CUDA: Cooperation Between Threads

Ben Cumming, CSCS

February 18, 2016

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

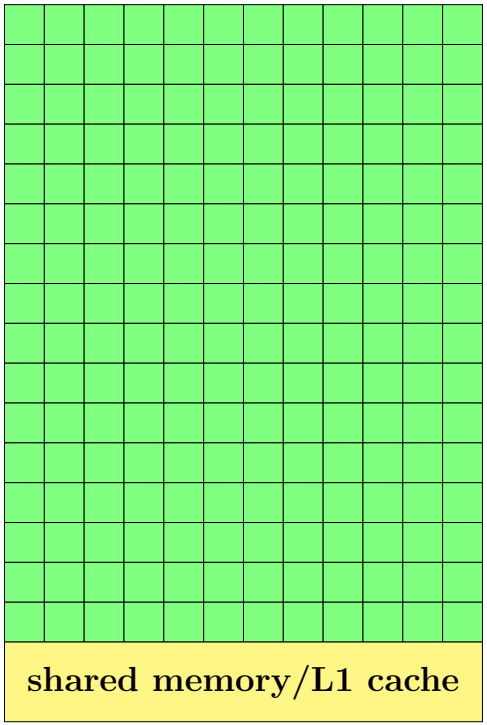
```
int dot(int *x, int *y, int n){
  int sum = 0.;
  for(auto i=0; i<n; ++i)
    sum += x[i]*y[i];
  return sum;
}
```

scan : e.g. prefix or partial sum

```
void prefix_sum(int *x, int n){
  for(auto i=1; i<n; ++i)
      x[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]+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 threads in different 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
- all threads in a block are on the same SMX
- no synchronization possible between threads in different thread blocks
  - ... except via atomic operations on global memory

Note: there are some low level intrinsics for cooperation between threads in the same **warp**

- warp votes and shuffles
- still restricted to the same thread block

One-dimensional blur kernel

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

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

Host implementation of blur kernel

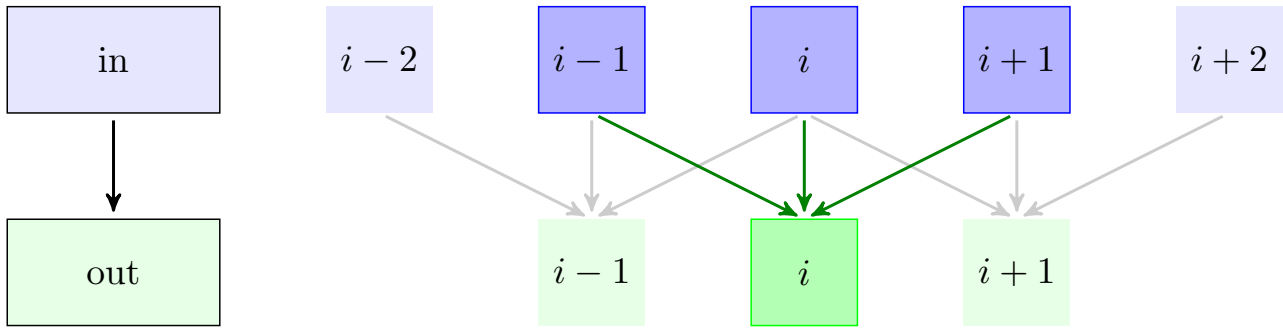
```
void blur(double *in, double *out, int n){
  float buff[n];
  for(auto i=1; i<n-1; ++i)
    out[i] = 0.25*(in[i-1] + 2*in[i] + in[i+1]);
}
```

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

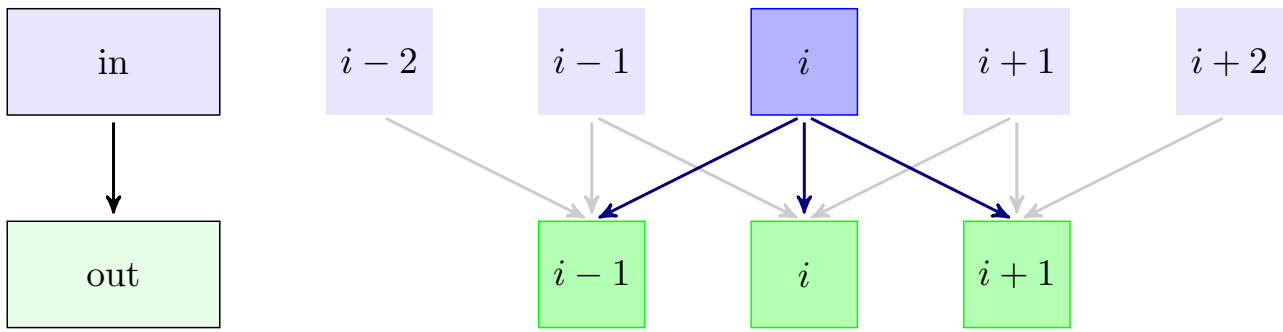
First implementation of blur kernel

