Computational Science on Many-Core Architectures

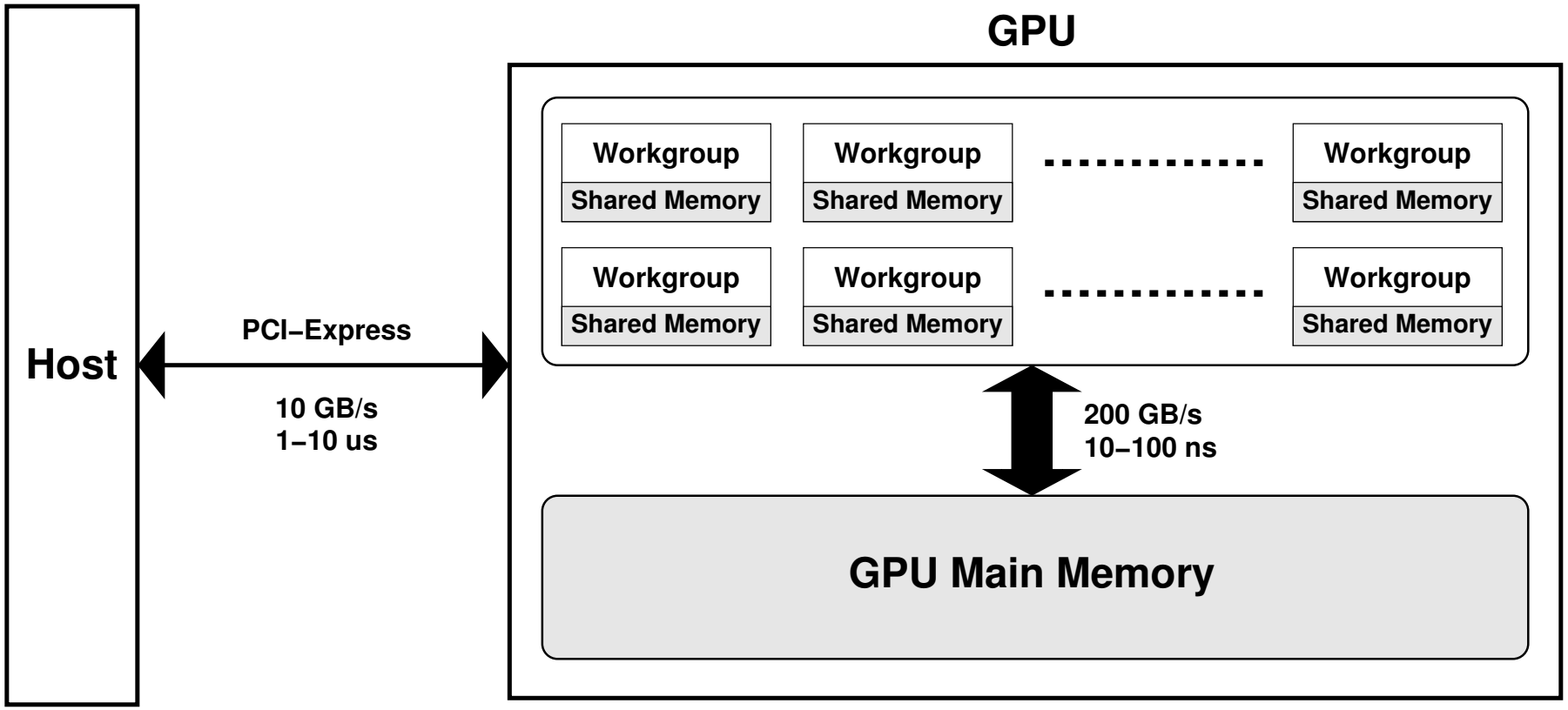
360.252

Karl Rupp

Institute for Microelectronics
Vienna University of Technology
http://www.iue.tuwien.ac.at

Zoom Channel 95028746244
Wednesday, October 21, 2020

GPU Overview



Details

- Workgroups consist of 32-64 hardware threads
- Up to 24 hardware workgroups
- Shared memory small: approx. 32-64 KB

CUDA

About

- Initial release in 2007
- Proprietary programming model by NVIDIA
- C++ with extensions
- Proprietary compiler extracts GPU kernels

Software Ecosystem

- Vendor-tuned libraries: cuBLAS, cuSparse, cuSolver, cuFFT, etc.
- Python bindings: pyCUDA
- Community projects: CUSP, MAGMA, ViennaCL, etc.

