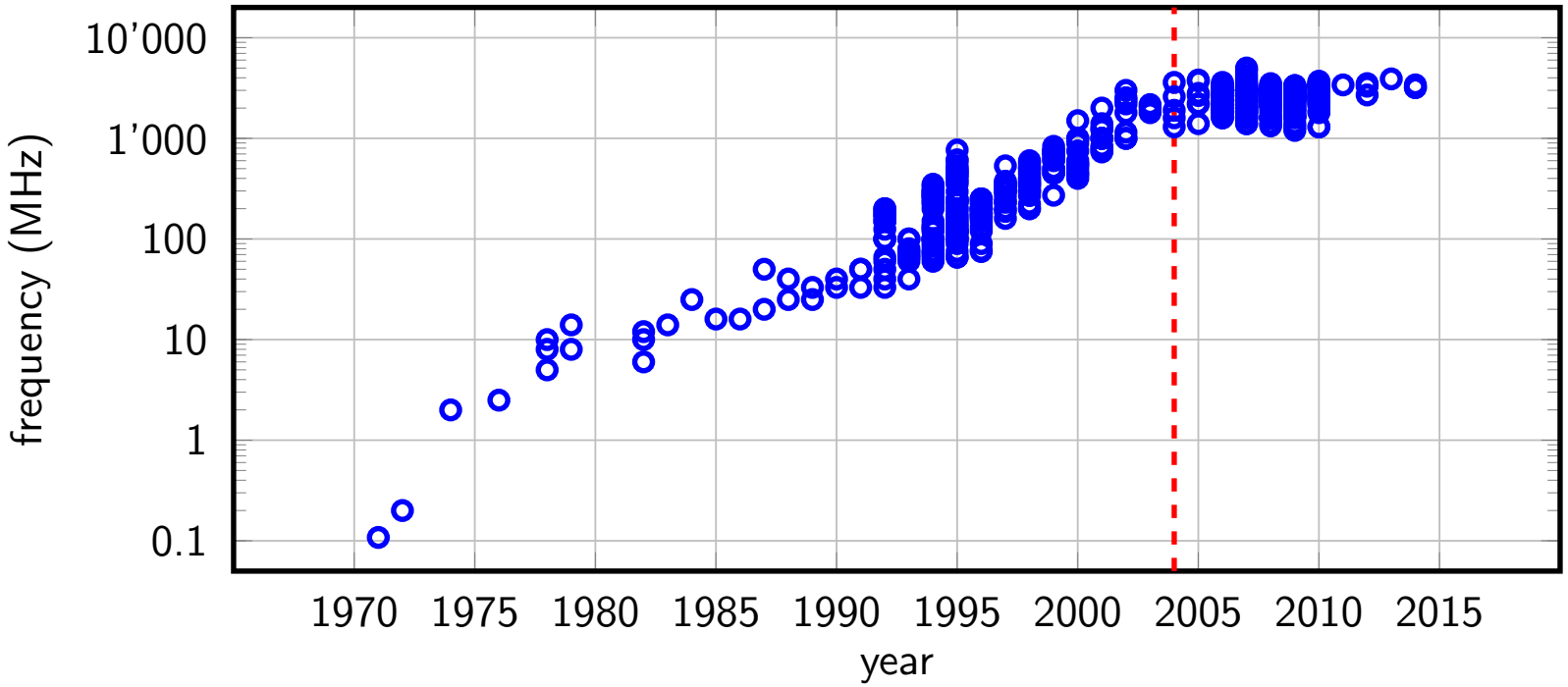
. . . but there is no such thing as a free lunch.



The number of transistors in processors has increased exponentially for 45 years.



Floating point performance per core is not keeping up ...



Clock speeds peaked around 2005.

The problem: power $\propto$ frequency^3

How to speed up an application

There are 3 ways to increase performance:

1. Increase clock speed.
2. Increase the number of operations per clock cycle:
   - vectorization;
   - instruction level parallelelism;
   - more cores.
3. Don't stall:
   - e.g. cache to avoid waiting on memory requests;
   - e.g. branch prediction to avoid pipeline stalls.

Clock frequency won't increase

In fact, clock frequencies have been going down as the number of cores increases:

- A 4-core Haswell processor at 3.5 GHz (4*3.5=14 Gops/second) has the same power consumption as a 12-core Haswell at 2.6 GHz (12*2.6=31 Gops/second);
- A P100 GPU with 3584 CUDA cores runs at 1.1 GHz.

Caveat

It is not reasonable to compare a CUDA core and an X86 core.

Parallelism will increase

- The number of cores in both CPUs and accelerators will continue to increase
- The width of vector lanes in CPUs will increase
  - 8×SIMD double for AVX512 and SVE (Intel and ARM).
- The number of threads per core will increase
  - Intel SkyLake: 2 threads/core
  - Intel KNL: 4 threads/core
  - IBM Power-8: 8 threads/core

Memory is slow

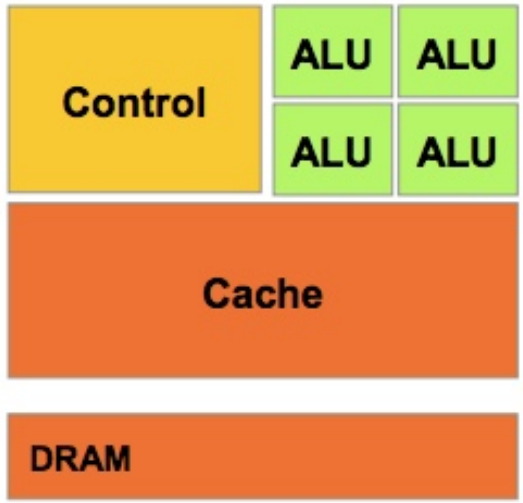
Memory is much slower than processors

- For both CPU and GPU the latency of fetching a cache-line from memory is 100s of cycles...
- ...100s of cycles that the processor is stalled
- Latency has to be hidden or reduced to minimise stalling

Low Latency or High Throughput?

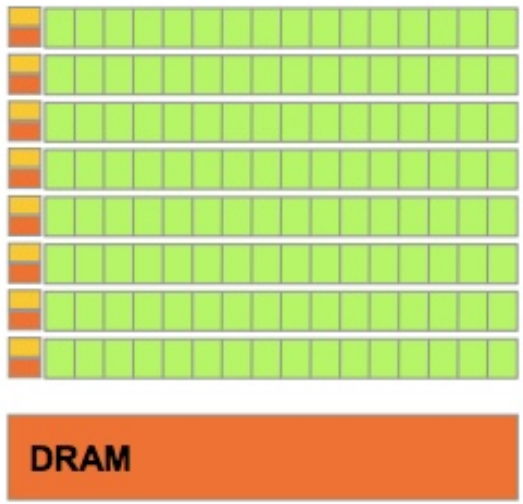
CPU

- Optimized for low-latency access to cached data sets.
- Control logic for out-of-order and speculative execution.



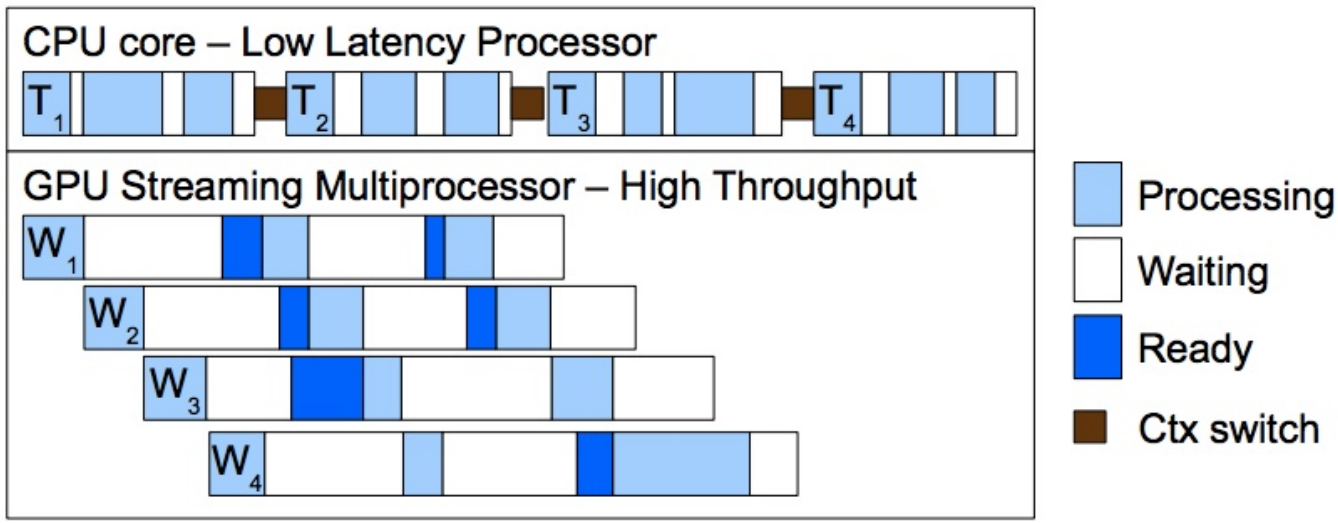
GPU

- Optimized for data-parallel, throughput computation.
- Architecture tolerant of memory latency.
- More transistors dedicated to computation.



GPUs are throughput devices

- CPU cores are optimized to minimize latency between operations.
- GPUs aim to minimize latency between operations by scheduling multiple warps (thread bundles).



Many applications aren't designed for many core

- Exposing sufficient fine-grained parallelism for multi and many core processors is hard.
- New programming models are required.
- New algorithms are required.
- Existing code has to be rewritten or refactored.

On-node parallelism will continue to increase:

- Piz Daint @ CSCS (2015): 1 GPU + 1 CPU.
- Marconi100 @ CINECA (2020): 4 GPU + 2 CPU.
- EUROHPC pre-exascale (2021): 4 GPU + 1 CPU.
- US ECP exascale (2021-2023): 4 GPU + 1 CPU.

TLDR: Change because power

Writing good concurrent code for many-core is difficult

- But the days of easy speed up each generation of CPU are over
  - Performance gains must not increase power consumption
- This course will be about one type of many-core architecture NVIDIA GPUs
  - CUDA is GPU-specific.
  - Conceptually very close to HIP, OpenCL and SYCL (AMD/NVIDIA/Intel).

Introduction to GPUs in HPC

Ben Cumming, CSCS
July 9, 2020

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Using GPUs in Your Application

Rule #1: **don't** develop your own GPU code!

Libraries

There are many open libraries for GPUs.

- cuBLAS: Dense linear algebra primitives.
- Thrust: C++ STL-like algorithms and containers.
- cuRAND and Random123: Random numbers.
- cuFFT: FFT
- Kokkos: Generic performance portable parallel motifs.

. . . And many more!

Take some time to investigate what is available before starting.

You are going to write your own code?

Directives

- OpenACC and OpenMP define **directives** that can be used to instruct the compiler how to generate GPU code.
- In theory the easiest path for porting.

GPU-specific Languages

- Languages designed for GPU programming.
- Maximum flexibility and performance.
- For example: CUDA, OpenCL and SYCL.

