CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich


# Writing GPU Kernels

Sebastian Keller, Prashanth Kanduri
and Ben Cumming, CSCS

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

# Going Parallel: Thread Cooperation

## 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;
}
```

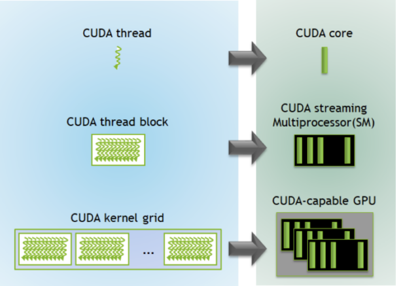
### scan : e.g. prefix sum

```
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

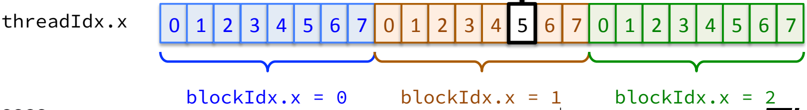
```
void twice_blur(float *in, float *out, int n){
  float buff[n];
  for(auto i=1; i<n-1; ++i)
    buff[i] = 0.25f*(in[i-1]+in[i+1]+2.f*in[i]);
  for(auto i=2; i<n-2; ++i)
    out[i] = 0.25f*(buff[i-1]+buff[i+1]+2.f*buff[i]);
}
```

# Quick Review




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

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



# Types of Cooperation

- **Cooperation between threads in a thread block**:
  - All threads in a block run on the same SM
  - Shared resources such as L1 cache and shared memory
  - CUDA provides mechanisms for synchronization at the block level and lower
  - No necessary synchronization between threads in different blocks

- **Cooperation between different blocks**:
  - Cooperation must occur through global memory
  - CUDA supports *atomic operations*

# Block Level Cooperation

Cooperating through **shared memory**:

- Shared memory is simply a user-managed cache
- Threads in a block can see the same shared memory
- Shared memory is not visible to threads in other blocks executing on the same SM
- Shared memory is a limited resource (**64 KB/SM on a P100**):
  - Shared memory usage per block limits how many blocks can run simultaneously on an SM
  - One thread block can allocate 64 KB for itself…
  - …two thread blocks can allocate 32 KB each

# Shared Memory

- What changes with different GPU architectures:
  - P100: L1 cache and shared mem. have **fixed** sizes
  - A100 & V100: L1 cache and shared mem. are now unified and their portion is **configurable**
  - 64KB/SM on a P100 and up to 128KB/SM on an A100
- What does not change from P100 upwards:
  - Shared memory is divided in 32 equal memory banks, where 32-bit words map to successive banks
  - Each memory bank has a bandwidth of 32 bits/cycle
  - When more than one thread writes to the same bank the write operations are serialized (**bank conflicts**)

# Moving Data To and From Shared Memory

- P100, V100 and A100 coalesce global memory access into 32 byte transactions:
  - For example, a block with 32 threads reading 1 `float` per thread requires only 4 global memory transactions (if and only if the `float`s are consecutive in global memory)
  - Each of these `float`s gets placed into consecutive shared memory banks without any bank conflicts

- Things to watch out for:
  - When reading strided data, part of that 32 byte memory transaction will go unused
  - This will decrease the effective memory bandwidth

# Copy Kernel w/ Shared Memory

- Do we need shared memory for this? Nope.
- This kernel is launched as usual.
- What if `DECIMATE > 1`?
- Have we got any memory bank conflicts?

**A naive downsampling kernel**

```
template <int BSIZE, int DECIMATE=1>
__global__ void downsample(const float* in, float* out, int n){

  // Allocate shared memory statically
  __shared__ float buffer[BSIZE];

  auto idx = threadIdx.x + blockIdx.x * BSIZE;

  // Coalesced reads - no bank conflicts
  if (idx * DECIMATE < n)
    buffer[threadIdx.x] = in[idx * DECIMATE];

  __syncthreads();

  // Coalesced writes
  if (idx < n/DECIMATE)
    out[idx] = buffer[threadIdx.x];
}
```

# Synchronizing threads

What does `__syncthreads()` do?

- All threads in the block wait for each other to finish loading data into shared memory.
- Only after `__syncthreads()` the memory read by other threads is guaranteed to be visible to all other threads in the block.
- Do we need synchronization in this example? There's no thread cooperation, so ... nope!
- What might happen if we place the sync inside the if statement?