Programming in OpenMP

```
void work(double *x, double *y, double *z, int N)
{
  #pragma omp parallel
{ int thread_id = omp_get_thread_num();
  for (size_t i=thread_id; i<N; i += omp_get_num_threads())
    z[i] = x[i] + y[i];
} }
```

```
int main(int argc, char **argv)
{
  int N = atoi(argv[1]);
  double *x = malloc(N*sizeof(double));
  ...

  ...
  work(x, y, z, N); // call kernel
  ...


  free(x);
}
```

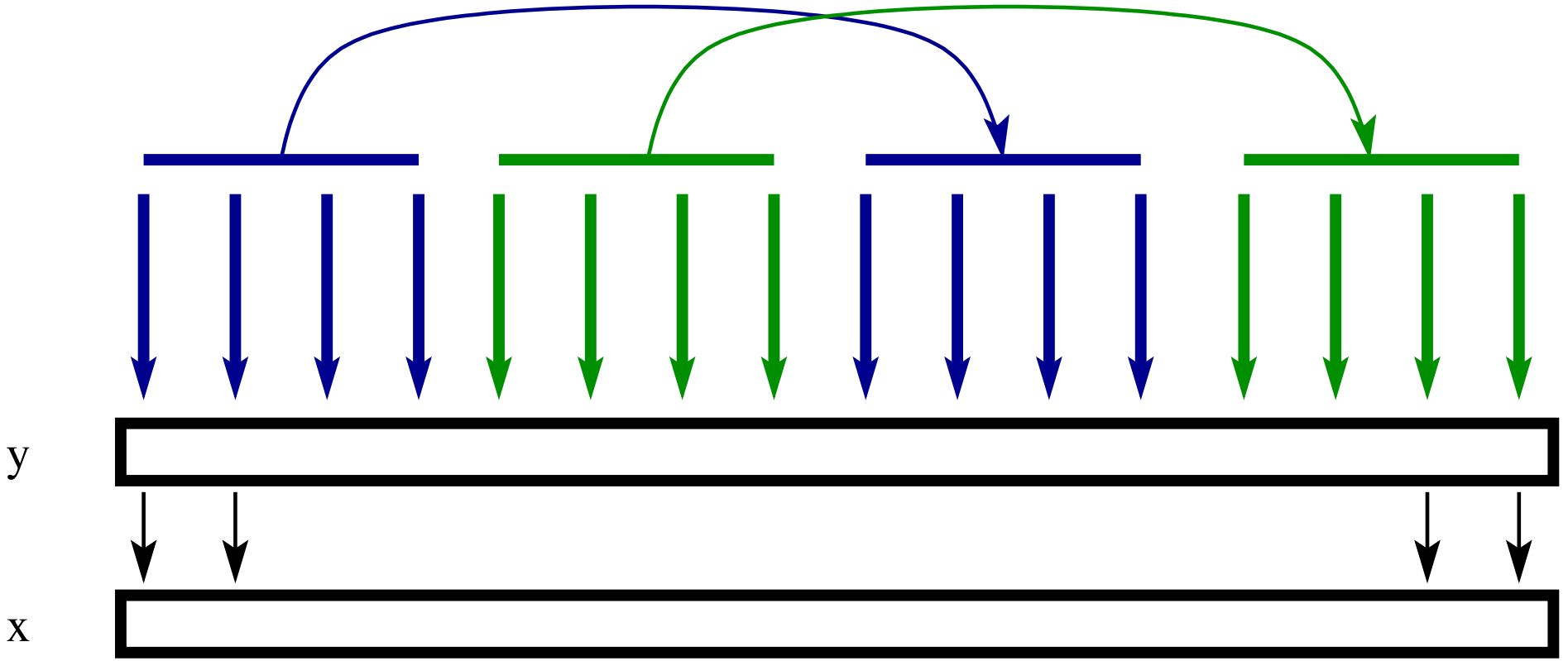
Programming in CUDA

```
__global__ void work(double *x, double *y, double *z, int N)
{

  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (size_t i=thread_id; i<N; i += blockDim.x * gridDim.x)
    z[i] = x[i] + y[i];
}
```

```
int main(int argc, char **argv)
{
  int N = atoi(argv[1]);
  double *x = malloc(N*sizeof(double));
  double *gpu_x; cudaMalloc(&gpu_x, N*sizeof(double));
  cudaMemcpy(gpu_x, x, N*8, cudaMemcpyHostToDevice);
  ...
  work<<<128, 256>>>(gpu_x, gpu_y, gpu_z, N); // call kernel
  ...
  cudaMemcpy(x, gpu_x, N*8, cudaMemcpyDeviceToHost);
  ...
  free(x); cudaFree(gpu_x); // similarly for y, z, gpu_y, gpu_z
}
```