Things to consider

Before starting on a GPU implementation, it pays to ask some questions and do some preliminary exploration:

1. Is my program computationally or bandwidth intensive?
2. Does it have enough parallel work to utilize the GPU?
3. Must I change algorithms to expose enough parallelelism?
4. Are there serial bottlenecks that will limit scaling?
5. Is the pain worth the gain?

- Questions 1, 2 and 3 will be discussed in this course.
- Question 4 will be considered briefly here.
- Questions 5 requires answers for 1–4.

Limitations to parallel speedup

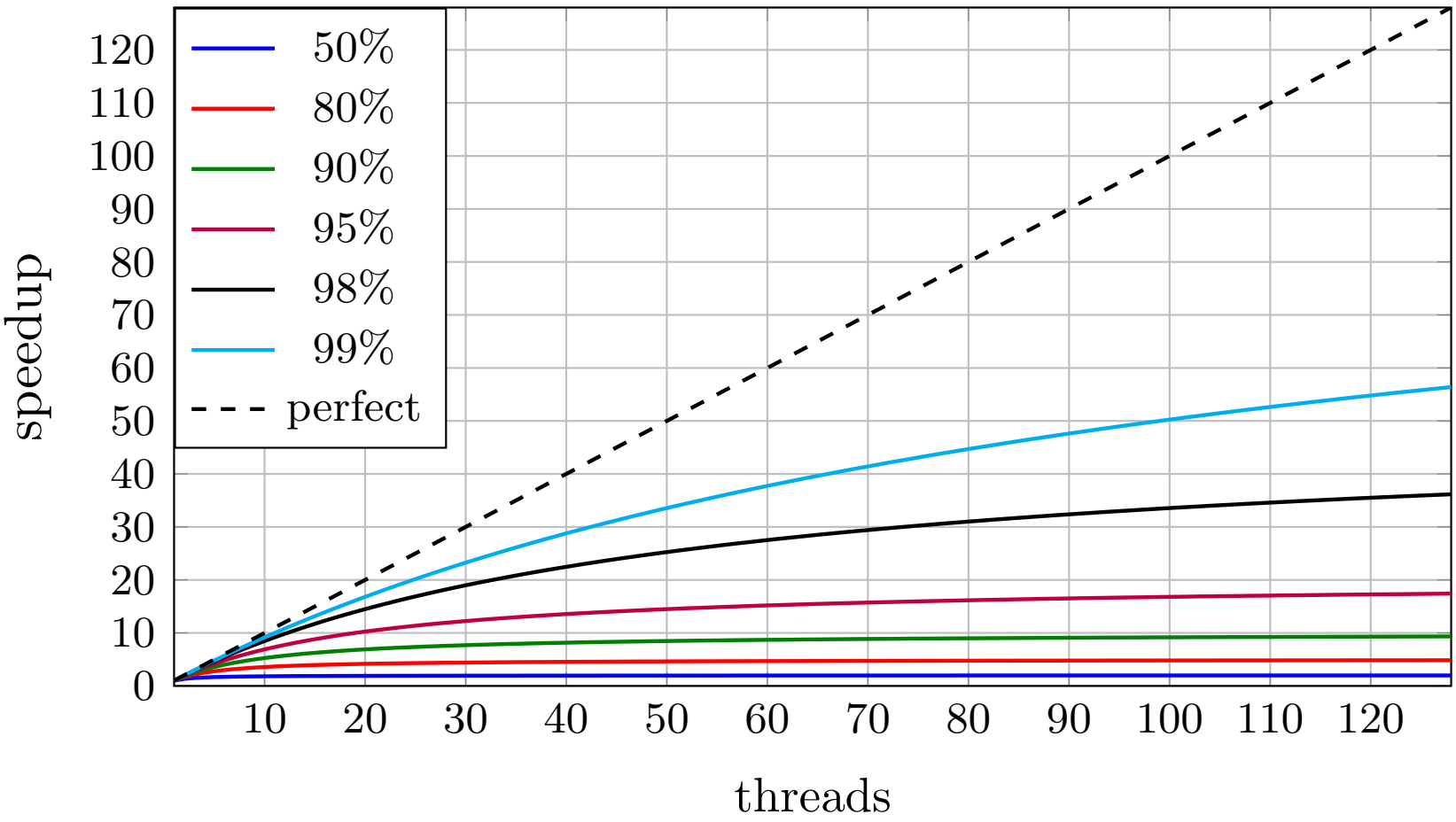
- Parallel speedup is limited by **the proportion of serial work** in your code.
- **Amdahl's law** defines the **maximum possible speedup** when only parts of the code can be parallelised

$$t_n = t_1 \left(p + \frac{(1-p)}{n} \right),$$

where t_n is time to solution for n threads and $p \in [0, 1]$ is the proportion of sequential code.

- The limit on time to solution is $\lim_{n \to \infty} = pt_1$
  - e.g. 1% of serial code gives a maximum 100× speedup.

Amdahl illustrated



CUDA

CUDA is a **parallel computing platform and API**

- For CUDA-enabled Nvidia GPUs.

We use CUDA as short hand for CUDA C/C++ and API

- CUDA C++ is a **superset** of C++
- Adds keywords for writing kernels to run on the GPU.
- Adds syntax for launching kernels on the GPU.

The CUDA toolkit is more than a programming language:

- Runtime API for managing GPU resources and execution.
- Tools including profilers and debuggers.

Compiling CUDA

CUDA code is compiled with the **nvcc** compiler driver

- source files have `.cu` extension
- headers have `.h`, `.hpp`, `.hcu` extension.

CUDA compilation involves multiple splitting, compilation, preprocessing and merging steps

- nvcc hides this complexity from the user.
- It closely mimics the interface of the GNU compiler.
- Behind the scenes it:
  - uses GCC to compile the code that runs on CPU;
  - and compiles the GPU code separately.

Compiling CUDA with Clang

Clang now supports compilation of CUDA code, targetting NVIDIA GPUs.

- Performance of the generated code is on par with nvcc.
- It is a good idea to test your CUDA code with both Clang and nvcc.
- The most recent version of the Cray C++ compiler on Daint is Clang based.

Compiling CUDA

Example CUDA compilation

```
> nvcc -arch=sm_60 -lineinfo -O2 -std=c++11 -g -o foo foo.cu
```

Some flags are for **device** code generation:

- `-arch=sm_60` target GPU architecture (Pascal)
- `-lineinfo` debug information for device code.

Some are for **host**:

- `-g` debug information for host code.

And some are for both **host and device**:

- `-O2` optimization level
- `-std=c++11` target language
- `-o foo` name of executable.

Compiling CUDA with Clang

Compilation with Clang uses different gpu-specific options:

```
> CC -xcuda --cuda-gpu-arch=sm_60 --cuda-path=$CUDA_ROOT
          -O2 -std=c++11 -g -o foo foo.cu
```

Where `cc` is the Cray compiler wrapper on Daint.

Exercise: Getting Started on Piz Daint

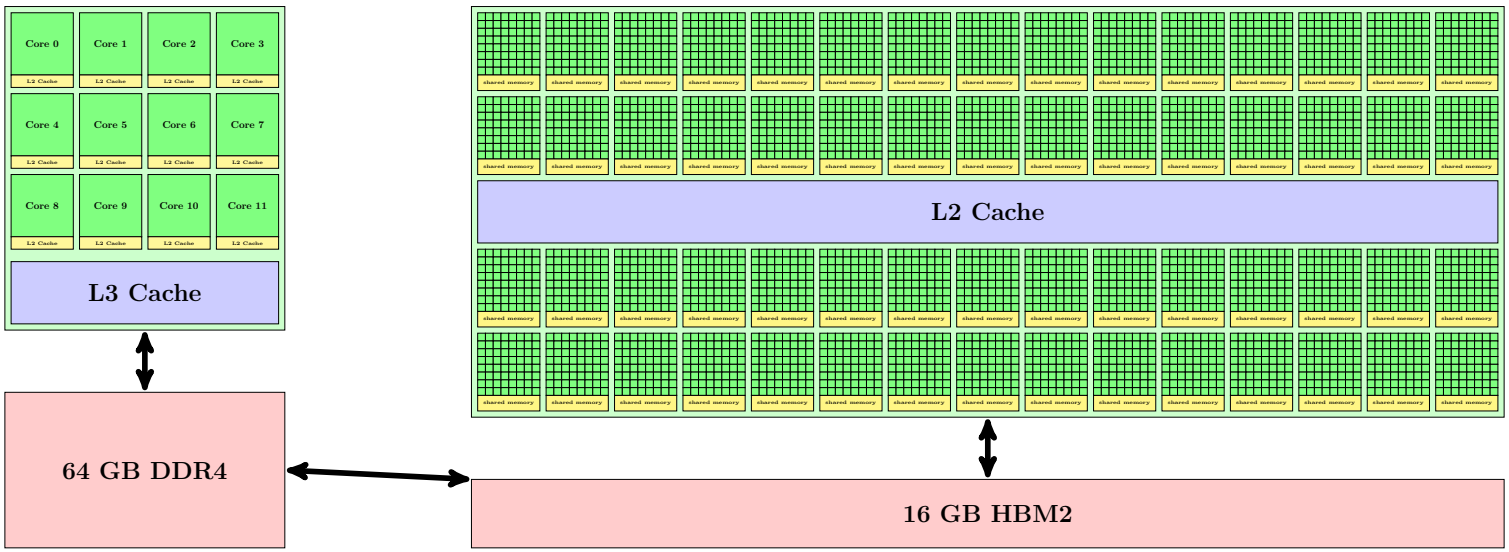
In this exercise we will get introduced to Daint and make sure that everybody is set up.

```
# log on to daint with your course username & password
> ssh -X <your account name>@daint

# get one node on the course reservation for 60 minutes
> salloc -Cgpu --reservation=summer_school -t60

# go to scratch and get the course material
> cd $SCRATCH/SummerSchool2020
> git pull

# compile and test the demo
> cd topics/cuda/practicals/demos
> cat hello.cu
> module load gcc cudatoolkit
> nvcc -arch=sm_60 hello.cu -o hello
> srun ./hello
```

