

Accelerator-Based Programming

-

CUDA – part 2

Jörn Zimmerling

September 7, 2022



UPPSALA
UNIVERSITET

Host and device memory

- Separate memory management by `cudaMalloc` and `cudaFree` leads to clear responsibilities
- Software must manage data transfer by `cudaMemcpy`
- Messy for more elaborate data structures (pointers, indirections)
- Coordination between CPU and GPU in `cudaMemcpy`
- A memory address only makes sense within either the CPU (host memory) or the GPU (device memory)
 - Access to wrong memory class invalid!



Unified memory

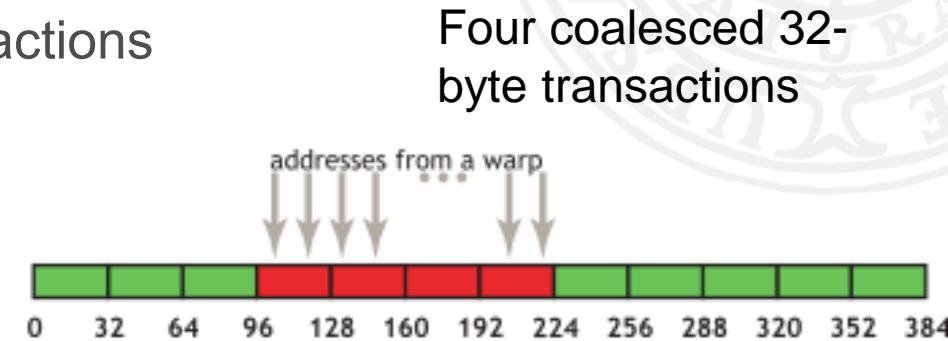
- Newer architectures support unified memory, i.e., accessing the same pointer from both host and device
 - See also <https://developer.nvidia.com/blog/parallelforall/unified-memory-in-cuda-6/>
- With CUDA: `cudaMallocManaged` allocates in common address space
- Unified memory pinned, no swapping to disk
- Simpler entry point to GPU programs
- Developer can concentrate on CUDA kernels
- But the copying still has to happen physically
 - Data locality important: Must do many operations on GPU before GPU makes next access
 - Copying should happen well in advance of use
 - Need `cudaDeviceSynchronize()` call
- Unified memory is tradeoff between simplicity and performance – trade specialization like e.g. `cudaMemcpyAsync` for transparency



Memory usage on a GPU

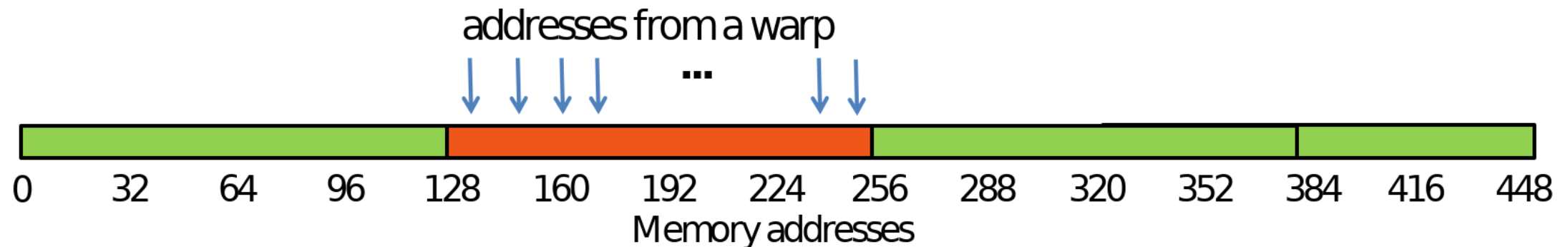
Global memory access in GPU memory:

- Memory accessed via 32-, 64-, or 128-byte memory transactions
- Threads in **warp** access memory together
- Hardware minimizes number of transactions



Coalesced access 1:

- Threads read contiguous memory – single transaction
- Memory address aligned by size of transaction

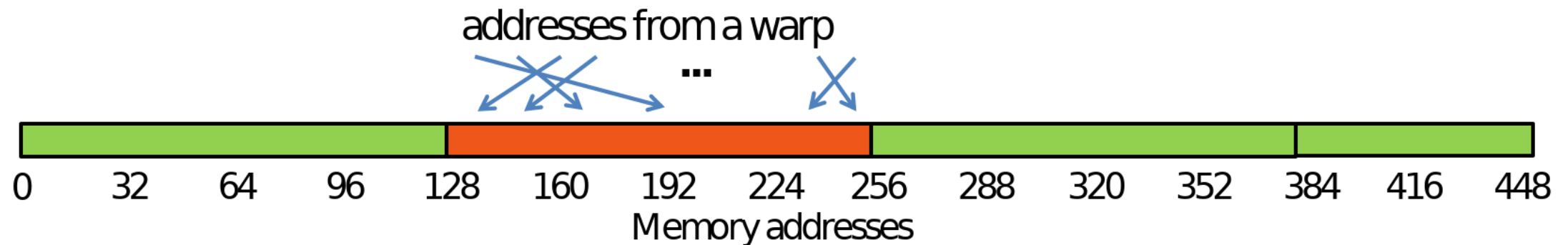


Global memory access in GPU memory:

- Memory accessed via 32-, 64-, or 128-byte memory transactions
- Threads in **warp** access memory together
- Hardware minimizes number of transactions

Coalesced access 1:

- Contiguous but permuted access – single transaction
- Access by the form `x[indices[threadIdx]]` with indirect addressing index array `indices`

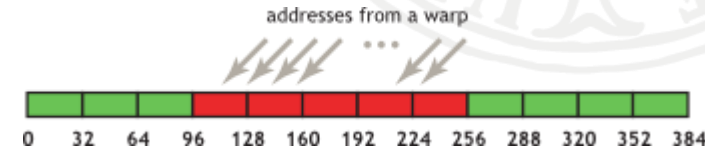


Memory usage on a GPU

Global memory access in GPU memory:

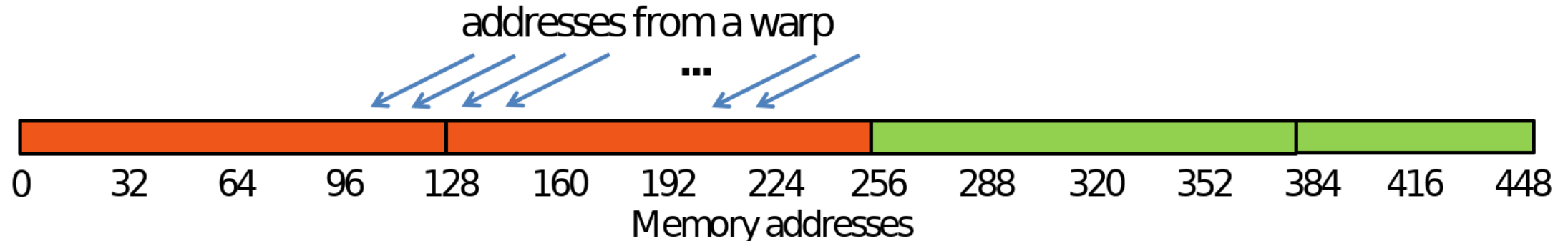
- Memory accessed via 32-, 64-, or 128-byte memory transactions
- Threads in **warp** access memory together
- Hardware minimizes number of transactions

