CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Introduction to CUDA

Ben Cumming, CSCS
June 17, 2015

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Introduction

The plan

- 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
- understand which types of workloads can best take advantage of GPU resources
- learn about thread cooperation and synchronization in CUDA
- learn about concurrent task-based parallelism with CUDA

Prerequisites for the course

- no GPU or graphics experience required
- I assume C++ knowledge
  - I will be using C++11 (the bits that make C++ easier!)
  - Fortran users are encouraged to work with a C++ user for the practical exercises
- the generic GPU programming concepts in the CUDA part will be useful for people interested in OpenACC

CUDA language is a superset of C++

- write CPU code using C++ (C++11 since CUDA 6.5)
- write kernels to run on GPU using new keywords
- use special syntax for launching kernels on GPU

CUDA is GPU-specific

- the CUDA language extensions define the **programming model**
- features map directly to hardware (e.g. shared memory, thread blocks)

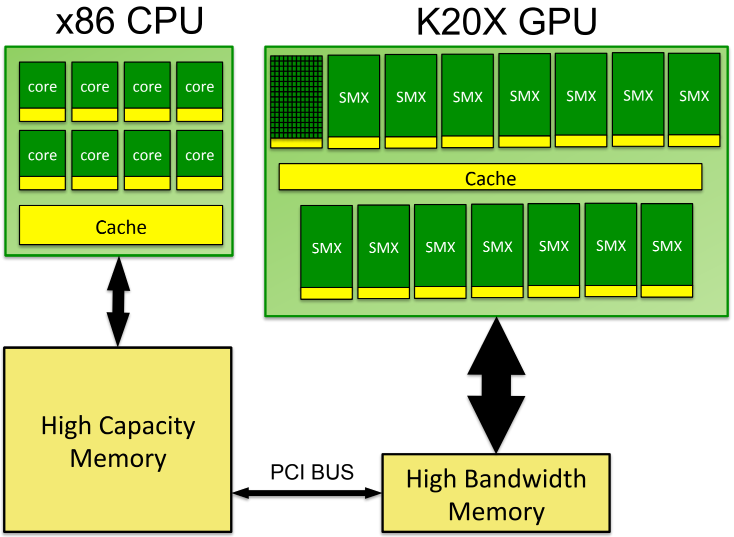
CUDA toolkit is more than just a language

- runtime library for managing GPU resources
- tools for profiling and debugging

What about the GPU in my laptop/desktop/cluster?

- the GPUs in Piz Daint are NVIDIA Tesla K20X devices
- Tesla devices are high-end products with features required for high-performance computing
  - higher double precision performance
  - large DRAM
  - ECC memory
- the K20X Tesla cards use the Kepler architecture
  - some features are not supported by older cards
- I focus on features of the K20X devices for this course

Working with GPU memory


x86 CPU
core
core
core
core
core
core
core
core
Cache
K20X GPU
SMX
SMX
SMX
SMX
SMX
SMX
SMX
Cache
SMX
SMX
SMX
SMX
SMX
SMX
SMX
High Capacity Memory
PCI BUS
High Bandwidth Memory

Host and device have separate memory spaces

- data must be copied between host and device memory via PCI
- data must be in device memory for kernels to access
  - not strictly true…
  - but a strict requirement for high performance
- ensure data is in the right memory space **before** computation starts
- on Piz Daint the respective bandwidths are:
  - PCIe2 = 6 GB/s each way
  - Sandy Bridge CPU = 35 GB/s
  - K20X = 180 GB/s

CUDA uses C pointers to reference GPU memory

```
double *data = //pass an address to either host or device memory
```

- a pointer can hold an address in **either** device **or** host memory
- accessing a device pointer in host code, or vice versa, is **undefined behaviour**
- we have to take care that we know which memory space a pointer is addressing

The CUDA runtime library provides functions that can be used to allocate, free and copy device memory

Allocating device memory

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

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

Freeing device memory

`cudaFree(void *ptr)`

Allocate memory for 100 doubles on device

```
double *v; // C pointer that will point to device memory
auto size_in_bytes = 100*sizeof(double);
cudaMalloc(&v, size_in_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: `cudaMemcpyHostToDevice`, also `DeviceToHost`, `DeviceToDevice`

Copy 100 doubles to device, then back to host

```
int 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;
int 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: API Basics

Open `cuda/exercises/axpy/util.h`

1. what does `cuda_check_error()` 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 from host to device
   - use the example for the reverse operation `copy_to_host`
   - remember to check for errors!
4. compile the test and run
   - it will pass with no errors on success

```
make axpy_cublas
aprun ./axpy_cublas 8
```

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Going Parallel : Kernels and Threads

Threads and kernels

- **threads** are run simultaneously on GPU (1000s)
- **kernel** is the task run by each thread
- CUDA provides language support for
  - writing kernels
  - launching many threads to execute parallel kernel
- CUDA hides the low-level details of launching threads

The process for porting to CUDA

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`)

The exercise in the first section used CUBLAS to perform scaled vector addition

$$y = y + \alpha x$$

- x and y are vectors of length n
- α is scalar

`axpy` can be expressed into n independent 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 with for loop

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

What is a kernel?