Working with GPU memory

Ben Cumming, CSCS
July 9, 2020

Memory on a Piz Daint Node



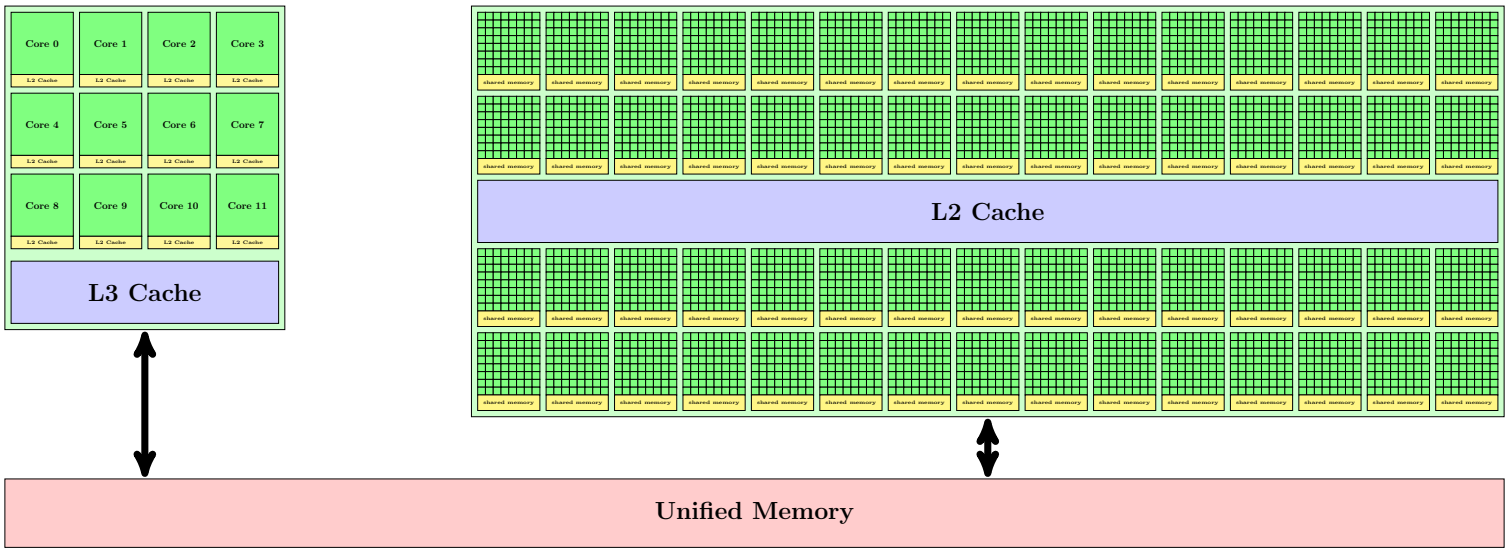
Host and Device Memory Spaces

- The GPU has separate memory to the host CPU
  - The host CPU has 64 GB of DDR4 **host memory**
  - The P100 GPU has 16 GB of HBM2 **device memory**
- Kernels executing on the GPU only have fast access to device memory
  - Kernel accesses to host memory are copied to GPU memory first over the (slow) PCIe connection.

host ↔ device	11×2 GB/s	PCIe gen3
host memory	45 GB/s	DDR4
device memory	558 GB/s	HBM2

- **Optimization tip**: The massive bandwidth of HBM2 on P100 GPUs can only help if data is in the right memory space **before** computation starts.

Unified Memory



CUDA unified memory presents a single memory space that can be accessed by both host and GPU code

Unified Memory

Unified memory presents a single memory space.

- Both CPU and GPU can access the same memory.
- First introduced with CUDA 6 and Keplar.
- Improved with CUDA 8 and Pascal:
  - All host and device memory can be addressed
  - The **page migration** engine transfers data between GPU and CPU memory as needed
  - API provides fine-grained control of page migration
- Simplifies memory management for GPU programming

Managed memory is useful for porting to the GPU.

- Not suitable as the default choice of memory management.
- Can lead to negative performance and subtle bugs.
- Host-device memory coherency will improve in future GPUs.

Accessing Memory

CUDA uses C pointers to address memory:

```
double* data = //address to either host, device or managed memory
```

- A pointer can hold an address in
  - **either** device or host memory
  - managed memory that can **migrate** between host and device
- The **CUDA runtime library** provides functions that can be used to allocate, free and copy managed and device memory.

Managing Managed Memory

Allocating managed memory

```
cudaMallocManaged(void** ptr, size_t size, unsigned flags)
```

- `size` number of **bytes** to allocate.
- `ptr` points to allocated memory on return.
- `flags` by default is set to `cudaMemAttachGlobal`.

Freeing managed memory

```
cudaFree(void* ptr)
```

Allocate memory for 100 doubles in managed memory

```
double* v;
auto bytes = 100*sizeof(double);
cudaMallocManaged(&v, bytes); // allocate memory
cudaFree(v);                  // free memory
```

Exercise: Getting started

We have to set up the environment before compiling.

```
> module load daint-gpu
> module swap PrgEnv-cray PrgEnv-gnu
> module load cudatoolkit
> gcc --version # nvcc uses gcc:
gcc (GCC) 8.3.0 20190222 (Cray Inc.)
...
> nvcc --version
...
Cuda compilation tools, release 10.1, V10.1.105
```

Exercise: Managed Memory Example

1. open the files `api/managed.cu` and `api/util.hpp`.
2. what does `managed.cu` do?
   - you can use Google!
3. run it with 20 and 22

```
> cd topics/cuda/practicals/api
> make
> srun ./managed 20
```

4. does it work?
5. run the cuda profiler

```
> srun nvprof -o managed.nvvp --profile-from-start off -f
    ./managed 25
> nvvp managed.nvvp &
```

Concurrent Host-Device Memory Access

The CPU:

- launches the GPU code `gpu_call`
- executes CPU function `cpu_call`

The GPU:

- executes gpu code asynchronously

The problem:

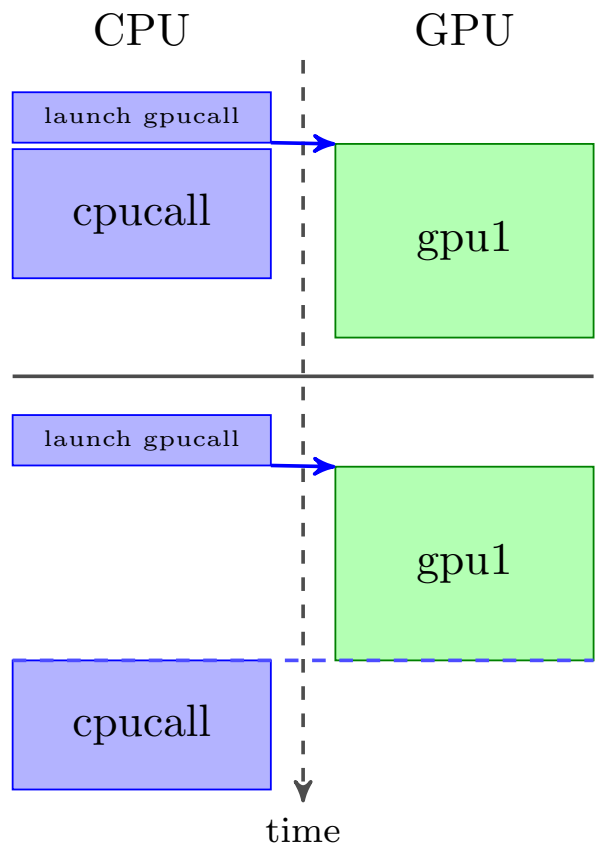
- both `cpu_call` and `gpu_call` may try to access the same memory

The solution:

- synchronize calls between host and device

application code

```
gpu_call<<<...>>>();
cpu_call();
```



Concurrent Host-Device Memory Access

The CPU can't access managed memory at the same time a GPU kernel is accessing it:

- Doing so causes a segmentation faults or undefined behavior.
- To test for synchronization issues run with an environment variable

```
> export CUDA_LAUNCH_BLOCKING=1
```

- The CUDA API function `cudaDeviceSynchronize` can be used to force synchronization.

```
gpu_call<<<...>>>();
cudaDeviceSynchronize();
cpu_call();
```

Exercise: Managed Memory Debugging

1. Test if concurrent host-device memory access caused the incorrect results in the `api/managed.cu` example.
2. Can you fix the issue by adding one `cudaDeviceSynchronize()` call?
3. does the profile from nvprof look different?

Allocating Device Memory

It is possible to allocate host and device memory directly

- Explicitly allocate memory on device.
  - can't be read from host.
- Manually copy data to and from host.
- For memory that should always reside on device.
- The programmer can optimize memory transfers by hand.
  - with effort, you can get the best performance this way.

Allocating device memory

```
cudaMalloc(void** ptr, size_t size)
```

- `size` number of **bytes** to allocate
- `ptr` points to allocated memory on return

Freeing device memory

```
cudaFree(void* ptr)
```

Allocate memory for 100 doubles on device

```
double* v; // C pointer that will point to device memory
auto bytes = 100*sizeof(double); // size in bytes!
cudaMalloc(&v, bytes); // allocate memory
cudaFree(v);           // free memory
```

Perform blocking copy (host waits for copy to finish)

```
cudaMemcpy(void *dst, void *src, size_t size, cudaMemcpyKind kind)
```

- `dst` destination pointer
- `src` source pointer
- `size` number of **bytes** to copy to `dst`
- `kind` enumerated type specifying **direction** of copy: one of `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, `cudaMemcpyHostToHost`

Copy 100 doubles to device, then back to host

```
auto size = 100*sizeof(double); // size in bytes
double *v_d;
cudaMalloc(&v_d, size);                  // allocate on device
double *v_h = (double*)malloc(size); // allocate on host
cudaMemcpy(v_d, v_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(v_h, v_d, size, cudaMemcpyDeviceToHost);
```

Errors happen...

All API functions return error codes that indicate either:

- success;
- an error in the API call;
- an error in an earlier asynchronous call.

The return value is the enum type `cudaError_t`

- e.g. `cudaError_t status = cudaMalloc(&v, 100);`
  - status is {`cudaSuccess`, `cudaErrorMemoryAllocation`}

Handling errors

```
const char* cudaGetErrorString(status)
```

- returns a string describing status

```
cudaError_t cudaGetLastError()
```

- returns the last error
- resets status to `cudaSuccess`

Copy 100 doubles to device **with error checking**

```
double *v_d;
auto size = sizeof(double)*100;
double *v_host = (double*)malloc(size);
cudaError_t status;

status = cudaMalloc(&v_d, size);
if(status != cudaSuccess) {
  printf("cuda error : %s\n", cudaGetErrorString(status));
  exit(1);
}

status = cudaMemcpy(v_d, v_h, size, cudaMemcpyHostToDevice);
if(status != cudaSuccess) {
  printf("cuda error : %s\n", cudaGetErrorString(status));
  exit(1);
}
```

It is essential to test for errors

But it is tedious and obfuscates our source code if it is done in line for every API and kernel call...

Exercise: Device Memory API

Open `topics/cuda/practicals/api/util.hpp`

1. what does `cuda_check_status()` do?
2. look at the template wrappers `malloc_host` & `malloc_device`
   - what do they do?
   - what are the benefits over using `cudaMalloc` and `free` directly?
   - do we need corresponding functions for `cudaFree` and `free`?
3. write a wrapper around `cudaMemcpy` for copying data `host`→`device` & `device`→`host`
   - remember to check for errors!
4. compile the test and run
   - it will pass with no errors on success

```
> make explicit
> srun ./explicit 8
```

Exercise: Device Memory API

1. How does performance compare with the managed memory version?
2. What does the nvprof profile look like?
   - contrast with managed memory profile.

```
> srun nvprof -o explicit.nvvp --profile-from-start off -f
    ./explicit 25
> nvvp explicit.nvvp &
```

Writing GPU Kernels

Ben Cumming, CSCS
July 8, 2020

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Going Parallel : Kernels and Threads

Threads and Kernels

- **Threads** are streams of execution, run simultaneously on GPU.
- A **kernel** is the function run by each thread.
- CUDA provides language support for:
  - writing kernels;
  - launching many threads to execute a kernel in parallel.
- CUDA hides the low-level details of launching threads.