# 1D blur kernel

A simple stencil operation:

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

- Each output value is a linear combination of neighbors 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]);
}
```

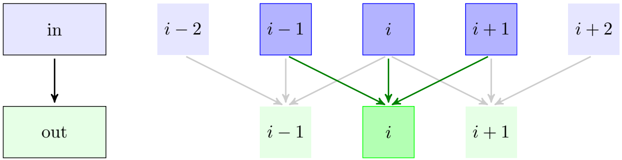
# 1D blur kernel on GPU

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

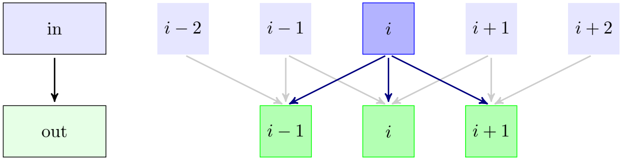
**First implementation of blur kernel**

```
__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

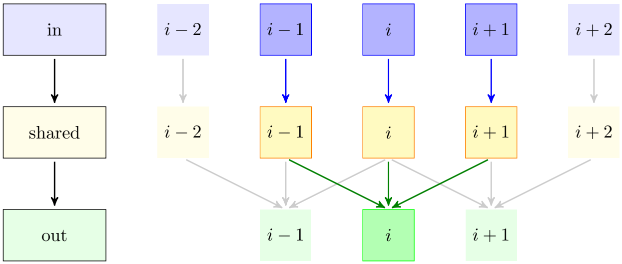



Alternatively, each value in the input array has to be added 3 times into the output array (why is this far worse than the above approach?)



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-1]`.
2. Use values `buffer[i-1:i+1]` to compute kernel.



## Blur kernel with shared memory - single thread block

```
template<int BSIZE>
__global__
void blur_shared_block(const double* in, double* out, int n) {
    __shared__ double buffer[BSIZE];

    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]);
    }
}
```

- Do we need synchronization in this example? Yes!
- Thread $i$ needs to wait for threads $i-1$ and $i+1$ to load values into `buffer`.

# Declaring shared memory

There are two ways to declare shared memory allocations.

## 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.

## 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.

# Launching with static shared memory

- Always need to allocate enough shared memory for the given block size.
- Our `blur_shared_block` kernel needed 2 extra elements ($num\_threads + 2$)

**Launching our blur_shared_block kernel**

```
// Setting the block size to 128 threads
auto n = 128;
blur_shared_block<128+2><<<num_blocks, n>>>(x0, x1, n);
```

# Launching with dynamic shared memory

An additional parameter is added to the launch syntax

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

- `shared_size` is the shared memory **in bytes** to be dynamically 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;
auto size_in_bytes = (n+2)*sizeof(double);

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

# 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

# Back to our blur kernel

A version of the blur kernel for arbitrarily large $n$ is provided in `blur.cu` in the example code. One relevant thing to note is:

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

**have a go at the code!**

- What's the speedup when using shared memory?
- **Extra**: Modify the `blur_shared` kernel to allocate shared memory dynamically.

# Blur kernel results

**and all this for ... no speedup at all?**

- This kernel operates on consecutive memory.
- Coalesced reads and writes.
- Turns out that L1 cache does a pretty good job!
- You might get some speedup if you try this out in a very old GPU.
  - GPUs prior to P100 do not cache on L1 by default.

# Fusing kernels

- Sometimes a 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.
  - On the GPU one can fuse the two operations into the same kernel and use shared memory.
- An example: two concatenated stencil operations.

**Naive double-blur**

```
// Setting the block size to 128 threads
auto n = 128;
blur_shared_block<128+2><<<num_blocks, n>>>(x0, x1, n);
blur_shared_block<128+2><<<num_blocks, n>>>(x0, x1, n);
```

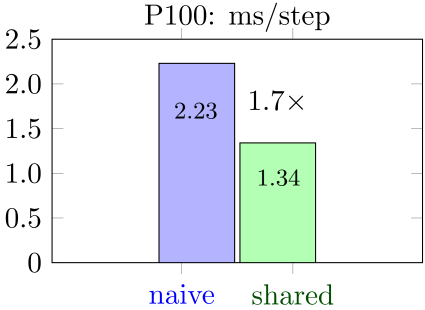
- Fusing these two operations will save us a round trip to global memory.

# 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, const double* 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);
  }
}
```

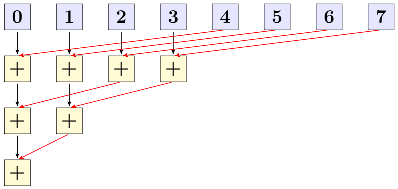
# Dissecting the speedup

- These types of kernels do very little math per loaded `double` from global memory
- Total runtime heavily dominated by the global memory bandwidth
  - Global memory BW: 500GB/s (P100), 1320GB/s (A100)
  - Shared memory BW: 8850GB/s (P100), 18140GB/s (A100)



# 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?



# Back to Cooperation

Cooperation in a GPU code can occur at multiple levels:

- Intra-block cooperation:
  - Between threads in a warp;
  - Between threads in thread block;
- Inter-block cooperation:
  - Between threads in grid;
  - Between threads in different kernels.

Synchronization might be required if more than one thread wants to modify (write) one of these shared resources.

# 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

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 functions, 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.

# Things to consider

- Atomics are slower than normal accesses:
  - Performance can degrade when many threads attempt atomic operations on few memory locations.
- Try to avoid or minimize 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.
- Further reading:
  - CUDA weakly-ordered memory model
  - Memory fence functions

# 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 $n$