Simulation and High-Performance Computing

Part 16: More GPU Computing

Steffen Börm

Christian-Albrechts-Universität zu Kiel

October 7th, 2020

Unified memory

Idea: Let the system handle data transfers between main and graphics memory.

Memory management:

- Memory is organized in pages, e.g., of 4 KB each.
- These pages logically reside in a unified memory space, but can physically be either in main or graphics memory.
- The system can move pages between main and graphics memory as required.
- This happens “behind the scenes”, but the user may notice performance impacts (“thrashing”).

Example: Gravitation with unified memory

```
error = cudaMallocManaged(&y1, sizeof(float) * n);
assert(error == cudaSuccess);
...

for(i=0; i<n; i++) {
  y1[i] = ... ;
  ...
}

grav<<<(n+255)/256, 256>>>(n, y1, y2, y3, m, phi);
cudaDeviceSynchronize();
```

- `cudaMallocManaged` allocates unified memory that can be used by both CPU and GPU.
- The `for`-loop accesses the array `y1` via the CPU.
- The kernel call accesses the same array via the GPU.

Nonblocking memory transfers

Challenge: While unified memory is very convenient, particularly on modern hardware, there are benefits to being able to manage memory transfers explicitly, e.g., to make them take place concurrently with actual computations.

Approach: Graphics cards can copy memory blocks in the background while both CPU and GPU are busy.

- Source or target blocks in main memory have to be pinned, i.e., the system has to be forbidden to move them.
- In order to allow the graphics card to perform computations and memory transfers concurrently, we have to be able to issue independent tasks.

Streams

Streams contain sequences of tasks like executing kernels or copying data that have to be performed in order, but independently from other streams.

```
cudaStream_t st[2];

cudaStreamCreate(st+1);

kernel<<<(n+255)/256, 256, 0, st[1]>>>(n, ...);

cudaStreamSynchronize(st[1]);

cudaStreamDestroy(st[1]);
```

Copy operations from pinned memory can also be added to streams:

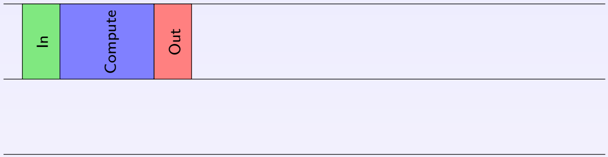
```
cudaMallocHost(&y1, sizeof(float) * n);

cudaMemcpyAsync(dev_y1, y1, sizeof(float) * n,
                cudaMemcpyHostToDevice, st[0]);
```

Example: Concurrent compute and copy operations

Idea: Use two streams. One computes while the other copies.

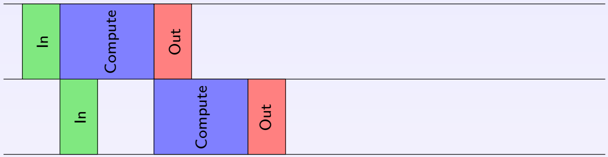
```
cudaStreamCreate(st+0);
cudaStreamCreate(st+1);
cur = 0;
for(i=0; i<n/chunk; i+=chunk) {
  cudaMemcpyAsync(in[cur], x+i, sizeof(float) * chunk,
                  cudaMemcpyHostToDevice, st[cur]);
  compute<<<chunk/256,256,0,st[cur]>>>(chunk, in[cur], out[cur]);
  cudaMemcpyAsync(y+i, out[cur], sizeof(float) * chunk,
                  cudaMemcpyDeviceToHost, st[cur]);
  cudaStreamSynchronize(st[1-cur]);
  cur = 1-cur;
}
cudaStreamSynchronize(st[1-cur]);
```

Interlaced streams



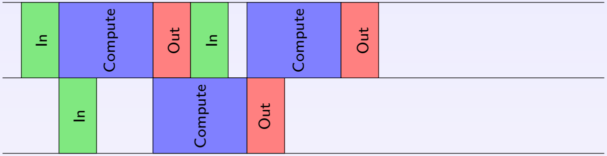
```
cur = 0;
for(i=0; i<n/chunk; i+=chunk) {
  cudaMemcpyAsync(in[cur], x+i, sizeof(float) * chunk,
                  cudaMemcpyHostToDevice, st[cur]);
  compute<<<chunk/256, 256, 0, st[cur]>>>(chunk, in, out);
  cudaMemcpyAsync(y+i, out[cur], sizeof(float) * chunk,
                  cudaMemcpyDeviceToHost, st[cur]);
  cudaStreamSynchronize(st[1-cur]);
  cur = 1-cur;
}
```