The process for developing CUDA kernels

1. Formulate algorithm in terms of parallel work items.
2. Write a kernel implementing a work item on one thread.
3. Launch the kernel with the required number of threads.

Scaled Vector Addition (axpy)

We have used CUBLAS to perform scaled vector addition:

$$\mathbf{y} = \mathbf{y} + \alpha\mathbf{x}$$

- $\mathbf{x}$ and $\mathbf{y}$ are vectors of length n; $x, y \in \mathbb{R}^n$
- α is scalar. $\alpha \in \mathbb{R}$

Applying `axpy` requires n operations:

$$y_i \leftarrow y_i + a * x_i, \quad i = 0, 1, \ldots, n-1$$

which can be performed **independently** and **in any order**.

`axpy` implemented on CPU with a loop

```
void axpy(double *y, const double *x, double a, int n) {
  for(int i=0; i<n; ++i)
    y[i] = y[i] + a*x[i];
}
```

Kernels

A **kernel** defines the work item for a single thread

- The work is performed by many threads executing the same kernel **simultaneously**.
- Conceptually corresponds to the inner part of a loop for BLAS1 operations like `axpy`.

host : add two vectors

```
void add_cpu(int *a, int *b, int n){
  for(auto i=0; i<n; ++i)
    a[i] = a[i] + b[i];
}
```

CUDA : add two vectors

```
__global__
void add_gpu(int *a, int *b, int n){
  auto i = threadIdx.x;
  a[i] = a[i] + b[i];
}
```

- `__global__` keyword indicates a kernel
- `threadIdx` used to find unique id of each thread

Launching a kernel

- Host code launches a kernel on the GPU **asynchronously**.
- CUDA provides the "triple chevron" `<<<_,_>>>` syntax for launching a kernel.

host : add two vectors

```
auto n = 1024;
auto a = host_malloc<int>(n);
auto b = host_malloc<int>(n);
add_cpu(a, b, n);
```

CUDA : add two vectors

```
auto n = 1024;
auto a = device_malloc<int>(n);
auto b = device_malloc<int>(n);
add_gpu<<<1,n>>>(a, b, n);
```

- `add_gpu<<<1, num_threads>>>(args... )` launches the kernel `add_gpu` with `num_threads` parallel threads.

Exercise: My First Kernel

Open `axpy/axpy.cu`

1. Write a kernel that implements `axpy` for `double`
   - `axpy_kernel(double *y, double *x, double a, int n)`
   - **extra**: can you write a C++ templated version for any type?
2. launch the kernel (look for `TODO`)
3. Compile the test and run
   - it will pass with no errors on success
   - first try with small vectors of size 8
   - try increasing launch size... what happens?
4. **extra**: can you extend the kernel to work for larger arrays?

Scaling Up : Thread Blocks

In the `axpy` exercises we were limited to 1024 threads for a kernel launch

- but we need to scale beyond 1024 threads for the **massive parallelism** we were promised!

Thread blocks and grids

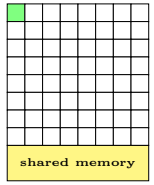
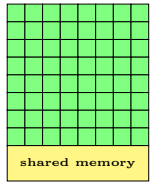
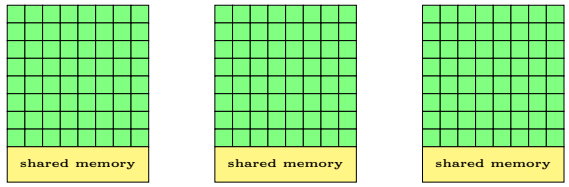
kernels are executed in groups of threads called **thread blocks**

- the launch configuration `axpy<<<grid_dim, block_dim>>>(...)`
  - launch a **grid** of `grid_dim` **blocks**
  - each **block** has `block_dim` **threads**
  - for a total of `grid_dim` × `block_dim` threads
- previously we launched just one thread block
  `axpy<<<1, n>>>(...)`

Why the additional complexity?

Coordination between threads doesn't scale:

- Threads in a block can synchronize and share resources
- This does not scale past a certain number of cores/threads
- EACH P100 GPU **streaming multiprocessor** (SMX) has 64 CUDA cores, and can run 2028 threads
- Threads in a block run on the same SMX, with shared resources and thread cooperation
- Work is broken into blocks, which are distributed over the 56 SMXs on the GPU.

concept	hardware	
thread		■ each thread executed on one core
block		■ block executed on 1 SMX ■ multiple blocks per SMX if sufficient resources ■ threads in a block share SMX resources
grid		■ kernel is executed in grid of blocks ■ blocks distributed over SMXs ■ multiple kernels can run at same time

Calculating thread indexes

A kernel has to calculate the index of its work item

- In `axpy` we used `threadIdx.x` for the index.
- With multiple blocks, we need more information, which is available in the following **magic variables**:

`gridDim`	:	total number of blocks in the grid
`blockDim`	:	number of threads in a thread block
`blockIdx`	:	index of block `[0, gridDim-1]`
`threadIdx`	:	index of thread in thread block `[0, blockDim-1]`

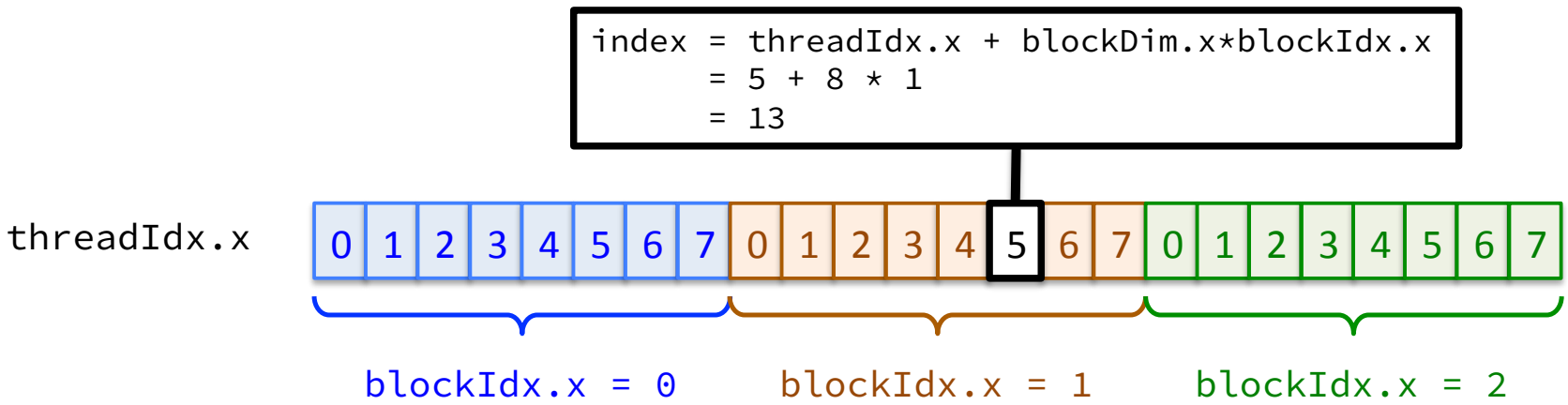
Calculating thread indexes

Consider accessing an array of length 24 with 8 threads per block. The **dimensions** of the kernel launch are:

- `blockDim.x == 8` (8 threads/block)
- `gridDim.x == 3` (3 blocks)

We calculate the index for our thread using the formula

```
auto index = threadIdx.x + blockIdx.x*blockDim.x
```



Calculating grid dimensions

The number of thread blocks and the number of threads per block are parameters for the kernel launch:

```
kernel<<<blocks, threads_per_block>>>(...)
```

Remember to guard against overflow when the number of work items is not divisible by the thread block size

vector addition with multiple blocks

```
__global__
void add_gpu(int *a, int *b, int n){
  auto i = threadIdx.x + blockIdx.x*blockDim.x;
  if(i<n) { // guard against access off end of arrays
    a[i] += b[i];
  }
}

// in main()
auto block_size = 512;
auto num_blocks = (n + (block_size-1)) / block_size;
add_gpu<<<num_blocks, block_size>>>(a, b, n);
```

Calculating grid dimensions

We have to take care when calculating the number of blocks in the grid, i.e. `blocks`:

```
kernel<<<blocks, threads_per_block>>>(...)
```

Most likely, the number of work items `n` is not a multiple of `threads_per_block`

- some threads in the last thread block will be idle.

Calculating grid dimensions

```
// in main()
auto block_size = 512;
auto num_blocks = (n + block_size-1) / block_size;
add_gpu<<<num_blocks, block_size>>>(a, b, n);
```

How many threads per block?

The number of threads per block has an impact on performance

- The optimal number depends on resources required by the kernel (registers, shared memory, computational intensity, etc).

The short answer is 64 or 128 on P100.

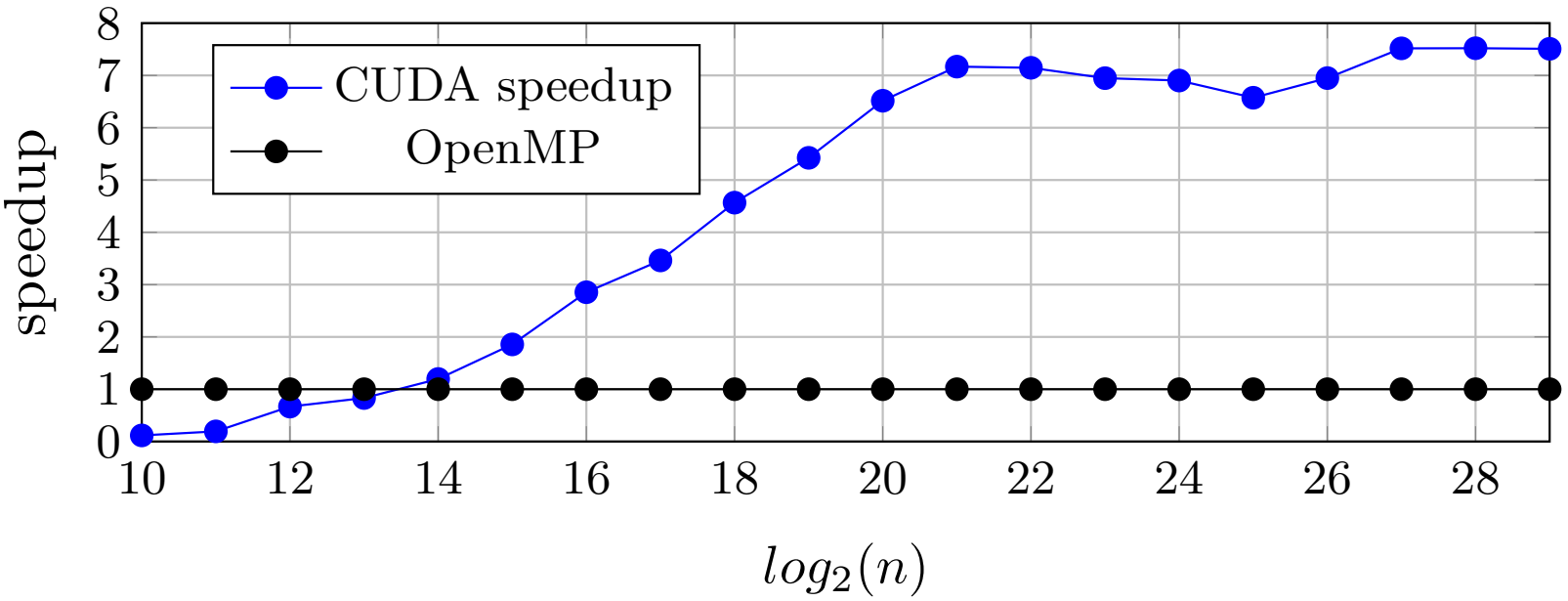
- For the main kernels in your application, perform experiments to find the ideal block size.