CUDA



Thread Control (1D)

- Local ID in block: `threadIdx.x`
- Threads per block: `blockDim.x`
- ID of block: `blockIdx.x`
- No. of blocks: `gridDim.x`

Recommended Default Values

- Typical block size: 256 or 512
- Typical number of blocks: 256
- At least $10\,000$ logical threads recommended

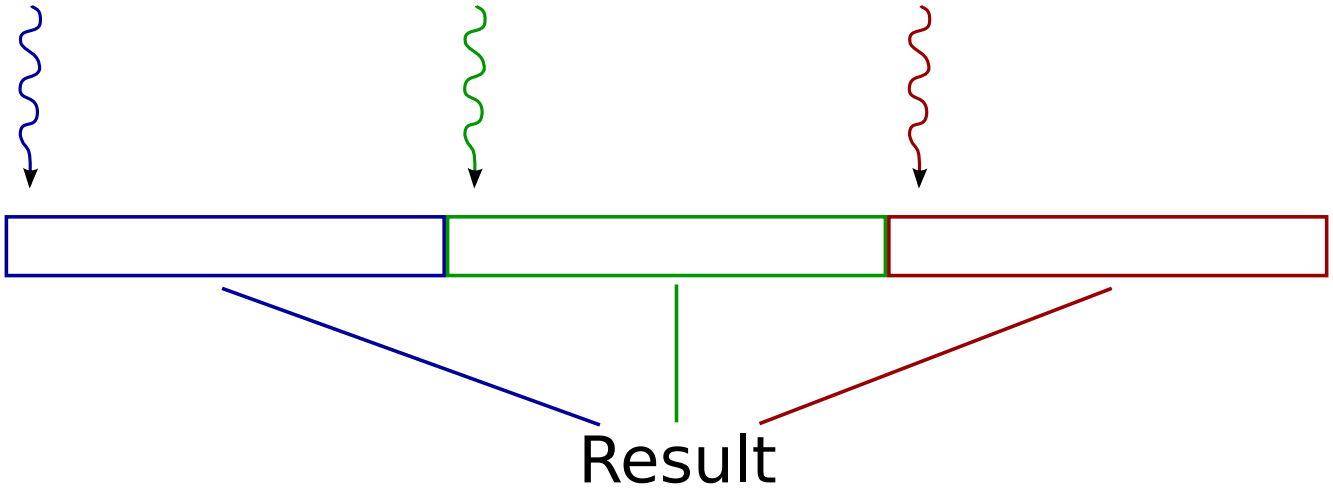
Parallel Primitives

Reductions

- Use N values to compute 1 result value
- Examples: Dot-products, vector norms, etc.