Interlaced streams



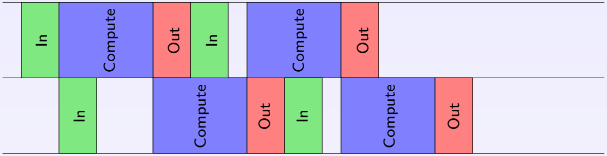
```
cur = 0;
for(i=0; i<n/chunk; i+=chunk) {
  cudaMemcpyAsync(in[cur], x+i, sizeof(float) * chunk,
                  cudaMemcpyHostToDevice, st[cur]);
  compute<<<chunk/256, 256, 0, st[cur]>>>(chunk, in, out);
  cudaMemcpyAsync(y+i, out[cur], sizeof(float) * chunk,
                  cudaMemcpyDeviceToHost, st[cur]);
  cudaStreamSynchronize(st[1-cur]);
  cur = 1-cur;
}
```

Interlaced streams



```
cur = 0;
for(i=0; i<n/chunk; i+=chunk) {
  cudaMemcpyAsync(in[cur], x+i, sizeof(float) * chunk,
                  cudaMemcpyHostToDevice, st[cur]);
  compute<<<chunk/256, 256, 0, st[cur]>>>(chunk, in, out);
  cudaMemcpyAsync(y+i, out[cur], sizeof(float) * chunk,
                  cudaMemcpyDeviceToHost, st[cur]);
  cudaStreamSynchronize(st[1-cur]);
  cur = 1-cur;
}
```

Interlaced streams



```
cur = 0;
for(i=0; i<n/chunk; i+=chunk) {
  cudaMemcpyAsync(in[cur], x+i, sizeof(float) * chunk,
                  cudaMemcpyHostToDevice, st[cur]);
  compute<<<chunk/256, 256, 0, st[cur]>>>(chunk, in, out);
  cudaMemcpyAsync(y+i, out[cur], sizeof(float) * chunk,
                  cudaMemcpyDeviceToHost, st[cur]);
  cudaStreamSynchronize(st[1-cur]);
  cur = 1-cur;
}
```

Thread blocks

Since threads in the same thread block are executed by the same multiprocessor, they can share data and synchronize.

Shared memory: Every multiprocessor is equipped with a small amount of very fast memory that is accessible for all threads in the current block.

```
__shared__ float xs[BLOCKDIM];
int ib = blockIdx.x * blockDim.x;
int it = threadIdx.x;

xs[it] = x[ib+it];
__syncthreads();
```

- This fragment copies `x[ib]` to `x[ib+BLOCKDIM-1]` to the shared-memory array `xs`.
- `__syncthreads()` provides a barrier that can only be passed once all threads have reached it.

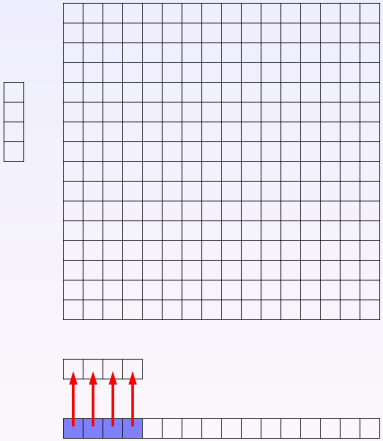
Example: Gravitation with shared memory

```
__global__ void
grav(int n, const float *x1, ...)
{
  __shared__ float x1s[BLOCKDIM];
  __shared__ float x2s[BLOCKDIM];
  __shared__ float x3s[BLOCKDIM];
  int ib = blockIdx.x * blockDim.x;
  int it = threadIdx.x;

  x1i = x1[i]; x2i = x2[i]; x3i = x3[i];
  for(jb=0; jb<n; jb+=BLOCKDIM) {
    x1s[it] = x1[jb+it];
    x2s[it] = x2[jb+it];
    x3s[it] = x3[jb+it];
    __syncthreads();
    /* ... compute potentials ... */
  }
}
```