Exercise: Blocks

Open `axpy/axpy.cu` from the last exercise

1. Extend the **axpy** kernel for arbitrarily large input arrays (any `n`)
2. Update the call site to calculate the grid configuration
3. Compile the test and run
   - it will pass with no errors on success
4. Experiment with varying the size of the arrays (scaling)
   - start small and increase
5. finish the `newton.cu` example
   - how do the h2d, d2h and kernel timings compare?
6. **extra**: Compare scaling with the `axpy_omp` benchmark
7. **extra**: Experiment with varying the block size

Exercise: Results



The GPU is a throughput device:

- CUDA breaks even for $n \geq 2^{14} \approx 16,000$
- requires $2^{21} \approx 2,000,000$ to gain "full" $7\times$ speedup

You have to provide enough parallelism to exploit many cores

Writing GPU Kernels

Ben Cumming, CSCS
July 12, 2020

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Going Parallel: Working Together

Most algorithms do not lend themselves to trivial parallelization

reductions : e.g. dot product

```
int dot(int *x, int *y, int n){
  int sum = 0;
  for(auto i=0; i<n; ++i)
    sum += x[i]*y[i];
  return sum;
}
```

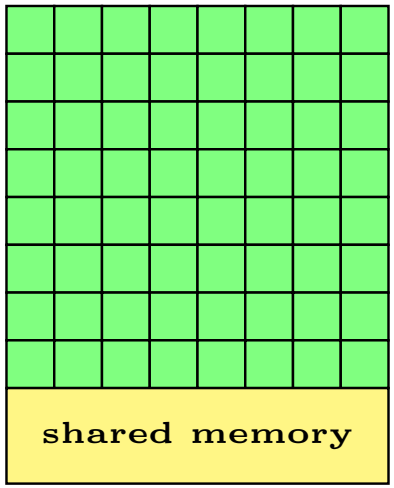
scan : e.g. prefix sum

```
void prefix_sum(int *x, int n){
  for(auto i=1; i<n; ++i)
    x[i] += x[i-1];
}
```

fusing pipelined stencil loops : e.g. apply blur kernel twice

```
void twice_blur(float *in, float *out, int n){
  float buff[n];
  for(auto i=1; i<n-1; ++i)
    buff[i] = 0.25f*(in[i-1]+in[i+1]+2.f*in[i]);
  for(auto i=2; i<n-2; ++i)
    out[i] = 0.25f*(buff[i-1]+buff[i+1]+2.f*buff[i]);
}
```

Block Level Synchronization



The P100 SMX has 64 KB of shared memory

CUDA provides mechanisms for **cooperation between threads in a thread block**.

- All threads in a block run on the same SMX
- Resources for synchronization are at SMX level
- No synchronization between threads in different blocks

CUDA also supports global **atomic operations** for coordination between threads

- We will cover this later...

Block Level Synchronization

Cooperation between threads requires sharing of data

- All threads in a block can share data using **shared memory**.
- Shared memory is **not visible** to threads in other thread blocks.
- All threads in a block are on the same SMX.
- There is 64 KB of shared memory on each SMX
  - one thread block can allocate 64 KB for itself
  - two thread blocks can allocate 32 KB each...
  - ...shared memory per thread block is a constraint on how many thread blocks can run simultaneously on an SMX.

1D blur kernel

A simple intensity preserving filter:

$$\text{out}_i \leftarrow 0.25 \times (\text{in}_{i-1} + 2 \times \text{in}_i + \text{in}_{i+1})$$

- Each output value is a linear combination of neighbours in input array
- First we look at naive implementation

Host implementation of blur kernel

```
void blur(double *in, double *out, int n){
  float buff[n];
  for(auto i=1; i<n-1; ++i)
    out[i] = 0.25*(in[i-1] + 2*in[i] + in[i+1]);
}
```

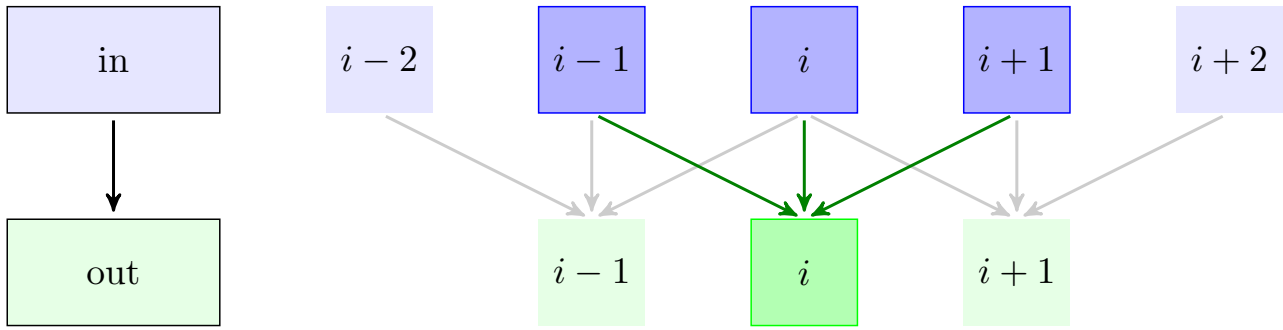
1D blur kernel on GPU

Our first CUDA implementation of the blur kernel has each thread load the three values required to form its output

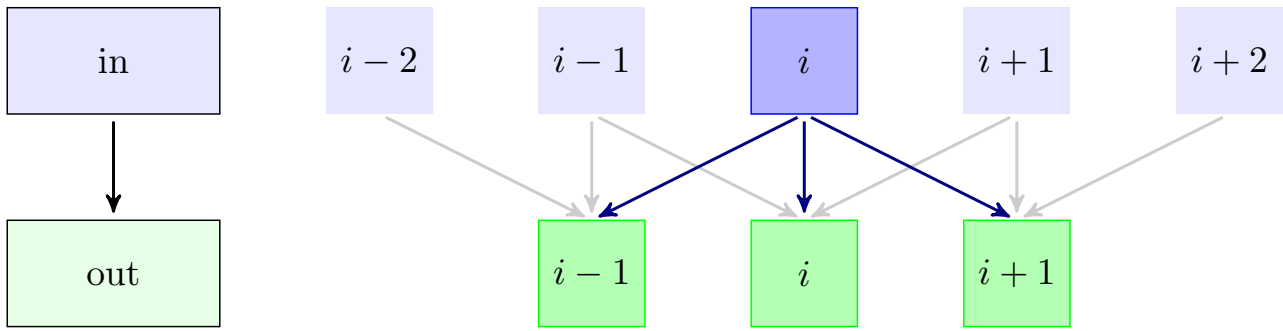
First implementation of blur kernel

```
__global__ void
blur(const double *in, double* out, int n) {
  int i = threadIdx.x + 1; // assume one thread block

  if(i<n-1) {
    out[i] = 0.25*(in[i-1] + 2*in[i] + in[i+1]);
  }
}
```

Each thread has to load 3 values from global memory to calculate its output

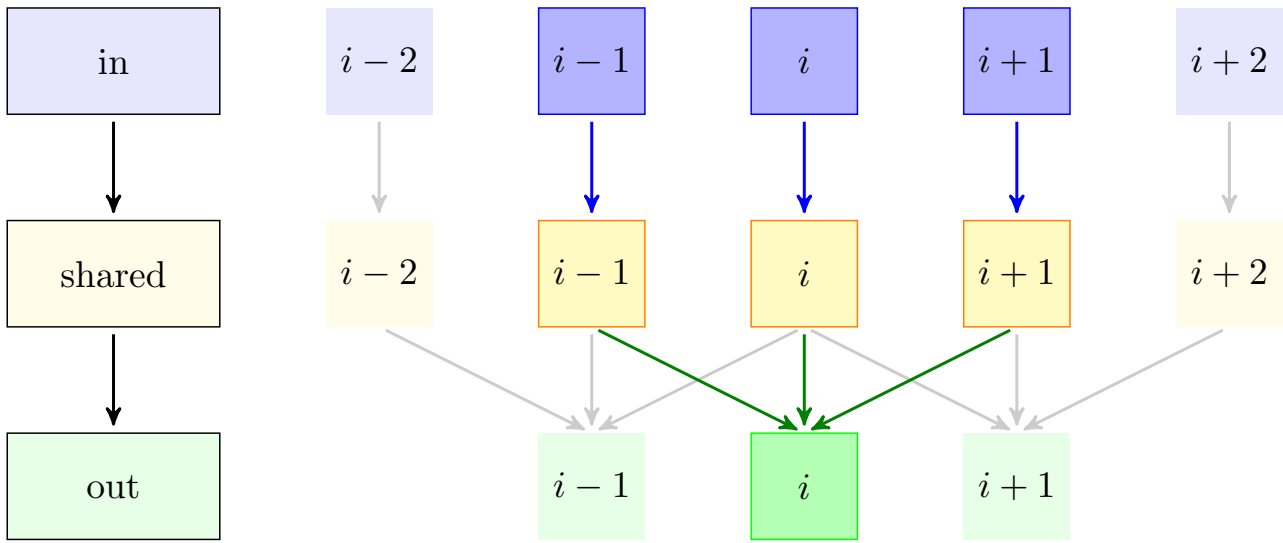


Alternatively, each value in the input array has to be loaded 3 times



To take advantage of shared memory the kernel is split into two stages:

1. Load `in[i]` into shared memory `buffer[i]`.
   - One thread has to load `in[0]` & `in[n]`.
2. Use values `buffer[i-1:i+1]` to compute kernel.



Blur kernel with shared memory

```
__global__
void blur_shared_block(double *in, double* out, int n) {
    extern __shared__ double buffer[];

    auto i = threadIdx.x + 1;

    if(i<n-1) {
        // load shared memory
        buffer[i] = in[i];
        if(i==1) {
            buffer[0] = in[0];
            buffer[n-1] = in[n-1];
        }

        __syncthreads();

        out[i] = 0.25*(buffer[i-1] + 2.0*buffer[i] + buffer[i+1]);
    }
}
```

Synchronizing threads

The built-in CUDA function `__syncthreads()` creates a barrier, where all threads in a thread block synchronize.