Reductions with Few Threads

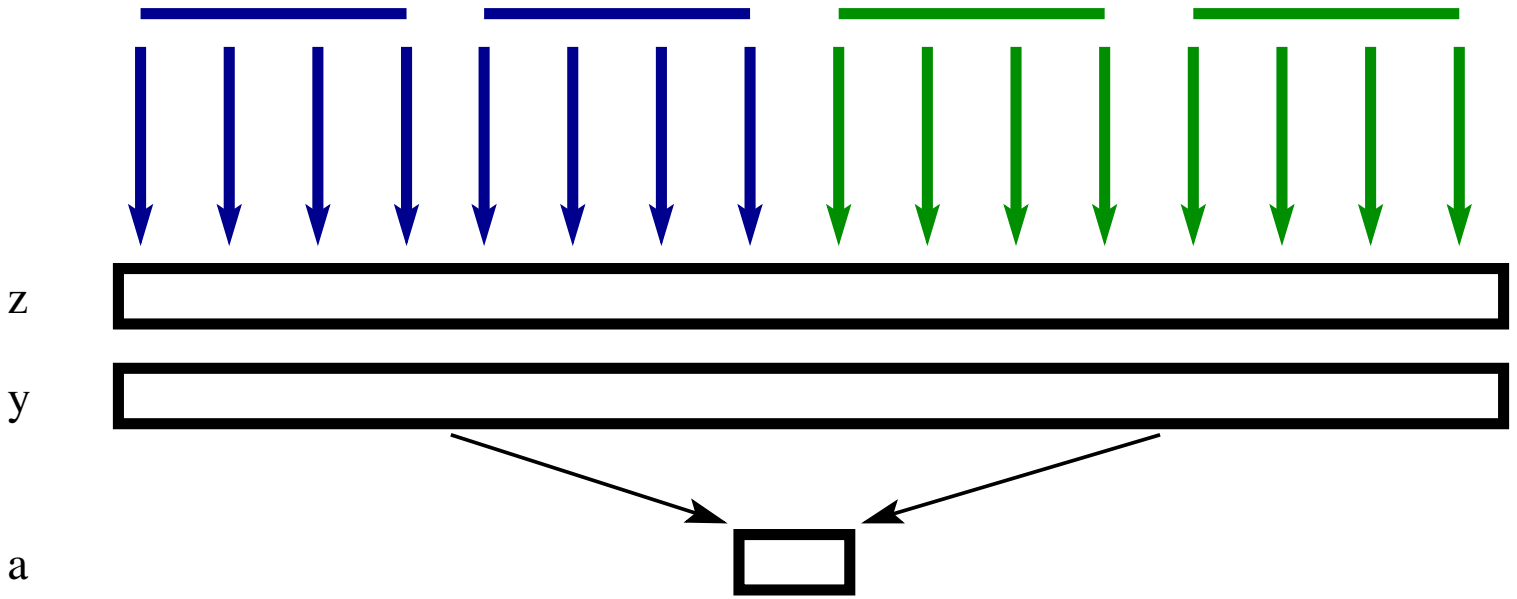
- Decompose N into chunks for each thread
- Compute chunks in parallel
- Merge results with single thread