Five coalesced 32-byte transactions



Uncoalesced access 1:

- Contiguous but misaligned – two transactions
- E.g. 128 byte transaction, address not divisible by 128



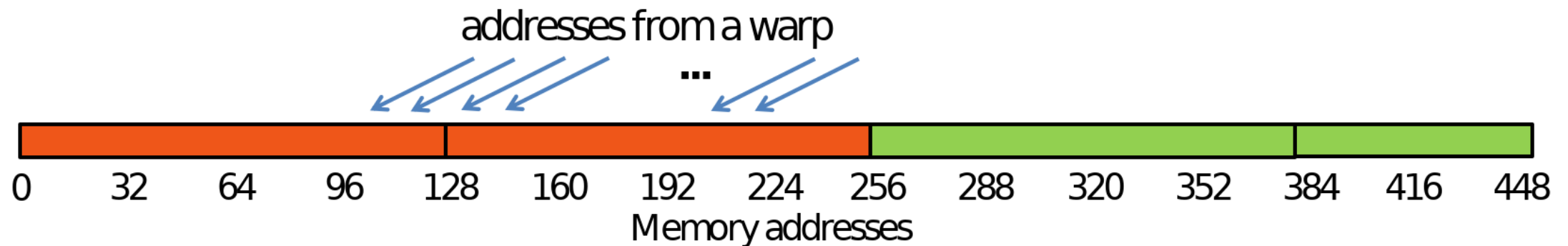
Memory usage on a GPU

Global memory access in GPU memory:

- Memory accessed via 32-, 64-, or 128-byte memory transactions
- Threads in **warp** access memory together
- Hardware minimizes number of transactions

Uncoalesced access 1:

- Contiguous but misaligned – two transactions
- E.g. 128 byte transaction, address not divisible by 128



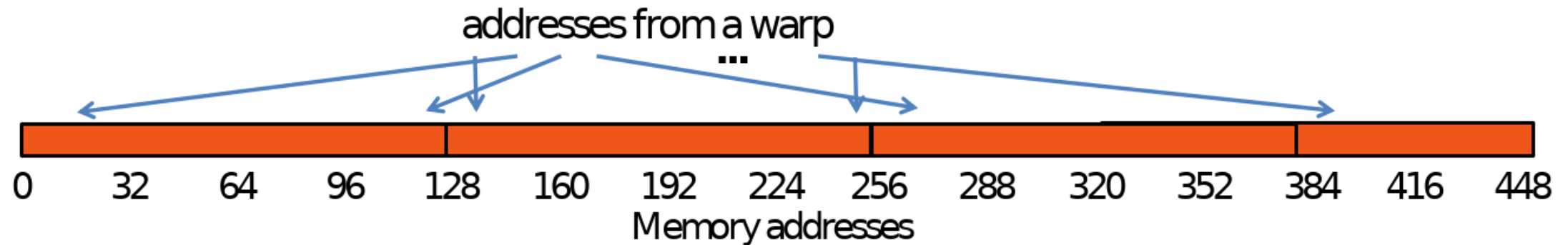
Memory usage on a GPU

Global memory access in GPU memory:

- Memory accessed via 32-, 64-, or 128-byte memory transactions
- Threads in **warp** access memory together
- Hardware minimizes number of transactions

Uncoalesced access 2:

- Non-contiguous access – N transactions
- Note: Caches help to reduce transactions to memory, but need temporal locality



Further reading:

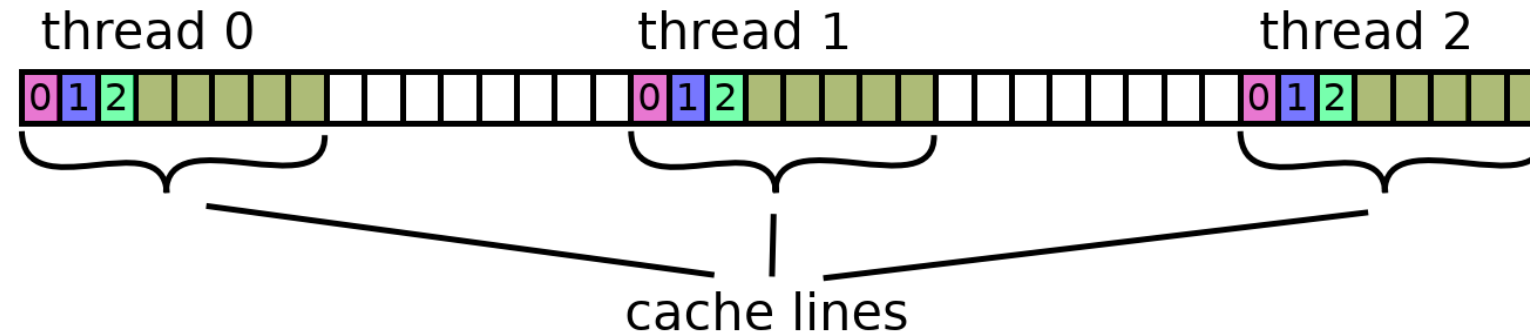
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

→Section 5.3.2, Device Memory Accesses

Comparison between CPU and GPU memory access

CPU:

- Local caching for each thread
- Locality for each thread
- Avoid same “cache line” on different threads/cores (ping-pong, false sharing)'



GPU:

- Threads within an SM prefer nearby access
- Opposite access pattern – GPU threads and CPU threads behave differently



Array-of-Structure vs Structure-of-Array

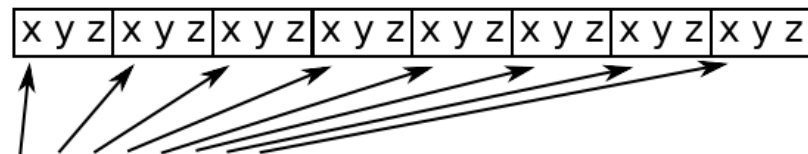
- Collection of 3D Points

```
/* Array of Structure */
typedef struct {
    float x,y,z;
} aos_t;
aos_t aos[1000];

/* access */
float xval = aos[i].x;
float yval = aos[i].y;
float zval = aos[i].z;
```

Array-of-Structure good on CPU

- Values of point on same cache line
- All data from same stream
- Easy to prefetch when walking over it