- Threads wait for all threads in thread block to finish loading shared memory buffer.
- Thread i needs to wait for threads $i-1$ and $i+1$ to load values into `buffer`.
- Synchronization required to avoid race conditions.
  - Threads have to wait for other threads to fill `buffer`.

Declaring shared memory

There are two ways to declare shared memory allocations.

Dynamic allocation

When the memory is determined at run time:

```
extern __shared__ double buffer[];
```

- Note the `extern` keyword.
- The size of memory to be allocated is specified when the kernel is launched.

Static allocation

When the amount of memory is known at compile time:

```
__shared__ double buffer[128];
```

- Here there are 128 double-precision values (1024 bytes) of memory shared by all threads.

Launching with static shared memory

The amount of shared memory should be sufficient for the number of threads.

Using compile time bounds

```
template <int THREADS>
__global__
void kernel(...) {
  __shared__ double buffer[THREADS];

  // ... THREADS must equal blockDim.x
}

// launch kernel with threads per block as a template parameter
kernel<128><<<num_blocks, 128>>>(...);
```

Launching with static shared memory

It is possible to allocate multiple variables as shared memory.

- If the shared memory is used separately, you can use a union to "overlap" the storage.
- Shared memory is a limited resource.

separate storage

```
__global__
void kernel1() {
  // 1536 bytes
  __shared__ int X[128];
  __shared__ double Y[128];

  // OK
  X[i] = (int)Y[i];
}
```

overlapping storage

```
__global__
void kernel2(int n) {
  // 1024 bytes
  __shared__ union {
    int X[128];
    double Y[128];
  } buf;

  // not OK
  buf.X[i] = (int)buf.Y[i];
}
```

Finding resource usage of kernels

The nvcc flag `--resource-usage` will print the resources used by each kernel during compilation:

- shared memory
- constant memory
- registers

using the `--resource-usage` on kernels in previous slide

```
> nvcc --resource-usage -arch=sm_60 shared.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z7kernel2i' for
ptxas info    : Function properties for _Z7kernel2i
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 6 registers, 1024 bytes smem, 324 bytes cmem[0]
ptxas info    : Compiling entry function '_Z7kernel1v' for
ptxas info    : Function properties for _Z7kernel1v
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 6 registers, 1536 bytes smem, 320 bytes cmem[0]
> c++filt _Z7kernel2i
kernel2(int)
```

Note: the kernel names have been mangled (use `c++filt`.)

Launching with dynamic shared memory

An additional parameter is added to the launch syntax

```
blur<<<grid_dim, block_dim, shared_size>>>(...);
```

- `shared_size` is the shared memory **in bytes** to be allocated **per thread block**

Launch blur kernel with shared memory

```
__global__
void blur_shared(double *in, double* out, int n) {
  extern __shared__ double buffer[];

  int i = threadIdx.x + 1;
  // ...
}

// in main()
auto block_dim = n-2;
auto size_in_bytes = n*sizeof(double);

blur_shared<<<1, block_dim, size_in_bytes>>>(x0, x1, n);
```

A version of the blur kernel for arbitrarily large n is provided in `blur.cu` in the example code. The implementation is a bit awkward:

- the `in` and `out` arrays use global indexes
- the shared memory uses thread block local indexes

Is it worth it?

- on Keplar this optimization was worth ≈10%.
- on P100 there is no speedup (I think due to improved read only L1 caching on P100)

The small performance improvement on Keplar was worth it if this was a key kernel in your application...

Buffering

A pipelined workflow uses the output of one "kernel" as the input of another

- On the CPU these can be optimized by keeping the intermediate result in cache for the second kernel.

e.g. two stencils, one applied to the output of the first.

Double blur: naive OpenMP

```
void blur_twice(const double* in , double* out , int n) {
  static double* buffer = malloc_host<double>(n);

  #pragma omp parallel for
  for(auto i=1; i<n-1; ++i) {
    buffer[i] = 0.25*( in[i-1] + 2.0*in[i] + in[i+1]);
  }
  #pragma omp parallel for
  for(auto i=2; i<n-2; ++i) {
    out[i] = 0.25*( buffer[i-1] + 2.0*buffer[i] + buffer[i+1]);
  }
}
```

Double blur: OpenMP with blocking for cache

```
void blur_twice(const double* in , double* out , int n) {
  auto const block_size = std::min(512, n-4);
  auto const num_blocks = (n-4)/block_size;
  static double* buffer = malloc_host<double>((block_size+4)*
       omp_get_max_threads());

  auto blur = [] (int pos, const double* u) {
    return 0.25*( u[pos-1] + 2.0*u[pos] + u[pos+1]);
  };

  #pragma omp parallel for
  for(auto b=0; b<num_blocks; ++b) {
    auto tid = omp_get_thread_num();
    auto first = 2 + b*block_size;
    auto last = first + block_size;

    auto buff = buffer + tid*(block_size+4);
    for(auto i=first-1, j=1; i<(last+1); ++i, ++j) {
      buff[j] = blur(i, in);
    }
    for(auto i=first, j=2;   i<last;   ++i, ++j) {
      out[i] = blur(j, buff);
    }
  }
}
```

Buffering with shared memory

Shared memory is important for caching intermediate results used in pipelined operations.

- Shared memory is an order of magnitude faster than global DRAM.
- By **fusing** pipelined operations in one kernel, intermediate results can be stored in shared memory.
- Similar to blocking and tiling for cache on the CPU.

Double blur: CUDA with shared memory

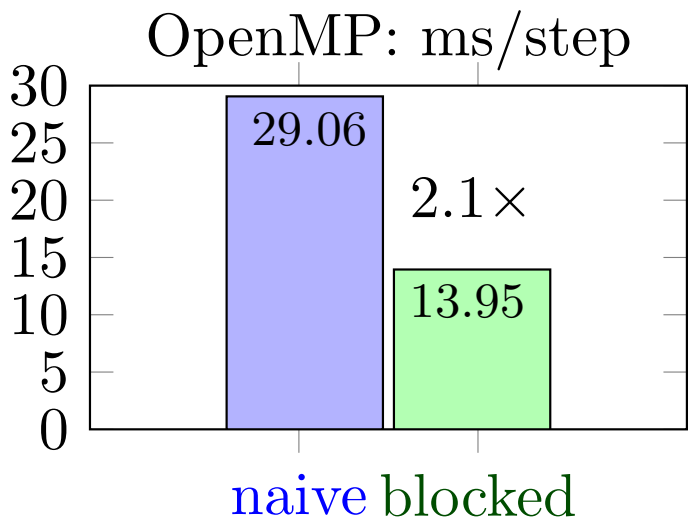
```
__global__ void blur_twice(const double *in, double* out, int n) {
  extern __shared__ double buffer[];

  auto block_start = blockDim.x * blockIdx.x;
  auto block_end   = block_start + blockDim.x;
  auto lid = threadIdx.x + 2;
  auto gid = lid + block_start;

  auto blur = [] (int pos, double const* field) {
    return 0.25*(field[pos-1] + 2.0*field[pos] + field[pos+1]);
  };

  if(gid<n-2) {
    buffer[lid] = blur(gid, in);
    if(threadIdx.x==0) {
        buffer[1]             = blur(block_start+1, in);
        buffer[blockDim.x+2] = blur(block_end+2, in);
    }

    __syncthreads();

    out[gid] = blur(lid, buffer);
  }
}
```

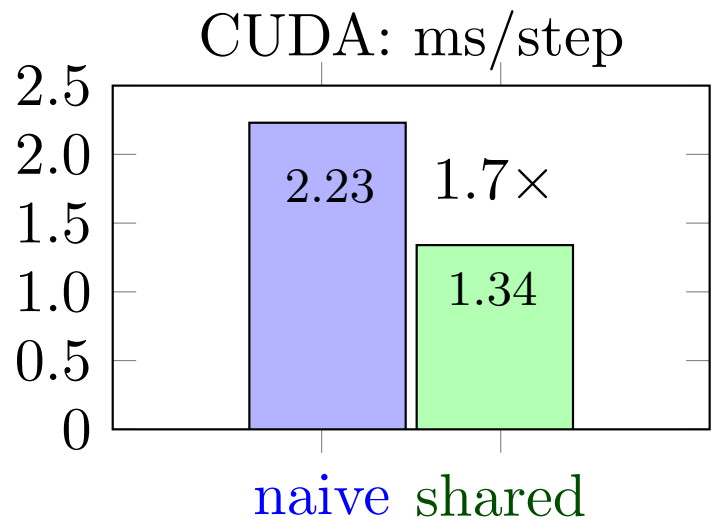
Fused loop results

The OpenMP cache-aware version was harder to implement than the shared-memory CUDA version:

- CUDA seems harder because we have to think and write in parallel from the start.

Both implementations benefit significantly from optimizations for fast on chip memory.





OpenMP results with 18-core Broadwell CPU; CUDA with P100 GPU; $n = 33,554,432$

CPU : optimizing for on-chip memory

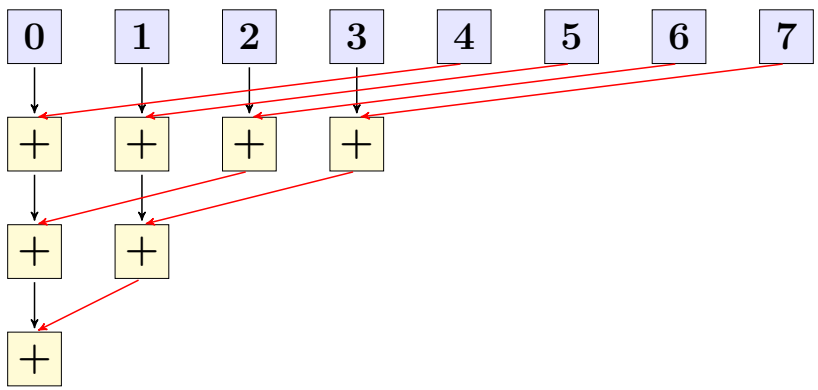
- let hardware prefetcher automatically manage cache
- choose block/tile sizes so that intermediate data will fit in a target cache (L1, L2 or L3)

GPU : optimizing for on-chip memory

- manage shared memory manually
  - more control
  - hardware-specific
- choose thread block sizes so that intermediate data will fit into shared memory on an SMX

Exercise: Shared Memory

- Finish the `shared/string_reverse.cu` example. Assume $n \leq 1024$.
  - With or without shared memory.
  - **Extra**: without any synchronization.
- Implement a dot product in CUDA in `shared/dot.cu`.
  - The host version has been implemented as `dot_host()`
  - Assume $n \leq 1024$.
  - **Extra**: how would you extend it to work for arbitrary $n > 1024$ and n threads?



Communication

Communication in a GPU code occurs at different levels:

- Between threads in a warp;
- Between threads in thread block;
- Between threads in grid;
- Between threads in different grids.

Involves reading and writing shared resources:

- Synchronization required if more than one thread wants to modify (write) a shared resource.

