Introduction to GPU Programming

Volodymyr (Vlad) Kindratenko
Innovative Systems Laboratory @ NCSA
Institute for Advanced Computing Applications and Technologies (IACAT)

Part III

- CUDA C and CUDA API
- Hands-on: reduction kernel
  - Reference implementation
  - GPU port

CUDA C

- CUDA C extends standard C as follows
  - Function type qualifiers to specify whether a function executes on the host or on the device
  - Variable type qualifiers to specify the memory location on the device
  - A new directive to specify how a kernel is executed on the device
  - Four built-in variables that specify the grid and block dimensions and the block and thread indices
  - Built-in vector types derived from basic integer and float types

Built-in Vector Types

Vector types derived from basic integer and float types

- char1, char2, char3, char4
- uchar1, uchar2, uchar3, uchar4
- short1, short2, short3, short4
- ushort1, ushort2, ushort3, ushort4
- int1, int2, int3, int4
- uint1, uint2, uint3 (dim3), uint4
- long1, long2, long3, long4
- ulong1, ulong2, ulong3, ulong4
- longlong1, longlong2
- float1, float2, float3, float4
- double1, double2

They are all structures, like this:

```
typedef struct {
   float x,y,z,w;
} float4;
```

They all come with a constructor function in the form **make_<type name>**, e.g.,

```
int2 make_int2(int x, int y);
```

Example

- dim3 dimBlock(width, height);
- dim3 dimGrid(10); *// same as dimGrid(10,0,0)*

- **myKernel<<<dimGrid, dimBlock>>>();**

Built-in Variables

variable	type	description
`gridDim`	dim3	dimensions of the grid
`blockID`	unit3	block index within the grid
`blockDim`	dim3	dimensions of the block
`threadIdx`	uint3	thread index within the block
`warpSize`	int	warp size in threads

It is not allowed to take addresses of any of the built-in variables
It is not allowed to assign values to any of the built-in variables

Example

```
myKernel<<<10, 32>>>();

__global__  void myKernel()
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   C[i] = A[i] + B[i];
}
```

- here
  - gridDim.x is 10
  - blockDim.x is 32

Variable Type Qualifiers

	Memory	Scope	Lifetime
`__device__ int GlobalVar;`	global	grid	application
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ __constant__ int ConstantVar;`	constant	grid	application
`volatile int GlobarVar or SharedVar;`			

__shared__ and __constant__ variables have implied static storage
__device__, __shared__ and __constant__ variables cannot be defined using external keyword
__device__ and __constant__ variables are only allowed at file scope
__constant__ variables cannot be assigned to from the devices, they are initialized from the host only
__shared__ variables cannot have an initialization as part of their declaration

Example

```
__global__ void myKernel()
{
    __shared__ float shared[32];
    __device__ float device[32];
    shared[threadIdx.x] = device[threadIdx.x];
}
```

Example

```
__global__ void myKernel()
{
   extern __shared__ int s_data[];

   s_data[threadIdx.x] = …
}

main()
{
   int sharedMemSize = numThreadsPerBlock * sizeof(int);
   dim3 dimGrid(numBlocks);
   dim3 dimBlock(numThreadsPerBlock);
   myKernel <<< dimGrid, dimBlock, sharedMemSize >>>();
}
```

Function Type Qualifiers

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

__device__ and __global__ functions do not support recursion, cannot declare static variables inside their body, cannot have a variable number of arguments

__device__ functions cannot have their address taken

__host__ and __device__ qualifiers can be used together, in which case the function is compiled for both

__global__ and __host__ qualifiers cannot be used together

__global__ function must have void return type, its execution configuration must be specified, and the call is asynchronous

Example

```
__device__ int get_global_index(void)
{
    return blockIdx.x * blockDim.x + threadIdx.x;
}

__global__ void myKernel(int *array)
{
    int index = get_global_index();
}

main()
{ …
     myKernel<<<gridSize, blockSize>>>(gArray);
… }
```

Execution Configuration

Function declared as

__global__ void kernel(float* param);

must be called like this:

kernel<<<Dg, Db, Ns, S>>>(param);

where

- **Dg** (type dim3) specifies the dimension and size of the grid, such that Dg.x*Dg.y equals the number of blocks being launched;
- **Db** (type dim3) spesifies the dimension abd size of each block of threads, such that Db.x*Db.y*Db.z equals the number of threads per block;
- optional **Ns** (type size_z) specifies the number of bytes of shared memory dynamically allocated per block for this call in addition to the statically allocated memory
- optional **S** (type cudaStream_t) specifies the stream associated with this kernel call