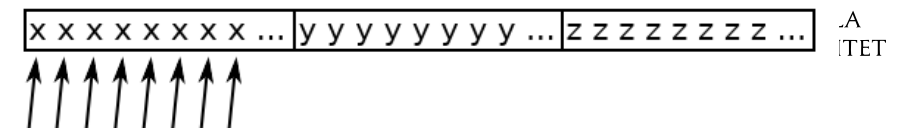


```
/* Structure of Array */
typedef struct {
    float x[1000];
    float y[1000];
    float z[1000];
} soa_t;
soa_t soa;

/* access */
float xval = soa.x[i];
float yval = soa.y[i];
float zval = soa.z[i];
```

Structure-of-Array good for GPU

- Contiguous access for multiple threads
- Parallelism across points
- (Preferable for SIMD on CPU, too)



Shared memory on the GPU

- Small and very fast memory shared by thread block
- E.g. user-managed cache, or scratchpad for cooperative algorithm
- Programmer responsible for avoiding race conditions

Use:

```
__shared__ int buffer[SIZE]
```

- Automatic L1 cache
- Since 2010 (Fermi architecture)
- Configurable, 16kB+48kB / 48kB+16kB INT/FP32




Access of shared memory: banks

Shared memory banks

- 32 banks
- Consecutive 4B-words belong to different banks, cyclically

Bank conflicts:

- Penalty if threads in warp access same bank
- Access into same bank serialized



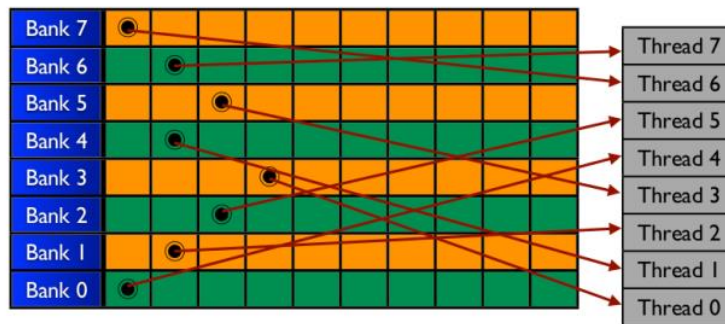
Bank 7	7	15	23							
Bank 6	6	14	22							
Bank 5	5	13	21							
Bank 4	4	12	20							
Bank 3	3	11	19							
Bank 2	2	10	18							
Bank 1	1	9	17							
Bank 0	0	8	16							



Access of shared memory: possible bank conflicts

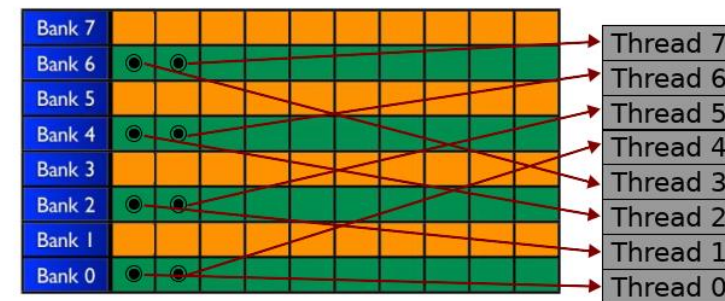
All different banks:

- No conflicts



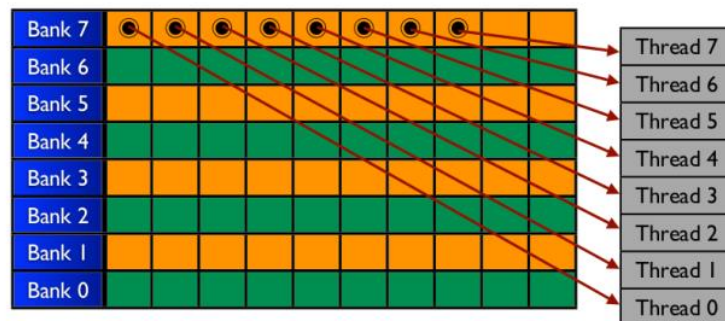
2-way bank conflict:

- Half performance



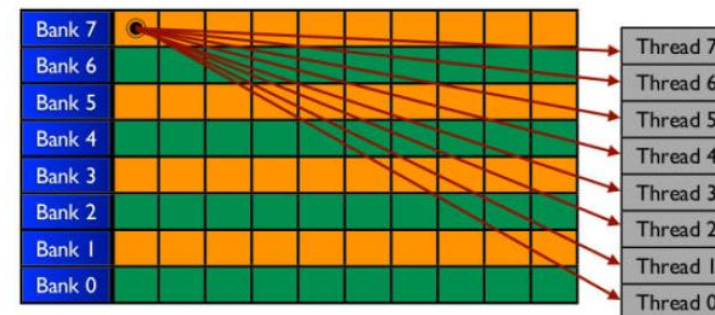
Full bank conflict:

- 1/8 performance



Broadcast:

- Threads access *same location* – no overhead



Memory access problems by type of memory

- Bank conflicts: transfer between GPU cores and **shared memory**
- Uncoalesced access: transfer between GPU cores and **global device memory**
- Slow PCI-e bus: **transfer** between GPU and host



Synchronization in CUDA

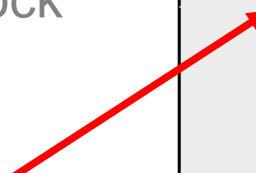
- Global synchronization across full GPU does not exist
- Implicit global synchronization at kernel boundaries
 - A kernel does not return anything
 - Synchronization: wait for kernel on CPU, send out next kernel, etc.
 - Feed data for iteration $i + 1$ while kernel for iteration i is running

Synchronization within block:

- `__syncthreads()`
- Barrier for threads in block
 - When one thread calls `__syncthreads()`, all threads in block must do so
- Avoid data race when using `__shared__` memory