Parallel Primitives

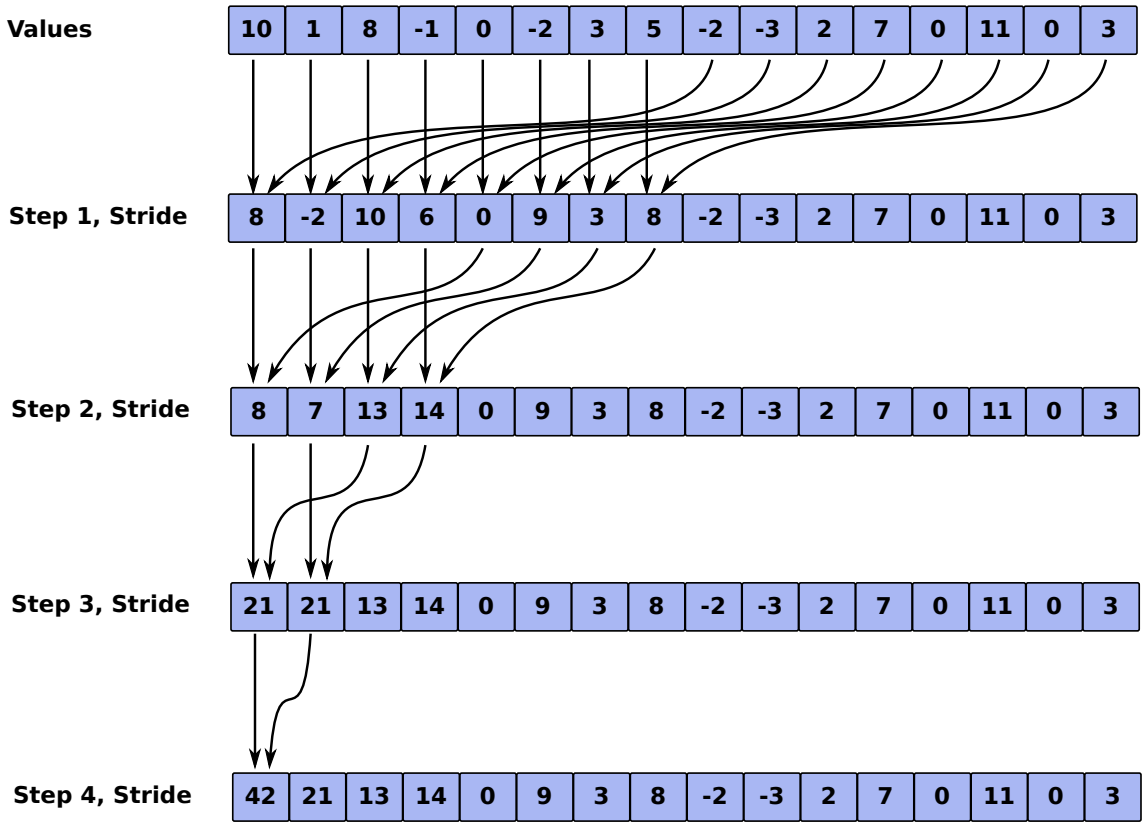
Reductions with Many Threads

- Decompose N into chunks for each workgroup
- Use fast on-chip synchronization within each workgroup
- Sum result for each workgroup separately



Parallel Primitives

Reductions with Many Threads




```
shared_m[threadIdx.x] = thread_sum;
for (int stride = blockDim.x/2; stride>0; stride/=2) {
  __syncthreads();
  if (threadIdx.x < stride)
    shared_m[threadIdx.x] += shared_m[threadIdx.x+stride];
}
```

Parallel Primitives

```
__global__ void sum_vector(const double * x, unsigned int N,
                           double * partial_results)
{
  __shared__ double shared_m[256]; // shared memory for each
      thread block

  double thread_sum = 0;  // local variable for each thread
  for (unsigned int i = threadIdx.x; i<N; i += blockDim.x)
      thread_sum += x[i];

  shared_m[threadIdx.x] = thread_sum;
  for (unsigned int stride = blockDim.x/2; stride>0; stride/=2)
  {
    __syncthreads();  // synchronize threads within thread block
    if (threadIdx.x < stride)
        shared_m[threadIdx.x] += shared_m[threadIdx.x + stride];
  }

  // only first thread of block writes result
  if (threadIdx.x == 0) {
    partial_results[blockIdx.x] = shared_m[0];
    // alternative: atomicAdd(result, shared_m[0]);
  }
}
```

Parallel Primitives

Prefix Sum

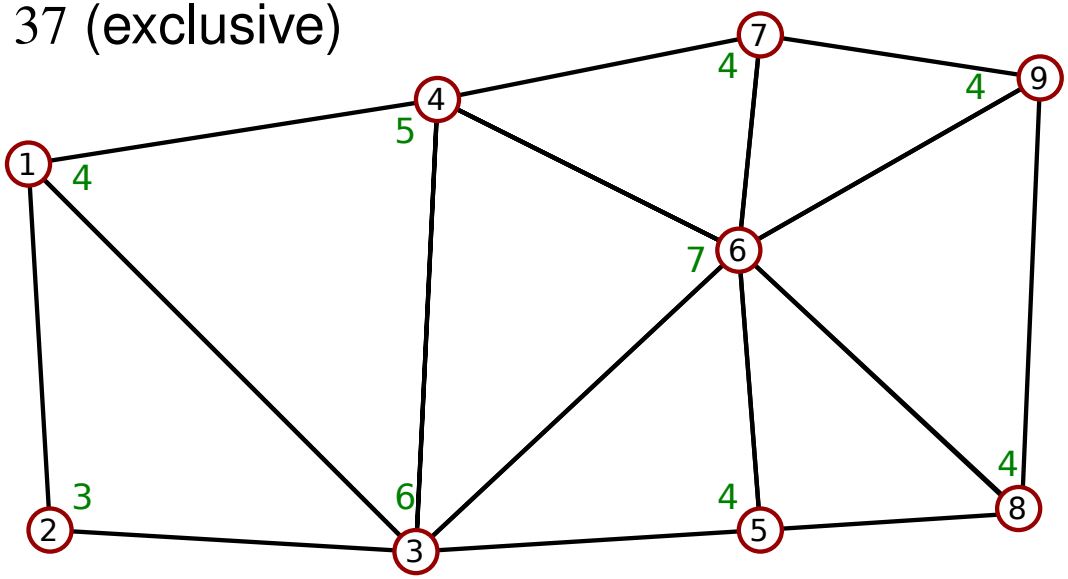
- Inclusive: Determine $y_i = \sum_{k=1}^{i} x_k$
- Exclusive: Determine $y_i = \sum_{k=1}^{i-1} x_k$, $y_1 = 0$

