

GPU programming in CUDA: Advanced topics in CUDA

Lukas Einkemmer

Department of Mathematics
University of Innsbruck

PRACE Autumn School, Innsbruck

Link to slides: <http://www.einkemmer.net/training.html>

Shared memory

Reminder on Caches

Cache is a type of **fast, but small, memory** that accelerates repeated access to the same memory location.

- ▶ Usually, i.e. on CPUs, completely transparent to the programmer.

On the GPU we can explicitly control the L1 cache.

- ▶ **Shared memory.**

Often essential to obtain good performance for memory bound problems.

Shared memory

The `__shared__` keyword declares a variable/array that resides in shared memory.

Such variables are **shared among the threads in a block**.

- ▶ No communication between blocks is possible using shared memory.

Example using shared memory

```
__global__ void k_stencil(double* x, double* y, int n) {
    int i = threadIdx.x + blockDim.x*blockIdx.x;

    // Local_x is an array in shared memory.
    __shared__ double local_x[1024];

    // Each thread loads its value of x into local_x that is shared
    if(i < n) local_x[threadIdx.x] = x[i];

    __syncthreads();

    if(threadIdx.x > 0 && threadIdx.x < blockDim.x-1)
        y[i] = local_x[threadIdx.x+1]-local_x[threadIdx.x-1];
    else if(threadIdx.x == 0 && i > 0)
        y[i] = local_x[threadIdx.x+1]-x[i-1];
    else if(threadIdx.x == blockDim.x-1 && i < n-1)
        y[i] = x[i+1] - local_x[threadIdx.x-1];
}
```

Synchronization

The `__syncthreads()` function acts as a barrier for all threads in a block.

- ▶ Threads in different blocks are not affected.

General philosophy: Threads in the same block can synchronize and exchange data (via shared memory).

- ▶ No synchronization between blocks in a single kernel.

Recommendation: If you need to synchronize between blocks rethink your work and data distribution.

- ▶ You might have a problem that does not map very well to the GPU hardware.

Dynamic shared memory

What about dynamically allocating shared memory?

```
__global__ void k_sum(double* x, double* out, int n) {  
    extern __shared__ char shared_mem[];  
}
```

```
k_sum<<<num_blocks, num_threads, shared_mem_size>>>(d_x, d_out, n);
```

The argument `shared_mem_size` specifies the size (in bytes) of the array `shared_mem`.

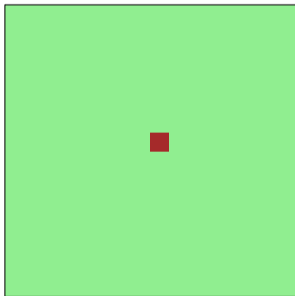
Only one such dynamic shared memory block is allowed per kernel.

Matrix-matrix multiplication

Matrix-matrix multiplication

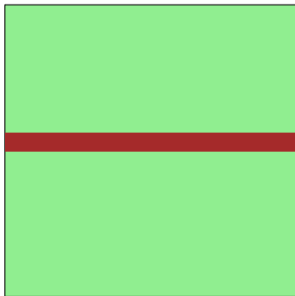
$$C_{i,j} = \sum_k A_{i,k} B_{k,j}.$$

C

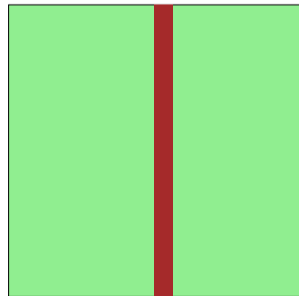


=

A



B



Straightforward implementation

Each thread computes one element of the output matrix $C_{i,j}$.

```
__global__  
void matmul(long n, double* A, double* B, double* C) {  
    long i = blockIdx.x*blockDim.x + threadIdx.x;  
    long j = blockIdx.y*blockDim.y + threadIdx.y;  
  
    double val=0.0;  
    for(long k=0;k<n;k++)  
        val += A[i+k*n]*B[k+j*n];  
  
    C[i+j*n] = val;  
}
```

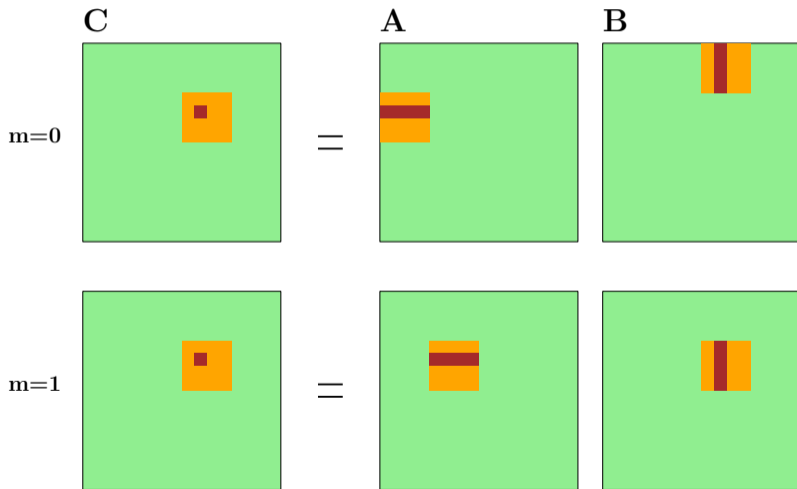
On the V100 (n=8192) we obtain 2.2 TFLOPS.

- ▶ Theoretical peak of 15 TFLOPS.
- ▶ Arithmetic operations $\mathcal{O}(n^3)$ vs memory accesses $\mathcal{O}(n^2)$.

Better algorithm

Each block computes a submatrix.

- ▶ Data loaded once and then stored in shared memory.



Implementation

```
__global__
void matmul_fast(long n, double* A, double* B, double* C) {
    long i = threadIdx.x; long bi = blockIdx.x;
    long j = threadIdx.y; long bj = blockIdx.y;

    // loop over all sub-matrices
    double val = 0.0;
    for(long m=0;m<n/BS;m++) {
        __shared__ double block_A[BS*BS];
        __shared__ double block_B[BS*BS];

        // load block into shared memory
        block_A[i+BS*j] = A[bi*BS+i + n*(m*BS+j)];
        block_B[i+BS*j] = B[m*BS+i + n*(bj*BS+j)];

        // wait until all threads have caught up
        __syncthreads();
    }
}
```

Implementation

```
// compute the (sub-)matrix-matrix product
for(long k=0;k<BS;k++)
    val += block_A[i+BS*k]*block_B[k+BS*j];

// make sure that all threads are finished before
// next loop iteration starts.
__syncthreads();
}

// update result in global memory
C[bi*BS+i + n*(bj*BS+j)] = val;
}
```

On the V100 (n=8192) we obtain 4.2 TFLOPS.

- ▶ Improvement by approximately a factor of two.

Exercise

Given a vector v in GPU memory compute $\sum_i v_i$.

- ▶ Start from `exercise-advanced-einkemmer.cu`.

Use shared memory to perform the sum in each block.

- ▶ Each block writes its (local) result to global memory.
- ▶ Repeat this procedure until you obtain the entire sum.

Once your code produces the correct result, time your code and report its performance.

Exercise

Run the implementation of the **matrix-matrix multiplication** (`matmul.cu`) on different GPUs.

- ▶ What do you observe?

Some interesting results:

- ▶ Performance on V100 and A100 for the *naive* implementation is the same.
- ▶ But, the shared memory implementation is faster on the A100 (5 TFLOPS).