Example: array reversal

```
__global__ void reverse(int *d,
                        int n)
{
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```



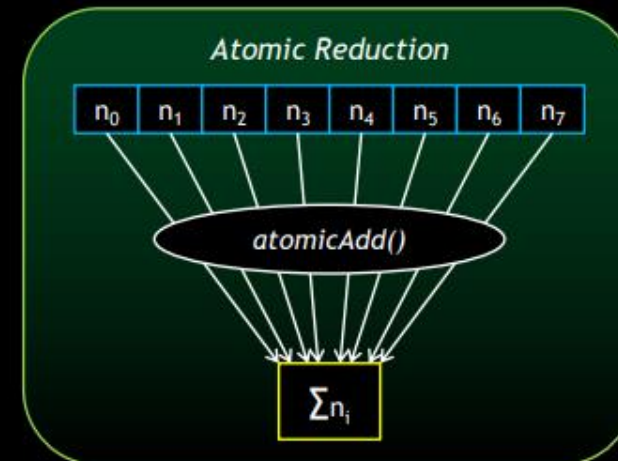
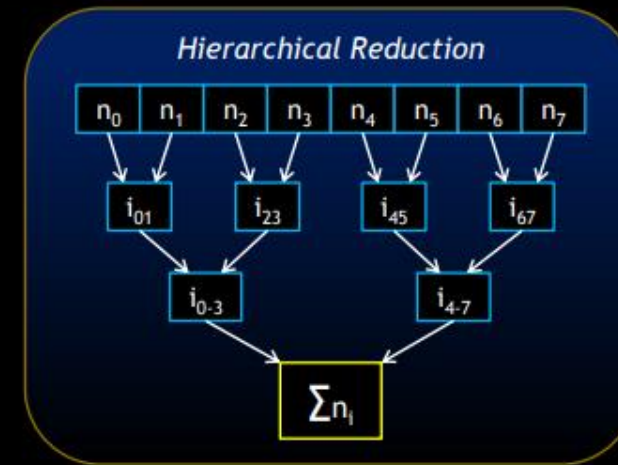
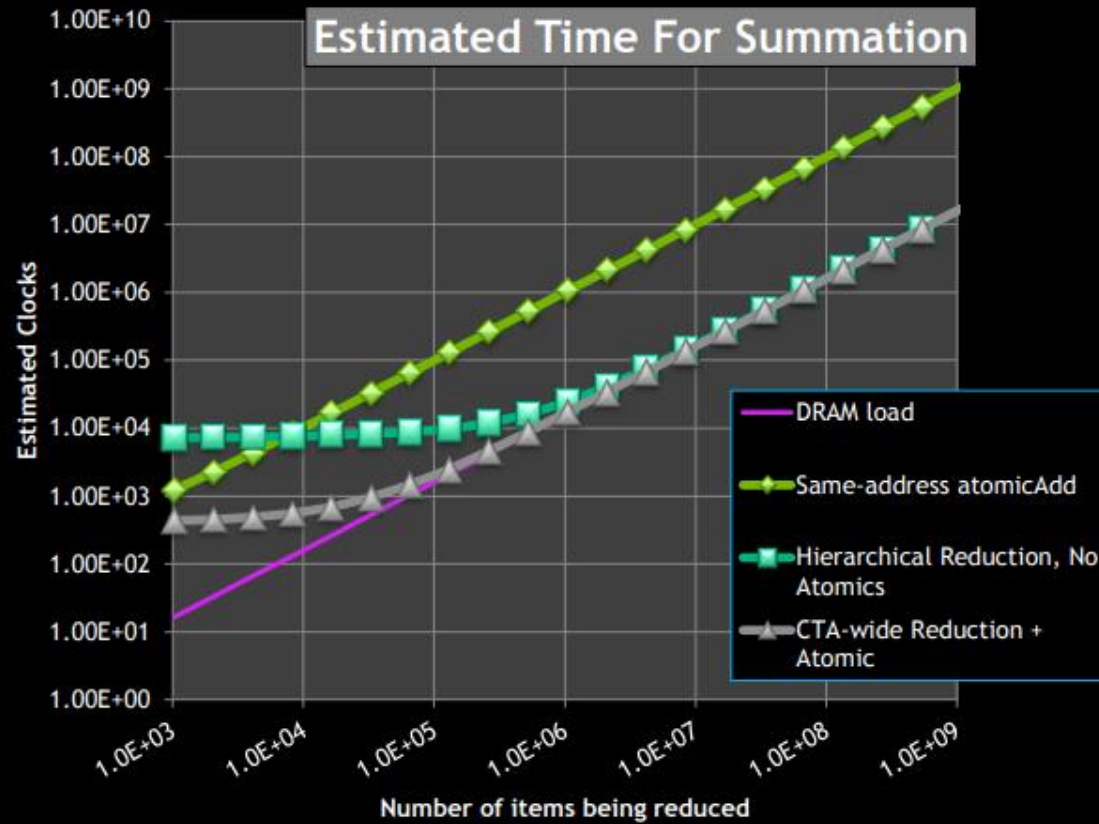
Atomic operations in CUDA

- Perform a combined read/write operation in thread-safe manner
- `atomicAdd`, `atomicSub`, `atomicMax`, etc
- Typically resolved by cache system
 - Instruction returns immediately
 - Conflict resolution on store
 - Fire-and-forget semantic
- Besides device-wide atomics, there are also
 - block-wide atomics (suffix `_block`, e.g. `atomicAdd_block`)
 - system-wide atomics across all GPUs and CPUs (suffix `_system`)
- Uninterruptable read-modify-write memory operation



But why? – Reduction at RAM-load speed

Atomic Arithmetical Operations



Parallel reductions

- Vector summations, min/max important algorithms (dot product, norms, means, . . .)

```
sum=0;
for(int i=0; i<N; ++i)
    sum += x[i];
```

- See first exercise for CUDA
- Uses concepts of shared memory, synchronization and atomic operations due to concurrent access
- Presentation: "Optimizing Parallel Reduction in CUDA", Mark Harris
<http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>



Example algorithm: Gaussian blur

Blur an image:

- Average neighboring 5x5 pixel values

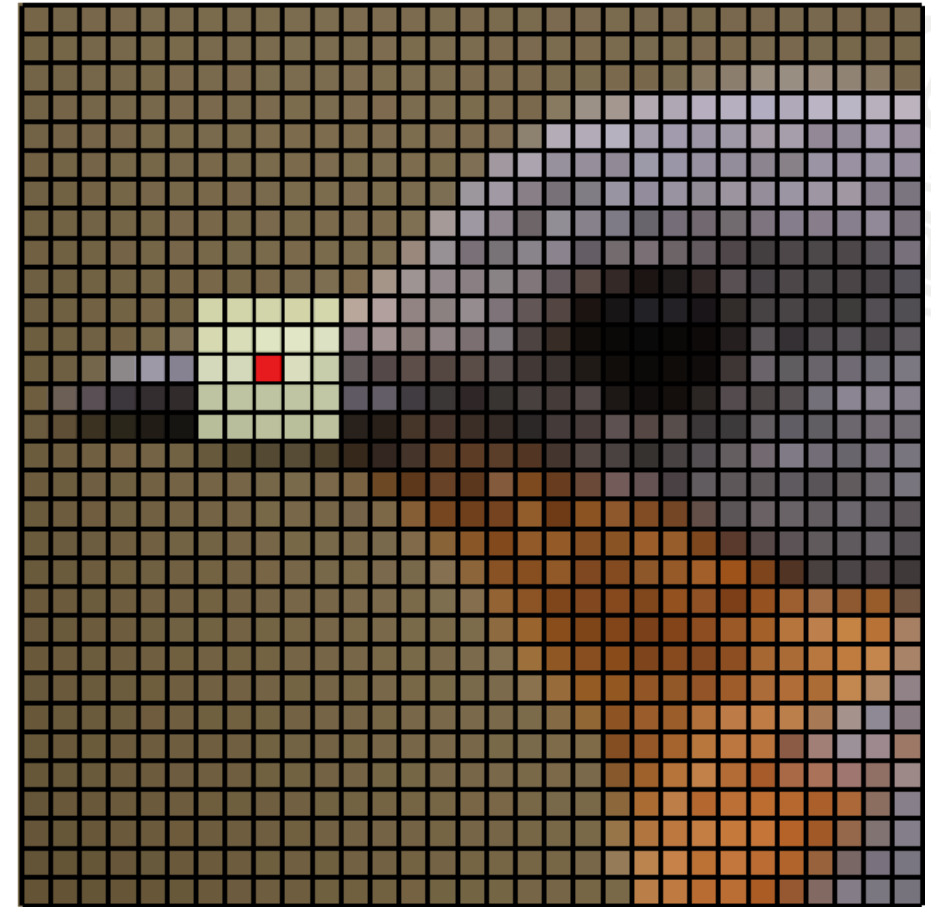
Algorithm C:

```
for(int i=2; i<Height-2; ++i)
  for(int j=2; j<Width-2; ++j) {
    float tmp=0;
    for(int ii=-2; ii<=2; ++ii)
      for(int jj=-2; jj<=2; ++jj)
        tmp +=
          Weight[(ii+2)*5+(jj+2)]*
          Ain[(i+ii)*Width+j+jj];
    Aout[i*Width + j] = tmp;
  }
```

Idea:

- All pixels are independent
- Both i and j loops are perfectly parallel

Explore this in Assignment 3 to simulate Game-of-Life



Example algorithm: Gaussian blur

- On the GPU, want as many threads as possible
- One thread per pixel
- 2D blocks and grid

Kernel code:

```
__global__  
void blur_kernel(int width, int height, float *W,  
                 float *Ain, float *Aout)  
{  
    int i = threadIdx.x + blockIdx.x*blockDim.x;  
    int j = threadIdx.y + blockIdx.y*blockDim.y;  
  
    if( (i>1 && i<height-2) && (j>1 && j<width-2) )  
    {  
        float tmp = 0;  
  
        for(int ii=-2; ii<=2; ++ii)  
            for(int jj=-2; jj<=2; ++jj)  
                tmp += W[(ii+2)*5+(jj+2)]*Ain[(i+ii)*width+j+jj];  
  
        Aout[i*width + j] = tmp;  
    }  
}
```



Example algorithm: Gaussian blur

- Host code to run the Gaussian blur filter:

```
// W, Ain, Aout are device arrays

// 2D grid and blocks
int nblocks_w = 1 + (width-1)/block_size; // int division
int nblocks_h = 1 + (height-1)/block_size; // rounding up
dim3 gd_dim(nblocks_h, nblocks_w);
dim3 bk_dim(block_size, block_size);

blur_kernel<<<gd_dim, bk_dim>>>(width, height, W, Ain, Aout);
```



Example algorithm: Matrix-matrix product

$$C_{ij} = \sum_{k=1}^N A_{ik} B_{kj}$$

Code:

```
for(int i=0; i<N; ++i)
  for(int j=0; j<N; ++j)
    for(int k=0; k<N; ++k)
      C[i*N+j] += A[i*N+k]*B[k*N+j]
```



Example algorithm: Matrix-matrix product

$$C_{ij} = \sum_{k=1}^N A_{ik} B_{kj}$$

Code:

```
for(int i=0; i<N; ++i)
  for(int j=0; j<N; ++j)
    for(int k=0; k<N; ++k)
      C[i*N+j] += A[i*N+k]*B[k*N+j]
```

Idea:

- Computation of the elements in C are independent
- Both i and j loops are perfectly parallel
- Parallelize both loops
- Apply ideas to second assignment!



Example algorithm: Matrix-matrix product

Parallelization:

- On GPU, want many threads
- Use one thread per element in C

Kernel code:

```
__global__  
void matmul(int N, float *A, float *B, float *C)  
{  
    int i = threadIdx.x + blockIdx.x*blockDim.x;  
    int j = threadIdx.y + blockIdx.y*blockDim.y;  
  
    if(i < N && j < N)  
        for(int k=0; k<N; ++k)  
            C[i*N+j] += A[i*N+k]*B[k*N+j];  
}
```

Invocation:

```
const int n_blocks = 1+(N-1)/block_size;  
dim3 grid_dim(n_blocks, n_blocks); // grid and  
dim3 block_dim(block_size, block_size); // block dimensions  
  
matmul<<<grid_dim, block_dim>>>(N, A_dev, B_dev, C_dev);
```



Example: Matrix-matrix product

```
__global__ void matmul(int N, float *A, float *B, float *C)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int j = threadIdx.y + blockIdx.y*blockDim.y;