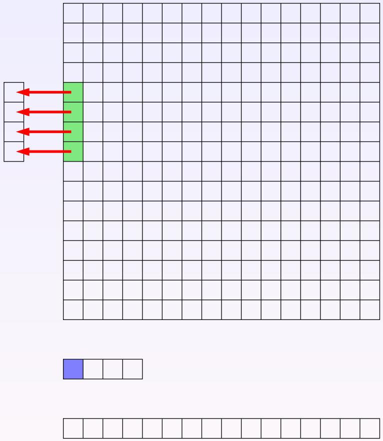
Example: Gravitation with shared memory

```
__global__ void
grav(int n, const float *x1, ...)
{
  ...

  for(jb=0; jb<n; jb+=BLOCKDIM) {
    /* ... copy to shared memory ... */
    for(jt=0; jt<BLOCKDIM; jt++) {
      d1 = x1s[jt] - x1i;
      d2 = x2s[jt] - x2i;
      d3 = x3s[jt] - x3i;
      if(ib+it != jb+jt)
        sum += 1.0 / sqrtf(d1*d1 + d2*d2 + d3*d3);
    }
    __syncthreads();
  }
}
```

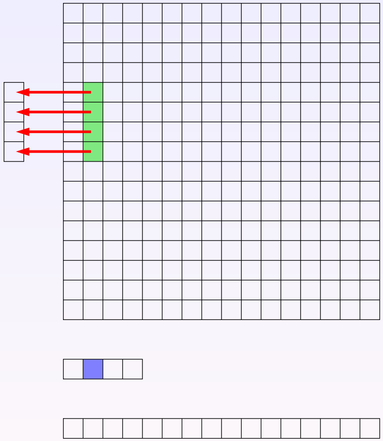
Example: Gravitation with shared memory



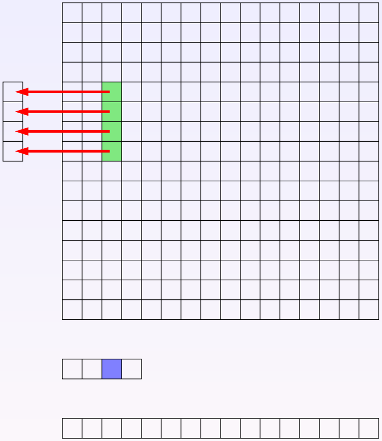
Example: Gravitation with shared memory



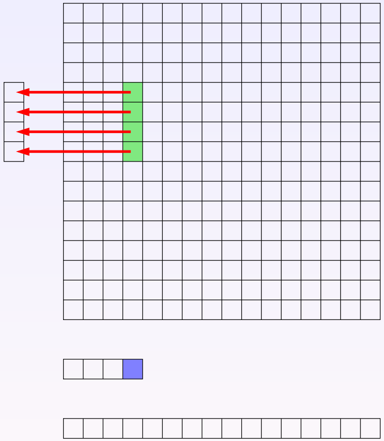
Example: Gravitation with shared memory



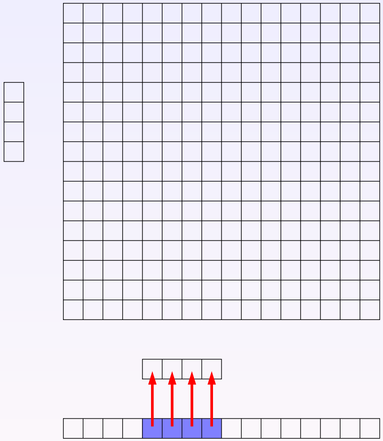
Example: Gravitation with shared memory



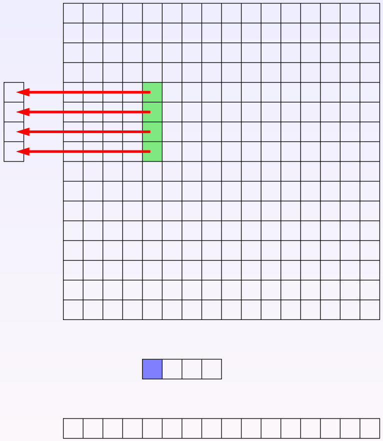
Example: Gravitation with shared memory



Example: Gravitation with shared memory



Example: Gravitation with shared memory



Experiment: Gravitation with shared memory

Goal: Compare direct-access and shared-memory implementation.

n	Direct	Shared
16 384	6.1	6.3
32 768	12.5	12.2
65 536	52.5	48.4
131 072	199.1	183.2
262 144	719.6	645.7
524 288	2 759.3	2 427.3

Result: In this example, shared memory leads only to moderate improvements.