Example

- x: 4, 3, 6, 5, 4, 7, 4, 4, 4
- y: 4, 7, 13, 18, 22, 29, 33, 37, 41 (inclusive)
- y: 0, 4, 7, 13, 18, 22, 29, 33, 37 (exclusive)

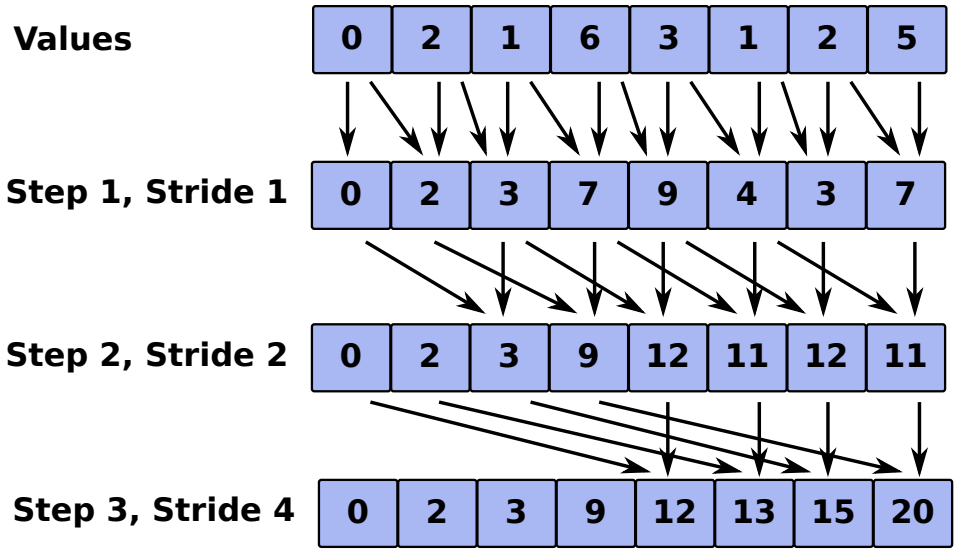
Applications

- Sparse matrix setup
- Graph algorithms



Parallel Primitives

Prefix Sum Implementation




```
for(int stride = 1; stride < blockDim.x; stride *= 2)
{
  __syncthreads();
  shared_m[threadIdx.x] = my_value;
  __syncthreads();
  if (threadIdx.x >= stride)
    my_value += shared_m[threadIdx.x - stride];
}
__syncthreads();
shared_m[threadIdx.x] = my_value;
```

Exercises

Environment

- https://gtx1080.360252.org/2020/ex2/
- (Might receive visual updates and additional hints over the next days)
- Due: Tuesday, November 3, 2020 at 23:59pm

Hints and Suggestions

- Consider version control for locally developed code
- Please let me know of any bugs or issues
- Use timer class as follows:

```
#include "timer.hpp"    // <----- include timer code
int main() {
  Timer timer;
  ...
  cudaDeviceSynchronize();  // make sure GPU is ready
  timer.reset(); // reset timer to 0
  your_kernel<<<BLOCKNUM,BLOCKSIZE>>>(x, N, y);
  cudaDeviceSynchronize(); // wait for GPU kernel to complete
  double time_elapsed = timer.get();  // in seconds
  ... }
```