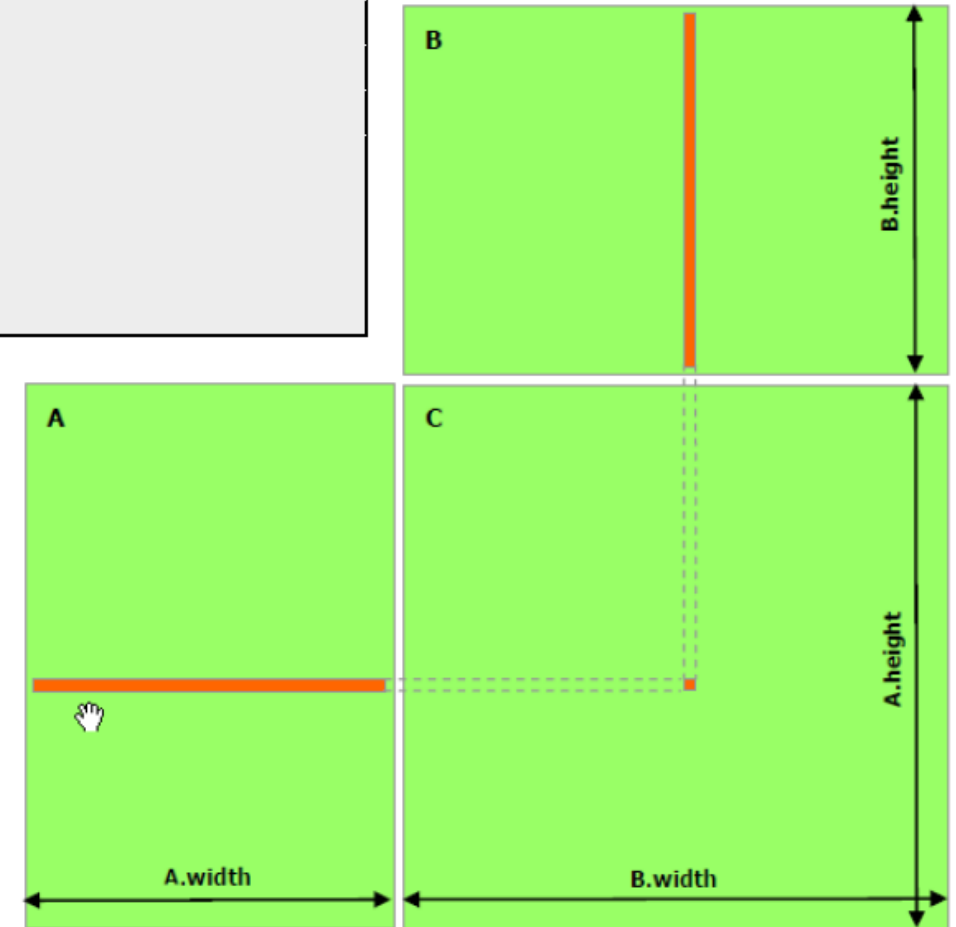
    if(i < N && j < N)
        for(int k=0; k<N; ++k)
            C[i*N+j] += A[i*N+k]*B[k*N+j];
}
```

Problem:

- All threads with same i re-read whole i 'th row of A
- Memory bandwidth wasted

Ideas for improvement

- Cache data in shared memory
- Change data layout to improve coalescing

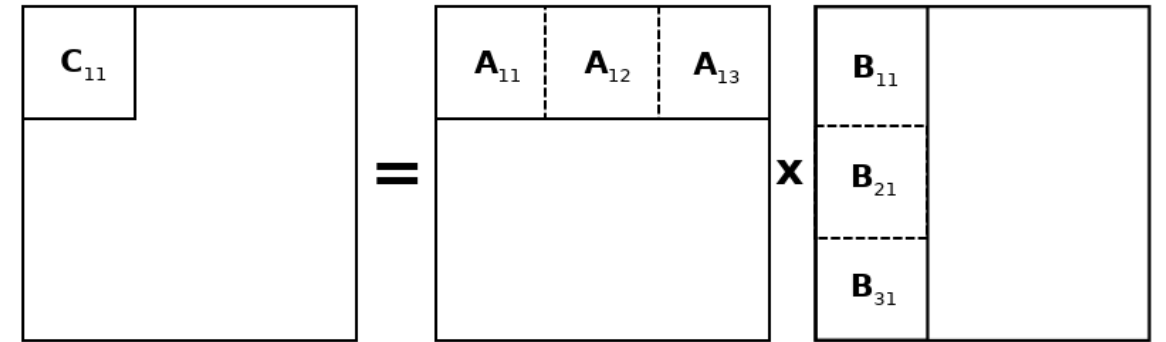


Example: Matrix-matrix product

Tiled approach:

- E.g., 3x3 tiles
- Top left block:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$$



Algorithm

- Load A_{11} and B_{11} into shared buffers
- Compute contribution from tiles
- Repeat with A_{12} and B_{21}
- Repeat with A_{13} and B_{31}

Motivation

- Only have to read each tile from global memory once per block



Example: Matrix-matrix multiplication

Possible kernel code:

```
--global__ void matmul(int N, float *A, float *B, float *C)
{
    int i = threadIdx.x; int j = threadIdx.y;
    int iglob = threadIdx.x + blockIdx.x*blockDim.x;
    int jglob = threadIdx.y + blockIdx.y*blockDim.y;
    int ntiles = gridDim.x; // assume square grid
    // shared buffers
    __shared__ float atile[block_size][block_size];
    __shared__ float btile[block_size][block_size];

    for(int t=0; t<ntiles; ++t)
    {
        // load t'th tile of A and B into atile and btile
        atile[i][j] = /* element [i,j] of tile [blockIdx.x,t] of A */
        btile[i][j] = /* element [i,j] of tile [t,blockIdx.y] of B */
        // ensure data is ready
        __syncthreads();
        for(int k=0; k<block_size; ++k)
            C[iglob*N+jglob] += atile[i][k]*btile[k][j];
        // ensure all threads are done with data
        __syncthreads();
    }
}
```



What can be done and cannot be done in CUDA

Possible with CUDA

- Memory allocation
- Recursive function calls
- Polymorphism/virtual methods (object on GPU)
- Call other CUDA kernels
- Some header-based C++ libraries

Impossible with CUDA

- C++ exception handling
 - Even if it compiles, will not throw properly
 - Cannot catch exceptions
- Exceptions also bad for performance in CPU programming
 - Might want to use them as little as possible anyway
- Parts of C++ library, very new C++ features



Performance optimization

Parallelism

- Use many small threads: ~5000 cores, 50k+ threads needed!
- Maximize utilization
- Avoid branch divergence, try to use same instruction for all 32 threads in a warp

Memory access

- Coalesced memory access
- Use shared memory – avoid bank conflicts

Data movement

- Avoid frequent host-device transfer
- Keep data on GPU for several phases
- See also NVIDIA's document

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

- →PerformanceGuidelines



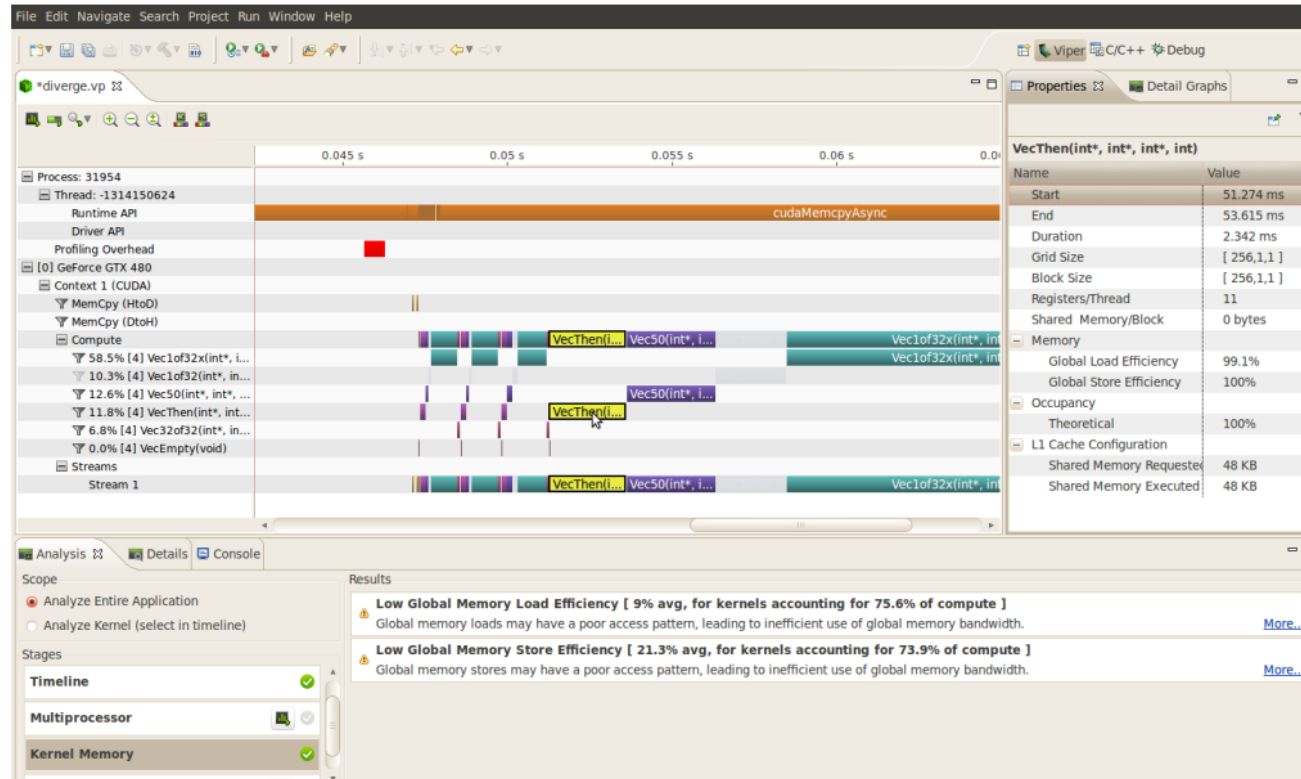
CUDA tools and libraries

Visual profiler

- Shows performance characteristics (cache usage, occupancy, compute/communication overlap, etc)
- Plugin for Eclipse / Visual studio

Also:

- Debugger `cuda-gdb`
- Memory checker
- `cuda-memcheck`



CUDA tools and libraries

- cuBLAS – dense linear algebra
- cuSPARSE – sparse linear algebra
- cuFFT – Fourier transforms etc
- Thrust – parallel data structures and algorithms – similar to C++ STL
<https://docs.nvidia.com/cuda/thrust/index.html>

Avoid writing CUDA code