- 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 that called from the host
- `threadIdx` used to find unique id of each thread

launching a kernel

- host code launches a kernel on the GPU **asyncronously**
- CUDA provides special `<<<_,_>>>` syntax for launching a kernel
  - `foo<<<1, num_threads>>>(args... )` will launch the kernel `foo` with `num_threads` parallel threads.

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);
```

Exercise: My First Kernel

Open `cuda/exercises/axpy/axpy_kernel.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. Replace the call to `cublasDaxpy` with an invocation of your new kernel
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?

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Scaling Up : Thread Blocks

In the `axpy` exercises we were limitted 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

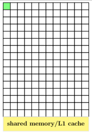
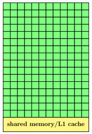
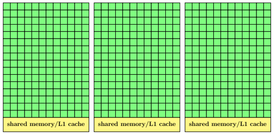
kernels are executed in groups of threads called **thread blocks**

- 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 of grids+blocks+threads?

Because coordination and sharing 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
- on the K20X GPU streaming multiprocessor (SMX) has 192 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 14 SMXs in the K20X 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	 shared memory/L1 cache shared memory/L1 cache shared memory/L1 cache	▪ 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
- when using 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]`

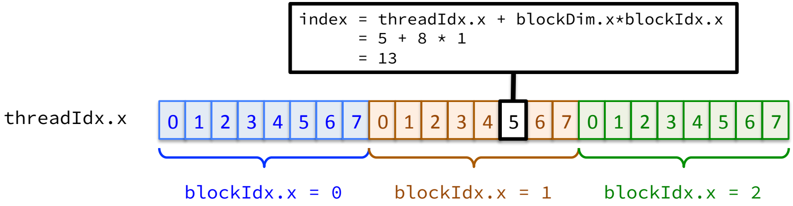
Calculating thread indexes

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
```



```
index = threadIdx.x + blockDim.x*blockIdx.x
      = 5 + 8 * 1
      = 13
```

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`.

- in this case we have one thread block in which not all threads will work

Calculating grid dimensions

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

The number of threads per block impacts performance

- the optimal number depends on the resources (registers, shared memory, etc) that a kernel requires

Choosing block size automatically (CUDA 6.5 and later)

```
int block_size, min_grid_dim;

cudaOccupancyMaxPotentialBlockSize(&min_grid_size, &block_size,
                                   add_gpu, 0, n);
auto num_blocks = (n + (block_size-1)) / block_size;

add_gpu<<<num_blocks, block_size>>>(a, b, n);
```

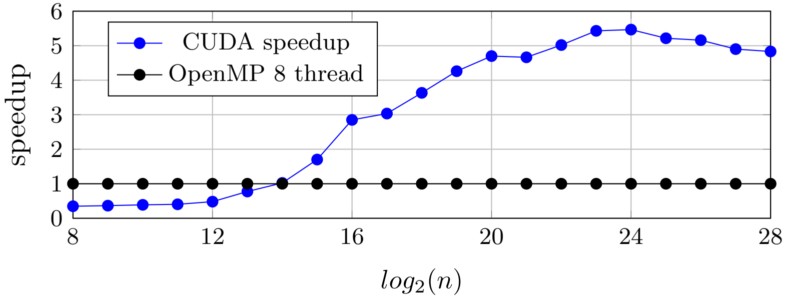
The variable `min_grid_size` is set to the minimum number of blocks required to **saturate** the GPU, i.e. provide the GPU with enough work to utilize all of the SMXs.

Exercise: Blocks

Open `cuda/exercises/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
   - how does it affect the kernel execution time?
5. **extra**: Compare scaling with the `axpy_omp` benchmark
6. **extra**: Experiment with varying the block size
   - try `block_size` calculated by `cudaOccupancyMaxPotentialBlockSize`.