Race conditions

A race condition can occur when more than one thread attempts to access the same memory location concurrently and at least one access is a write.

```
__global__
void race(int* x) {
  ++x[0]
}

int main(void) {
  int* x =
    malloc_managed<int>(1);
  race<<<1, 2>>>(x);
  cudaDeviceSynchronize();
  // what value is in x[0]?
}
```

No Race

t0	t1	x
R		0
I		0
W		1
	R	1
	I	1
	W	2

Race

t0	t1	x
R		0
	R	0
I		0
W		1
	I	1
	W	1

Example where two threads `t0` and `t1` both increment x in memory. The threads use: read (R); write (W); and increment (I).

- Race conditions produce strange and unpredictable results.
- Synchronization is required to avoid race conditions.

Synchronization within a block

Threads in the same thread block can use `__syncthreads()` to synchronize on access to shared memory and global memory

synchronization on global memory

```
__global__
void update(int* x, int* y) {
  int i = threadIdx.x;
  if (i == 0) x[0] = 1;
  __syncthreads();
  if (i == 1) y[0] = x[0];
}

int main(void) {
  int* x = malloc_managed<int>(1);
  int* y = malloc_managed<int>(1);
  update<<<1,2>>>(x, y);
  cudaDeviceSynchronize();
  // both x[0] and y[0] equal 1
}
```

Note: All threads in a block must reach the `__syncthreads()`

- otherwise strange things (may) happen!

Atomic Operations: motivation

What is the output of the following code?

```
#include <cstdio>
#include <cstlib>
#include <cuda.h>
#include "util.hpp"

__global__ void count_zeros(int* x, int* count) {
  int i = threadIdx.x;
  if (x[i]==0) *count+=1;
}

int main(void) {
  int* x = malloc_managed<int>(1024);
  int* count = malloc_managed<int>(1);
  count = 0;
  for (int i=0; i<1024; ++i) x[i]=i%128;

  count_zeros<<<1, 1024>>>(x, count);
  cudaDeviceSynchronize();
  printf("result %d\n", *x); // expect 8

  cudaFree(x);
  return 0;
}
```

Atomic Operations

An **atomic memory operation** is an uninterruptable read-modify-write memory operation:

- Serializes contentious updates from multiple threads;
- **The order** in which concurrent atomic updates are performed **is not defined**;
- However none of the atomic updates will be lost.

race

```
__global__ void inc(int* x) {
  *x += 1;
}
```

no race

```
__global__ void inc(int* x) {
  atomicAdd(x, 1);
}
```

```
// pseudo-code implementation of atomicAdd
__device__ int atomicAdd(int *p, int v) {
  int old;
  exclusive_single_thread {
    old = *p; // Load from memory
    *p = old + v; // Store after adding v
  }
  return old; // return original value before modification
}
```

Atomic Functions

CUDA has a range of atomic funtions, including:

- **Arithmetic**: `atomicAdd()`, `atomicSub()`, `atomicMax()`, `atomicMin()`, `atomicCAS()`, `atomicExch()`.
- **Logical**: `atomicAnd()`, `atomicOr()`, `atomicXor()`.

These functions take both 32 and 64 bit arguments

- `atomicAdd()` gained supported for `double` in CUDA 8 with Pascal.
- see the CUDA Programming Guide for specific details.

Atomic Performance

Atomic operations are a blunt instrument:

- Even without contention, atomics are slower than normal accesses (loads, stores);
- Performance can degrade when many threads attempt atomic operations on few memory locations.

Try to avoid or minimise the number of atomic operations.

- Attempt to use shared memory and structure algorithms to avoid synchronization wherever possible.
- Try performing operation at warp level or block level.
- Use atomics for infrequent, sparse and/or unpredictable global communication.

Exercises: Atomics

- What is `shared/hist.cu` supposed to do?
  - What is the output?
  - Fix it to get the expected output.
- Improve `shared/dot.cu` to work for arbitrary n

Introduction to GPUs in HPC

Ben Cumming, CSCS
July 12, 2020

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Concurrency

Concurrency

Concurrency is the ability to perform multiple CUDA operations simultaneously, including:

- CUDA kernels;
- Copying from host to device;
- Copying from device to host;
- Operations on the host CPU.

Concurrency enables

- Both CPU and GPU can work at the same time.
- Multiple tasks can be run on GPU simultaneously.
- Overlapping of communication and computation.

Host code

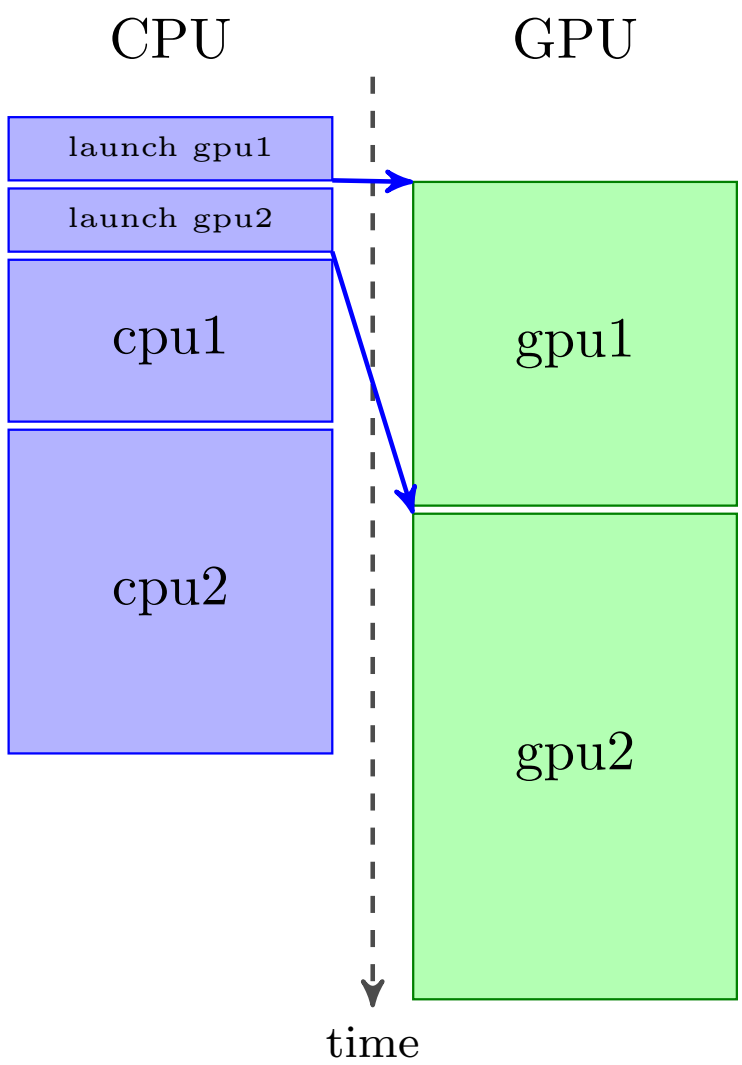
```
kernel_1<<<...>>>(...);
kernel_2<<<...>>>(...);
host_1(...);
host_2(...);
```

The host:

- launches the two CUDA kernels;
- then executes host calls sequentially.

The GPU:

- executes asynchronously to host;
- executes kernels sequentially.



The CUDA language and runtime libraries provide mechanisms for coordinating asynchronous GPU execution:

- **CUDA streams** can concurrently run independent kernels and memory transfers;
- **CUDA events** can be used to synchronize streams and query the status of kernels and transfers.

Streams

A CUDA stream is a sequence of operations that execute in **issue order** on the GPU.

- CUDA operations are kernels and copies between host and device memory spaces.

Streams and concurrency

- Operations in different streams **may** run concurrently
  - requires sufficient resources on the GPU (registers, shared memory, SMXs, etc).
- Operations in the same stream **are** executed sequentially.
- If no stream is specified, all kernels are launched in the default stream.

Managing streams

A stream is represented using a `cudaStream_t` type

- `cudaStreamCreate(cudaStream_t* s)` and `cudaStreamDestroy(cudaStream_t s)` can be used to create and free CUDA streams respectively.
- To launch a kernel on a stream specify the stream id as a fourth parameter to the launch syntax:

```
kernel<<<grid_dim, block_dim, shared_size, stream>>>(...)
```

- The default CUDA stream is the `NULL` stream, or stream 0 (`cudaStream_t` is an integer).

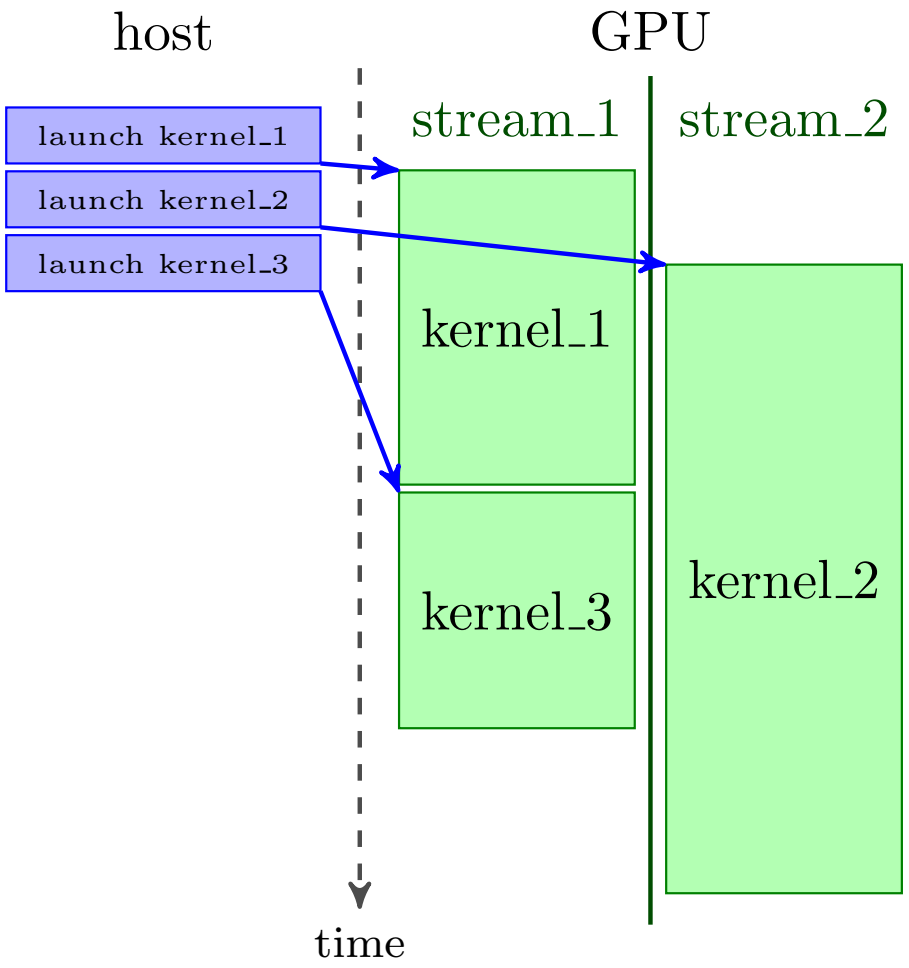
Basic cuda stream usage

```
// create stream
cudaStream_t stream;
cudaStreamCreate(&stream);
// launch kernel in stream
my_kernel<<<grid_dim, block_dim, shared_size, stream>>>(..)
// release stream when finished
cudaStreamDestroy(stream);
```

Host code

```
kernel_1<<<_,_,_,stream_1>>>();
kernel_2<<<_,_,_,stream_2>>>();
kernel_3<<<_,_,_,stream_1>>>();
```

- `kernel_1` and `kernel_3` are serialized in `stream_1`.
- `kernel_2` can run asynchronously in `stream_2`.
- **Note** `kernel_2` will only run concurrently if there are sufficient resources available on the GPU, i.e. if `kernel_1` is not using all of the SMXs.