```
// generate random data on the host
thrust::host_vector<int> h_vec(100);
// initialize h_vec with your own function init()
init(h_vec);
// transfer to device and compute sum
thrust::device_vector<int> d_vec = h_vec;

// binary operation used to reduce values
thrust::plus<int> binary_op;
// compute sum on the device
int sum = thrust::reduce(d_vec.begin(), d_vec.end(),
                        0, binary_op);
```



Finite element application

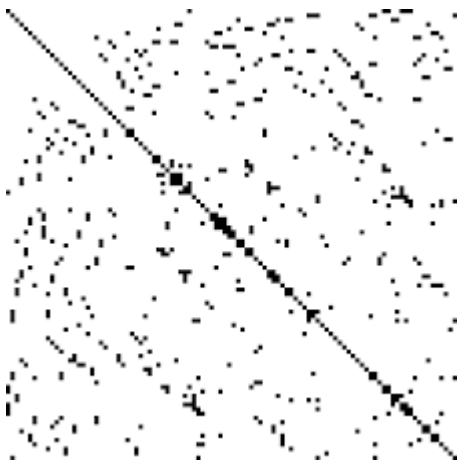
- We are using Finite-Element to solve the Laplace equation

$$\Delta u(x, y, z) = f(x, y, z)$$

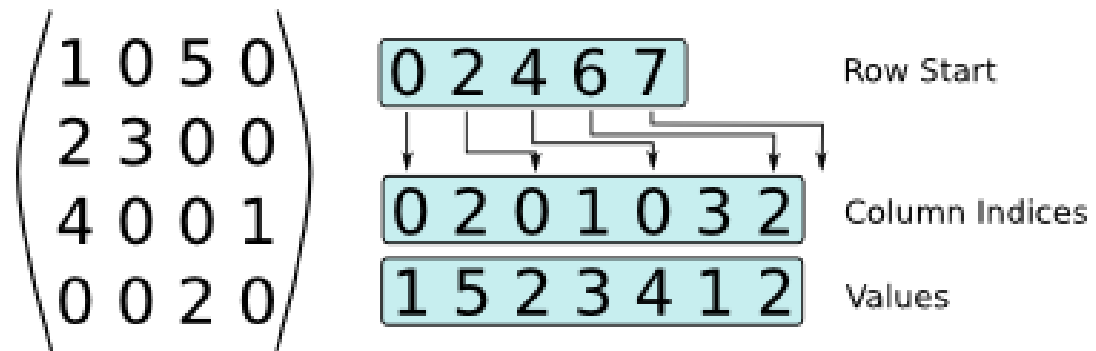
- You are provided a hexahedral meshing in three dimensions. After discretization we obtain linear system

$$Ax = b$$

with a sparse matrix which we store in CRS form Compressed Row Storage.



Example: CSR Storage

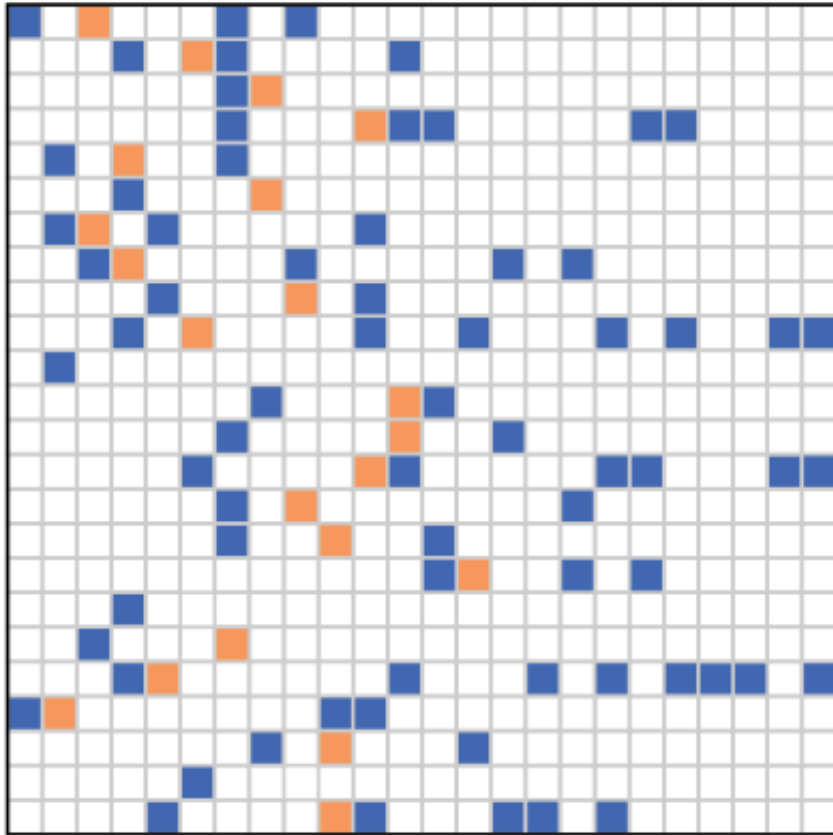


MORITZ KREUTZER[†], GEORG HAGER[†], GERHARD WELLEIN[†], HOLGER FEHSKE[‡],
AND ALAN R. BISHOP[§]

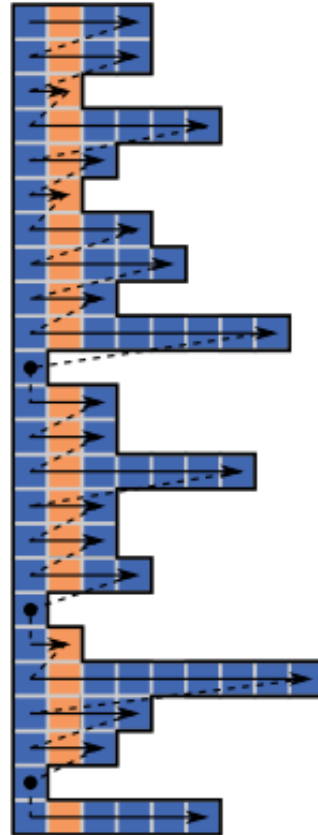
Abstract. Sparse matrix-vector multiplication (spMVM) is the most time-consuming kernel in many numerical algorithms and has been studied extensively on all modern processor and accelerator architectures. However, the optimal sparse matrix data storage format is highly hardware-specific.



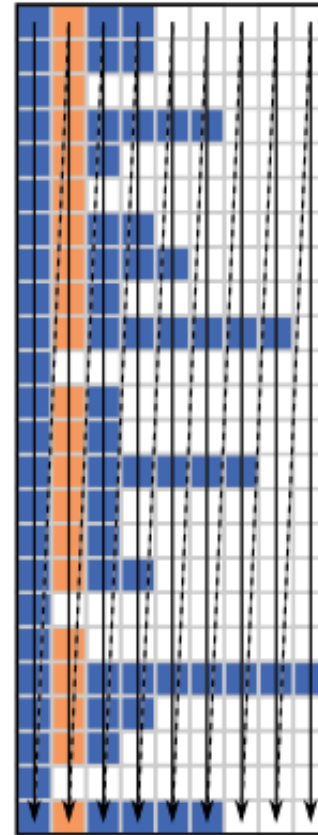
Standard Sparse matrix storing



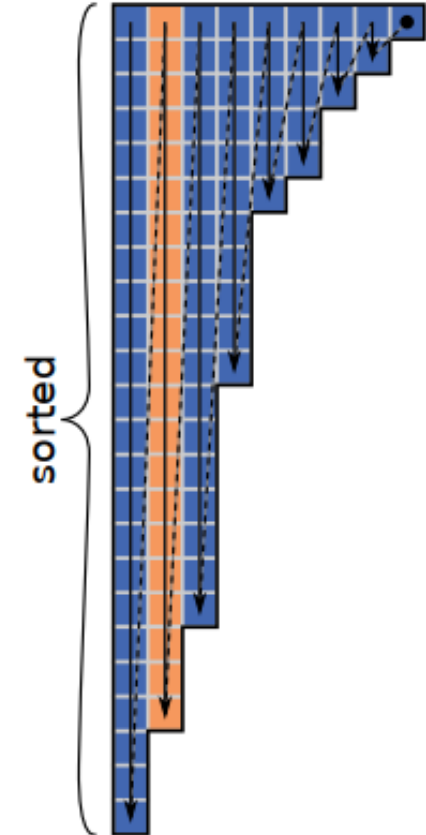
(a) Source matrix



(b) CRS

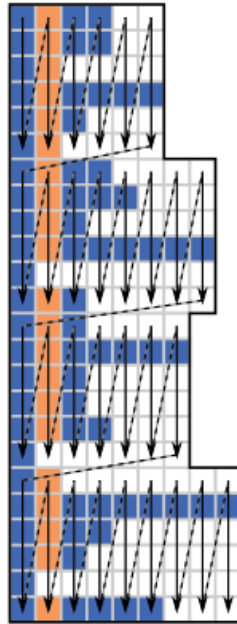


(c) ELLPACK

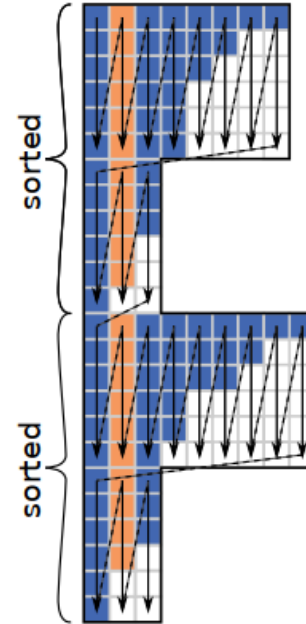


(d) JDS

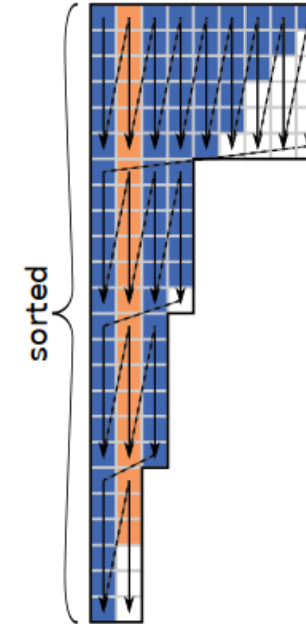




(a) SELL-6-1,
 $\beta = 0.51$



(b) SELL-6-12,
 $\beta = 0.66$



(c) SELL-6-24,
 $\beta = 0.84$

FIG. 2. Variants of the SELL-C- σ storage format for the matrix structure in Figure 1(a). Arrows indicate the storage order of matrix values and column indices.

C= block size, sigma = sorting length.