Exercise: Results



The GPU is a throughput device:

- the CUDA implementation is faster for $2^{15} \approx 32,000$
- requires $2^{20} \approx 1,000,000$ to get “advertised” $5\times$ speedup

You have to provide enough parallelism to exploit many cores

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Cooperating Threads

Most algorithms do not lend themselves to trivial parallelization

reductions : e.g. dot product

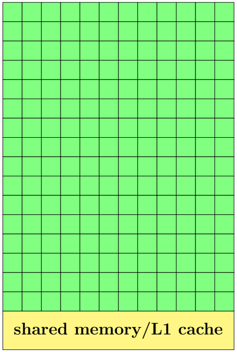
```
int dot(int *x, int *y, int n){
  int sum = 0.;
  for(auto i=0; i<n; ++i)
    sum += x[i]*y[i];
  return sum;
}
```

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 piplined 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]+2f*in[i]);
  for(auto i=2; i<n-2; ++i)
    out[i] = 0.25f*(buff[i-1]+buff[i+1]+2f*buff[i]);
}
```



Block-Level synchronization

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 blocks

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

One-dimensional blur kernel

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

- each output value is a linear combination of neighbours in input array
- first we look at naiive implementation

Host implementation of blur kernel

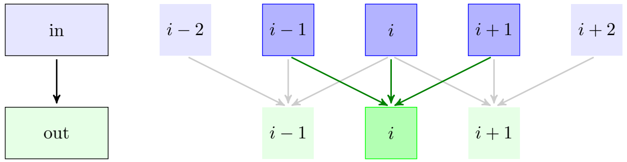
```
void blur(double *in, double *out, int n){
  float buff[n];
  for(auto i=1; i<n-1; ++i)
    out[i] = 0.25*(2*in[i]+in[i-1]+in[i+1]);
}
```

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

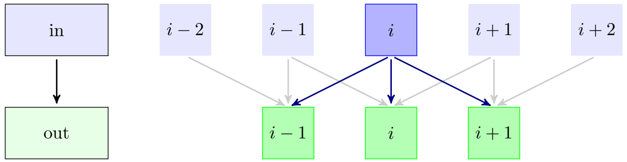
First implementation of blur kernel

```
__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

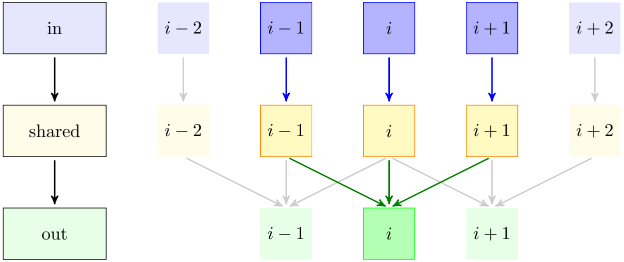


Alternatively, each value in the input array has to be loaded 3 times



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] = in[n];
        }

        __syncthreads();

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

Declaring shared memory

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

- the size of memory to be allocated is specified when the kernel is launched

Synchronizing threads

```
__syncthreads();
```

- 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`

Launching kernels with 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);
```

Is it worth it?

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

The ˜10% performance improvement might be worth it, depending on how important the kernel is to overall application performance

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

An example is two stencils, applied in order

Double blur: basic 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

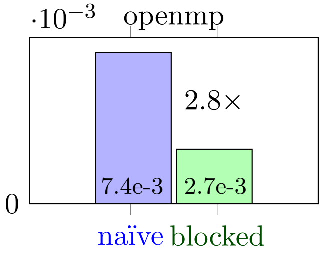
```
__global__ void blur_twice(const double *in, double* out, int n) {
  extern __shared__ double buffer[];

  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[li] = blur(gi, in);
    if(threadIdx.x==0) {
        buffer[1]              = blur(block_start+1, in);
        buffer[blockDim.x+2] = blur(block_end+2, in);
    }