```
__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

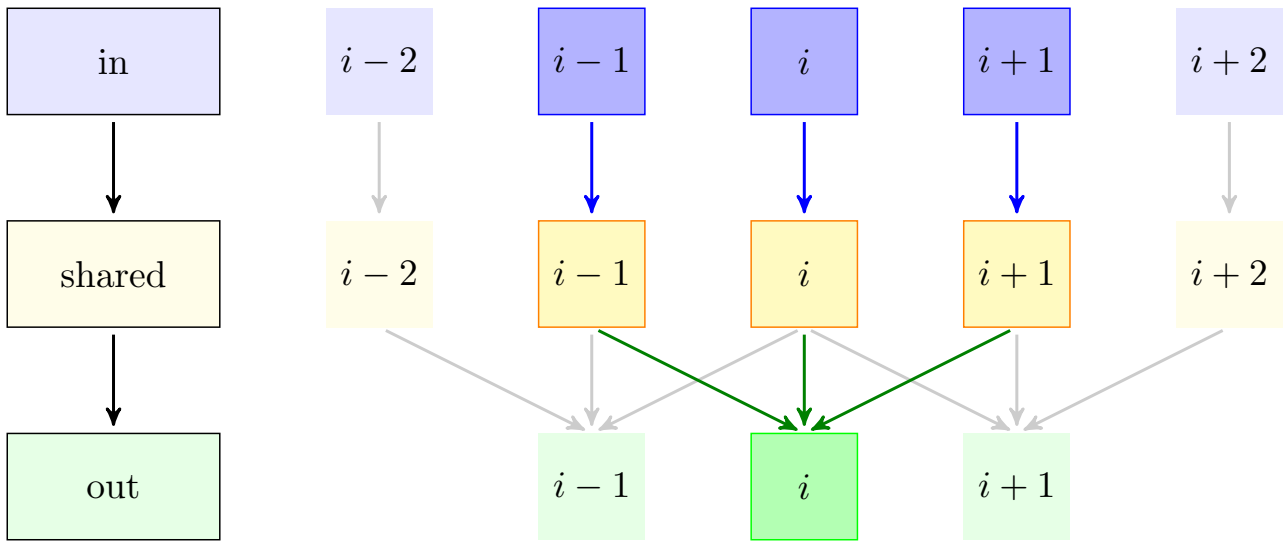


Alternatively, each value in the input array has to be loaded 3 times



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

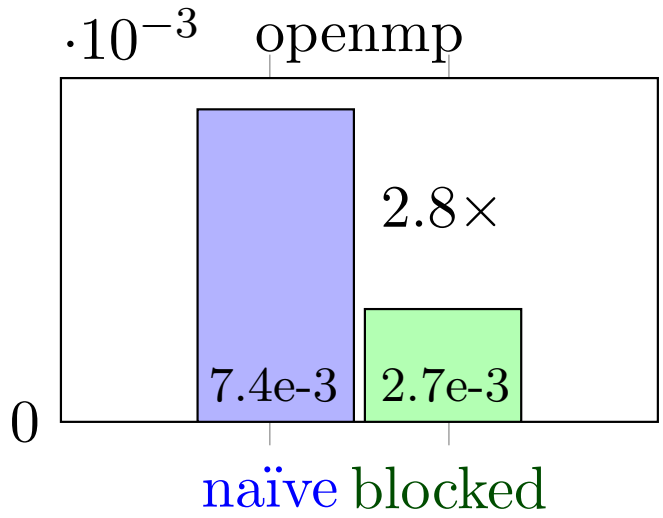
```
__global__ void blur_twice(const double *in, double* out, int n) {
  extern __shared__ double buffer[];

  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);
  }
}
```

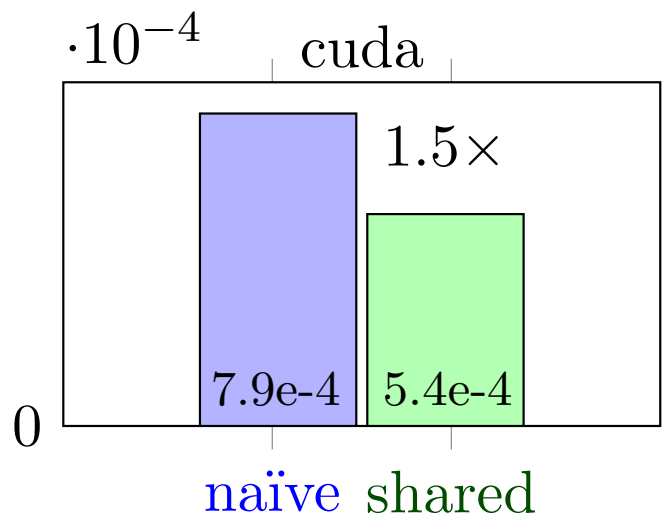
Fused loop results

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

- CUDA is initially harder because we have to think and write in parallel from the start

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

- finish the `shared/string_reverse.cu` example
- implement a dot product in CUDA in `shared/dot.cu`.
  - the host version has been implemented as `dot_host()`
  - assume that n is a power of 2 and $n \leq 1024$
  - **extra**: can you make it work for arbitrary $n \leq 1024$?
  - **extra**: how would you extend it to work for arbitrary $n > 1024$ and n threads?