Reduction

Goal: Accumulate contributions from all threads in the current block.

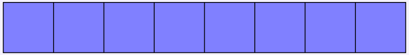
Idea: Use shared memory to exchange information between threads.

```
__shared__ float r[BLOCKDIM];
int it = threadIdx.x;
int k;

r[it] = value;
__syncthreads();

for(k=BLOCKDIM/2; k>0; k/=2) {
  if(it<k)
    r[it] += r[it+k];
  __syncthreads();
}

if(it == 0)
  result = r[0];
```

Reduction

Goal: Accumulate contributions from all threads in the current block.

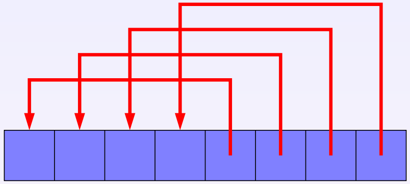
$k = 4$



```
for(k=BLOCKDIM/2; k>0; k/=2) {
  if(it<k)
    r[it] += r[it+k];
  __syncthreads();
}
```

Reduction

Goal: Accumulate contributions from all threads in the current block.



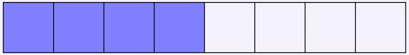
$k = 4$

```
for(k=BLOCKDIM/2; k>0; k/=2) {
  if(it<k)
    r[it] += r[it+k];
  __syncthreads();
}
```

Reduction

Goal: Accumulate contributions from all threads in the current block.

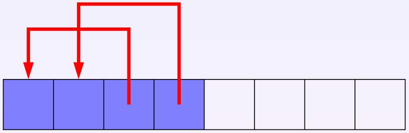
$k = 2$



```
for(k=BLOCKDIM/2; k>0; k/=2) {
  if(it<k)
    r[it] += r[it+k];
  __syncthreads();
}
```

Reduction

Goal: Accumulate contributions from all threads in the current block.



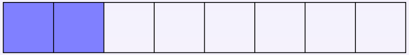
$k = 2$

```
for(k=BLOCKDIM/2; k>0; k/=2) {
  if(it<k)
    r[it] += r[it+k];
  __syncthreads();
}
```

Reduction

Goal: Accumulate contributions from all threads in the current block.

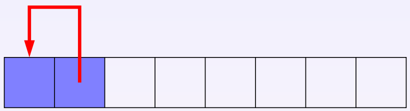
$k = 1$



```
for(k=BLOCKDIM/2; k>0; k/=2) {
  if(it<k)
    r[it] += r[it+k];
  __syncthreads();
}
```

Reduction

Goal: Accumulate contributions from all threads in the current block.



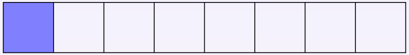
$k = 1$

```
for(k=BLOCKDIM/2; k>0; k/=2) {
  if(it<k)
    r[it] += r[it+k];
  __syncthreads();
}
```

Reduction

Goal: Accumulate contributions from all threads in the current block.

$k = 0$

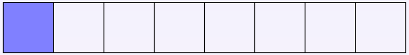


```
for(k=BLOCKDIM/2; k>0; k/=2) {
  if(it<k)
    r[it] += r[it+k];
  __syncthreads();
}
```

Reduction

Goal: Accumulate contributions from all threads in the current block.

$k = 0$



```
for(k=BLOCKDIM/2; k>0; k/=2) {
  if(it<k)
    r[it] += r[it+k];
  __syncthreads();
}
```

Advantage: Every step halves the size of the array.
$\rightarrow$ Even large arrays can be handle efficiently.

Summary

Unified memory allows us to avoid explicitly copying data between main memory and graphics memory.

Streams allow us to “hide” the latencies of data transfer by performing computations concurrently.

Shared memory can be used to speed up computations where all threads in a block need the same data from graphics memory.
It is also useful for exchanging information between the threads in a block.