Asynchronous copy

```
cudaMemcpyAsync(*dst, *src, size, kind, cudaStream_t stream = 0);
```

- Takes an additional parameter stream, which is 0 by default.
- Returns immediately after initiating copy:
  - Host can do work while copy is performed;
  - Only if **pinned memory** is used.
- Copies in the same direction (i.e. H2D or D2H) are serialized.
  - Copies from host→device and device→host are concurrent if in different streams.

Pinned memory

Pinned (or page-locked) memory will not be paged out to disk:

- The GPU can safely remotely read/write the memory directly without host involvement;
- Only use for transfers, because it easy to run out of memory.

Managing pinned memory

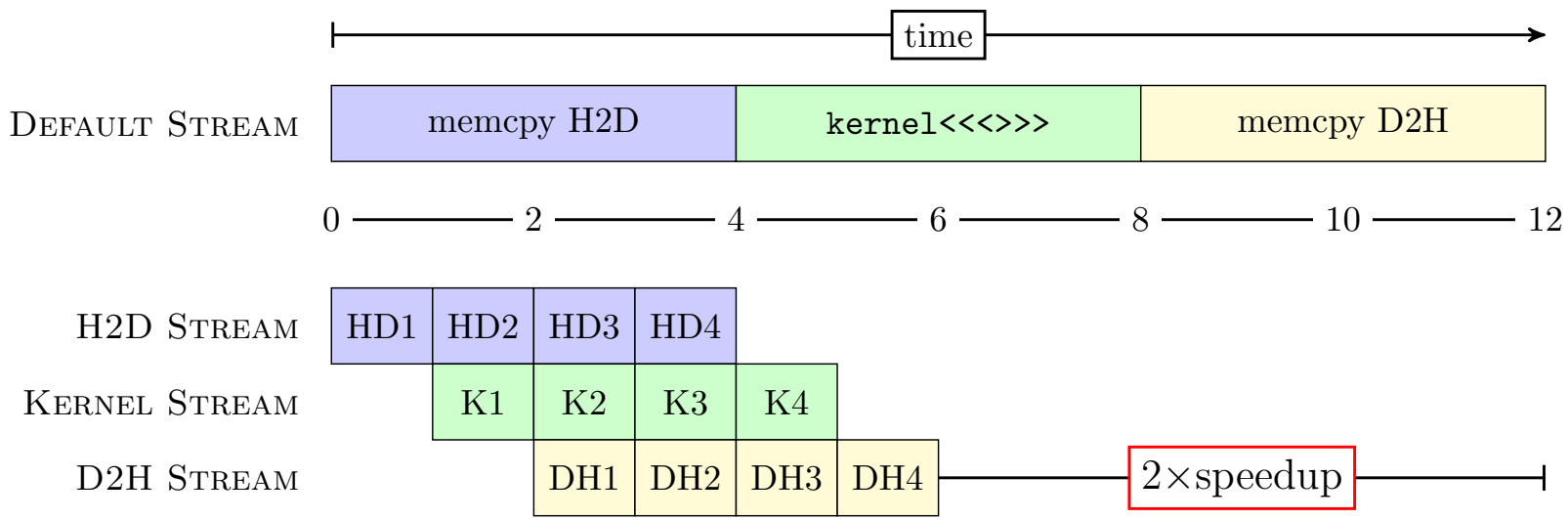
`cudaMallocHost(**ptr, size);` and `cudaFreeHost(*ptr);`

- Allocate and free pinned memory (`size` is in bytes).

Asynchronous copy example: streaming workloads

Computations that can be performed independently, e.g. our `axpy` example:

- Data in host memory has to be copied to the device, and the result copied back after the kernel is computed.
- Overlap copies with kernel calls by breaking the data into chunks.



CUDA events

To implement the streaming workload we have to coordinate operations on the GPU. CUDA events can be used for this purpose.

- Synchronize tasks in different streams, e.g.:
  - Don't start kernel in kernel stream until data copy stream has finished;
  - Wait until required data has finished copy from host before launching kernel.
- Query status of concurrent tasks:
  - Has kernel finished/started yet?
  - How long did a kernel take to compute?

Managing events

`cudaEventCreate(cudaEvent_t*);` and `cudaEventDestroy(cudaEvent_t);`

- Create and free `cudaEvent_t`.

`cudaEventRecord(cudaEvent_t, cudaStream_t);`

- Enqueue an event in a stream.

`cudaEventSynchronize(cudaEvent_t);`

- Make host execution wait for event to occur.

`cudaEventQuery(cudaEvent_t)`

- Test if the work before an event in a queue has been completed.

`cudaEventElapsedTime(float*, cudaEvent_t, cudaEvent_t);`

- Get time between two events.

Using events to time kernel execution

```
cudaEvent_t start, end;
cudaStream_t stream;
float time_taken;

// initialize the events and streams
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaStreamCreate(&stream);

cudaEventRecord(start, stream); // enqueue start in stream
my_kernel<<<grid_dim, block_dim, 0, stream>>>();
cudaEventRecord(end, stream);   // enqueue end in stream
cudaEventSynchronize(end);      // wait for end to be reached
cudaEventElapsedTime(&time_taken, start, end);

std::cout << "kernel took " << 1000*time_taken << " s\n";

// free resources for events and streams
cudaEventDestroy(start);
cudaEventDestroy(end);
cudaStreamDestroy(stream);
```

Copy→kernel synchronization

```
cudaEvent_t event;
cudaStream_t kernel_stream, h2d_stream;
size_t size = 100*sizeof(double);
double *dptr, *hptr;

// initialize
cudaEventCreate(&event);
cudaStreamCreate(&kernel_stream);
cudaStreamCreate(&h2d_stream);

cudaMalloc(&dptr, size);
cudaMallocHost(&hptr, size); // use pinned memory!

// start asynchronous copy in h2d_stream
cudaMemcpyAsync(dptr, hptr, size,
                cudaMemcpyHostToDevice, h2d_stream);
// enqueue event in stream
cudaEventRecord(event, h2d_stream);
// make kernel_stream wait for copy to finish
cudaStreamWaitEvent(kernel_stream, event, 0);
// enqueue my_kernel to start when event has finished
my_kernel<<<grid_dim, block_dim, 0, kernel_stream>>>();

// free resources for events and streams
cudaEventDestroy(event);
cudaStreamDestroy(h2d_stream);
cudaStreamDestroy(kernel_stream);
cudaFree(dptr);
cudaFreeHost(hptr);
```

Exercises

1. Open `include/util.hpp` and understand
   `copy_to_{host/device}_async()` and `malloc_pinned()`
2. Open `include/cuda_event.h` and `include/cuda_stream.h`
   - what is the purpose of these classes?
   - what does `cuda_stream::enqueue_event()` do?
3. Open `async/memcopy1.cu` and run
   - what does the benchmark test?
   - what is the effect of turning on `USE_PINNED`?
     Hint: try small and large values for `n` (8, 16, 20, 24)
4. Inspect `async/memcopy2.cu` and run
   - what effect does changing the number of chunks have?
5. Inspect `async/memcopy3.cu` and run
   - how does it differ from `memcopy2.cu`?
   - what effect does changing the number of chunks have?

Using events to time kernel execution: **with helpers**

```
CudaStream stream(true);

auto start = stream.enqueue_event();
my_kernel<<<grid_dim, block_dim, 0, stream.stream()>>>();
auto end = stream.enqueue_event();
end.wait();
auto time_taken = end.time_since(start);

std::cout << "kernel took " << 1000*time_taken << " s\n";
```

Copy→kernel synchronization: **with helpers**

```
CudaStream kernel_stream(true), h2d_stream(true);
auto size = 100;
auto dptr = device_malloc<double>(size);
auto hptr = pinned_malloc<double>(size);

copy_to_device_async<double>(hptr,dptr,size,h2d_stream.stream());
auto event = h2d_stream.enqueue_event();
kernel_stream.wait_on_event(event);
my_kernel<<<grid_dim, block_dim, 0, kernel_stream.stream()>>>();

cudaFree(dptr);
cudaFreeHost(hptr);
```

Rough guidelines for concurrency

Ideally for most workloads you don't want to rely on streams to fill the GPU with work.

- A sign that the working set per GPU is not large enough;
- Full concurrency is difficult in practice;
  - A low-level optimization strategy for the last few %.
- This isn't a hard and fast rule.

Streams come into their own for overlapping communication and computation.

- Possible to transfer data in both directions concurrently with kernel execution.

Introduction to GPUs in HPC

Ben Cumming, CSCS
July 15, 2020

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

2D and 3D Launch Configurations

Launch Configuration

- So far we have used one-dimensional launch configurations:
  - Threads in blocks indexed using `threadIdx.x`.
  - Blocks in a grid indexed using `blockIdx.x`.
- Many kernels map naturally onto 2D and 3D indexing:
  - e.g. Matrix-matrix operations;
  - e.g. Stencils.

Full Launch Configuration

Kernel launch dimensions can be specified with `dim3` structs

```
kernel<<<dim3 grid_dim, dim3 block_dim>>>(...);
```

- `dim3.x`, `dim3.y` and `dim3.z` specify the launch dimensions;
- Can be constructed with 1, 2 or 3 dimensions;
- Unspecified `dim3` dimensions are set to 1.

launch configuration examples

```
// 1D: 128x1x1 for 128 threads
dim3 a(128);
// 2D: 16x8x1  for 128 threads
dim3 b(16, 8);
// 3D: 16x8x4  for 512 threads
dim3 c(16, 8, 4);
```

The `threadIdx`, `blockDim`, `blockIdx` and `gridDim` can be treated like 3D points via the `.x`, `.y` and `.z` members.

matrix addition example

```
__global__
void MatAdd(float *A, float *B, float *C, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if(i<n && j<n) {
        auto pos = i + j*n;
        C[pos] = A[pos] + B[pos];
    }
}
int main() {
    // ...
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(n / threadsPerBlock.x, n / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    // ...
}
```

Exercise: Launch Configuration

- Write the 2D diffusion stencil in `diffusion/diffusion2d.cu`
- Set up 2D launch configuration in the main loop
- Draw a picture of the solution to validate it
  - a plotting script is provided for visualizing the results
  - use a small domain for visualization

```
# Build and run after writing code
srun diffusion2d 8 1000000

# Do the plotting
module load daint-gpu
module load PyExtensions/3.6.5.7-CrayGNU-19.10
python plotting.py -s
```