CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Writing GPU Kernels

Ben Cumming, CSCS
July 12, 2020

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;
}
```

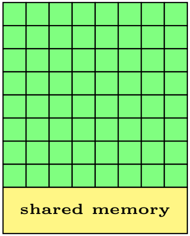
scan : e.g. prefix sum

```
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]);
}
```

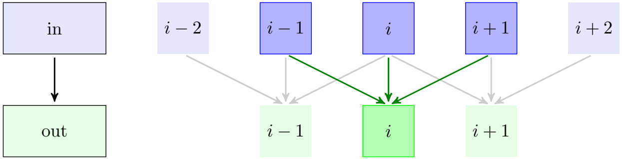
1D blur kernel on GPU

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

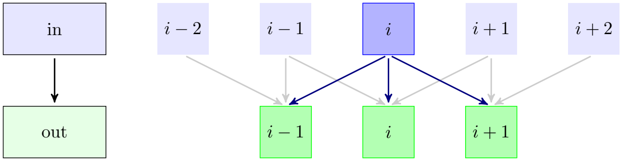
First implementation of blur kernel

```
__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

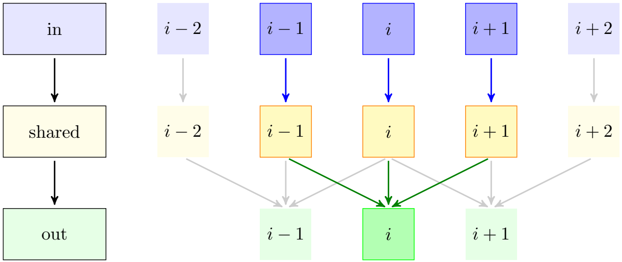


Alternatively, each value in the input array has to be loaded 3 times



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

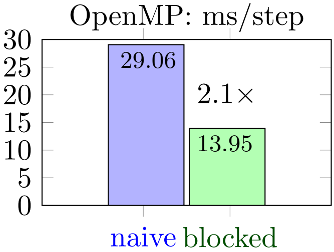
```
__global__ void blur_twice(const double *in, double* out, int n) {
  extern __shared__ double buffer[];

  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);
  }
}
```

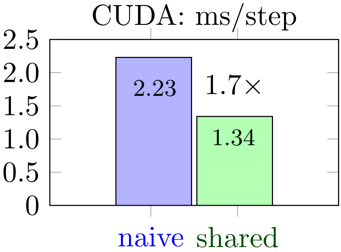
Fused loop results

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

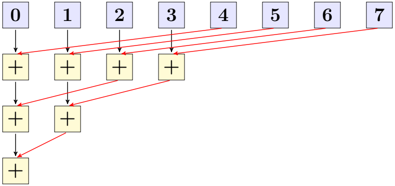
- let hardware prefetcher automatically manage cache
- choose block/tile sizes so that intermediate data will fit in a target cache (L1, L2 or L3)

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