    __syncthreads();

    out[gi] = blur(li, buffer);
  }
}
```

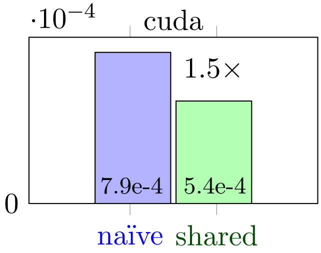
Fused loop results

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

- CUDA is harder to start because it forces us to think and write in parallel

both implementations benefit significantly from optimizations for fast on chip memory





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

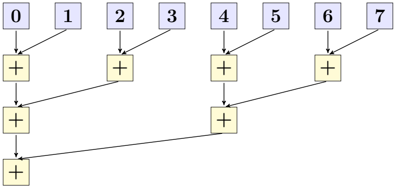
Exercise: Shared Memory

Your task is to implement dot product in CUDA in `cuda/exercises/dot.cu`.

- the host version has been implemented as `dot_host()`
- assume that n is a power of 2 and $n \leq 1024$

Extensions :

1. can you make it work for arbitrary $n < 1024$?
2. how would you extend it to work for arbitrarily large n?



Concurrency

Concurrency

Concurrency is the ability to perform multiple CUDA operations simultaneously

- 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

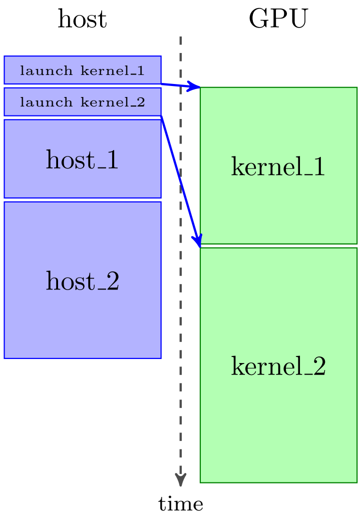
```
kernel_1<<<...>>>(...);
kernel_2<<<...>>>(...);
host_1(...);
host_2(...);
```

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 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
  - there have to be sufficient resources on the GPU (registers, shared memory, blocks, 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)

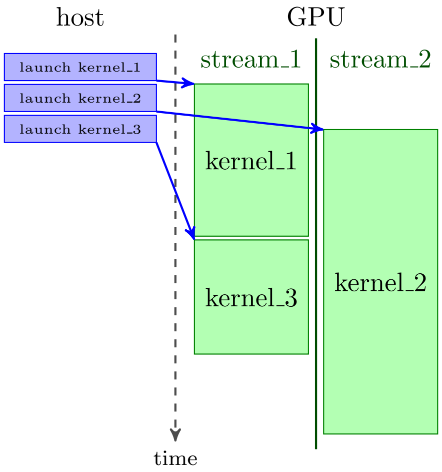
Basic cuda stream useage

```
// 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_2` are serialized in `stream_1`
- `kernel_2` can run asynchronously in `stream_2`
- note that `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, count, 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 in opposite directions are concurrent if in different streams

What is pinned memory?

Pinned memory (or page-locked) memory will not be paged out to disk when memory runs low

- 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

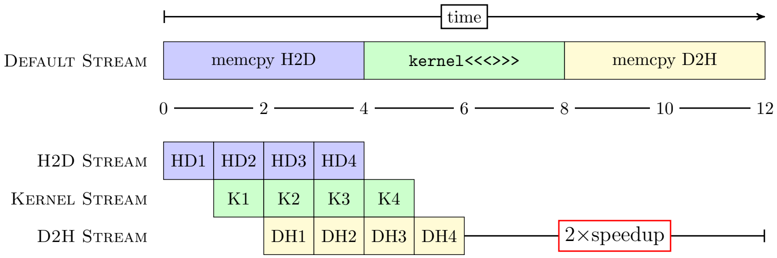
`cudaMallocHost(**ptr, size);` and `cudaFreeHost(*ptr);`

- 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.
- we can overlap the copies with the 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!

cudaMemcpyAsync // start asynchronous copy in h2d_stream
  (dptr, hptr, size, cudaMemcpyHostToDevice, h2d_stream);
cudaEventRecord(event, h2d_stream);  // enqueue event in stream
// make kernel_stream wait for copy to finish
cudaStreamWaitEvent(kernel_stream, event, 0);
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 `util.h` in `cuda/examples/async` and look at the helpers for
   - asynchronous copy `copy_to_{host/device}_async()`
   - pinned allocation `malloc_pinned_host()`
2. Open `CudaEvent.h` and `CudaStream.h`
   - what is the purpose of these classes?
   - what does `CudaStream::enqueue_event()` do?
3. Open `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 `memcopy2.cu` and run
   - what effect does changing the number of chunks have?
5. Walk through `memcopy3.cu`
   - what effect does changing the number of chunks have?

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

```
CudaEvent start, end;
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**

```
CudaEvent event;
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);
```

Profiling CUDA applications

To analyze concurrent applications we need tools that can visually represent application flow.
The CUDA toolkit provides the tools **nvprof** and **nvvp** for profiling our GPU applications

- there are visual tools for Windows and Eclipse too
- they work for OpenACC applications too

nvprof

nvprof is a command line tool

- can be used to generate text reports
- `nvprof --help` for a full list of options
- `nvprof app.exe` will perform basic profling of application and print text summary
- `nvprof -o profile.out app.exe` will save profile information to file `profile.out` for visualization with nvvp

Demonstration

Use nvprof on the memcopy test codes

nvvp

nvvp is a graphical tool for visualizing CUDA applications

- can also be used to perform interactive profiling and guided analysis
- this is not so easy on Cray systems
- we can also use the output from nvprof
  - use `nvprof -o profile.out ... ./app.out` to generate detailed analysis
  - this can take a long time, because each kernel has to be replayed multiple times to collect all of the information required for the report.

Demonstration

Use nvvp on the output of nvprof for the memcopy examples

Some 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 kernels execution