(We use C=32, s=1)



Tasks for a 3

- **provide a correct program** for the basic tasks involving the CRS matrix both on the CPU and the GPU,
- provide **computational measurements** of acceptable performance on both platforms and display them in an understandable manner,
- provide **explanations** for the expected performance of the sparse matrix-vector product and conjugate gradient solver with basic CRS matrix storage as well as the improved ELLPACK/SELL-C-Sigma approach by Kreutzer et al. and
- discuss the **topics of data** both in the report and during oral discussion of the project.



Tasks for a 4

- provide a **correct implementation** of the **ELLPACK/SELL-C-Sigma** approach by Kreutzer et al. and relate its performance to the capabilities of the GPU architecture (up to around 270 GB/s from global memory), and
- **critically assess** the topic of matrix-vector product and conjugate gradient algorithm in terms of **data locality**, including **measurements** of the **time to solve** the linear system (profiling) and **to copy** the matrix and vectors between host and device.



For Grade 5

- discuss and analyze the **data access patterns** of the CRS and SELL-C-Sigma approaches when executed locally, including an explanation between the difference in performance,
- **obtain good performance** of the tested scenarios, and
- **relate** the **CUDA** code to a **framework-based solution with Kokkos** or similar, and discuss possible **limitations** in terms of the data layout possible with multi-dimensional arrays.



Examination

- Can work in groups on the assignment
- 3-10 page report is individual
- Examination 20/10 is individually
- Main Focus:
 - Implement 25 lines of code in the basic algorithm
 - Implement advanced algorithm (more lines) + data layout + copy data correctly
 - Scientific discussion: For this algorithm CPU versus GPU