V. Kindratenko, **Introduction to GPU Programming (part III)**, December 2010, The American University in Cairo, Egypt

Intrinsic Functions

Supported on the device only

Start with **__**, as in **__sinf(x)**

End with
_rn (round-to-nearest-even rounding mode)
_rz (round-towards-zero rounding mode)
_ru (round-up rounding mode)
_rd (round-down rounding mode)
as in **__fadd_rn(x,y);**

There are mathematical **(__log10f(x)**), type conversion **(__int2float_rn(x)**), type casting **(__int_as_float(x)**), and bit manipulation **(__ffs(x)**) functions

Example

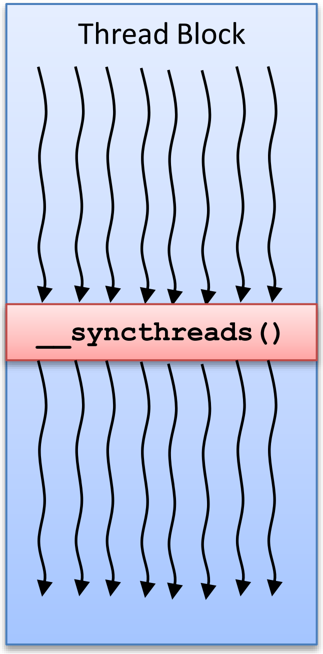
```
__global__ void myKernel(float *a1, float *a2)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    a1[index] = sin(a1[index]);

    // faster, but less precise than sin()
    a2[index] = __sin_rn(a2[index]);
}
```

Synchronization and Memory Fencing Functions

function	description
`void __threadfence()`	wait until all global and shared memory accesses made by the calling thread become visible to all threads in the device for global memory accesses and all threads in the thread block for shared memory accesses
`void __threadfence_block()`	Waits until all global and shared memory accesses made by the calling thread become visible to all threads in the thread block
`void __syncthreads()`	Waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads become visible to all threads in the block

Example

```
__global__ void myKernel(float *a1, float *a2)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    a1[index] = a1[index] + a2[index];

    __syncthreads();

    a2[index] = a1[blockDim.x-index-1];
}
```



Atomic Functions

<table>
<tr><th>function</th><th>Description</th></tr>
<tr><td>atomicAdd()</td><td>new = old + val</td></tr>
<tr><td>atomicSub()</td><td>new = old – val</td></tr>
<tr><td>atomicExch()</td><td>new = val</td></tr>
<tr><td>atomicMin()</td><td>new = min(old, val)</td></tr>
<tr><td>atomicMax()</td><td>new = max(old, val)</td></tr>
<tr><td>atomicInc()</td><td>new = ((old >= val) ? 0 : (old+1))</td></tr>
<tr><td>atomicDec()</td><td>new = (((old==0) | (old > val)) ? val : (old-1))</td></tr>
<tr><td>atomicCAS()</td><td>new = (old == compare ? val : old)</td></tr>
<tr><td>Atomic{And, Or, Xor}()</td><td>new = {(old & val), (old | val), (old^val)}</td></tr>
</table>

An atomic function performs read-modify-write atomic operation on one 32-bit or one 64-bit word residing in global or shared memory. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads.

Example

```
__shared__ totalSum;
if (threadIdx.x == 0) totalSum = 0;
__syncthreads();

int localVal = pValues[blockIdx.x * blockDim.x + threadIdx.x];
atomicAdd(&totalSum, 1);
__syncthreads();
```

Device Management

function	description
`cudaGetDeviceCount()`	Returns the number of compute-capable devices
`cudaGetDeviceProperties()`	Returns information on the compute device
`cudaSetDevice()`	Sets device to be used for GPU execution
`cudaGetDevice()`	Returns the device currently being used
`cudaChooseDevice()`	Selects device that best matches given criteria

Device Management Example

```
void cudaDeviceInit() {
    int devCount, device;
    cudaGetDeviceCount(&devCount);
    if (devCount == 0) {
        printf("No CUDA capable devices detected.\n");
        exit(EXIT_FAILURE);
    }
    for (device=0; device < devCount; device++) {
        cudaDeviceProp props;
        cudaGetDeviceProperties(&props, device);
        // If a device of compute capability >= 1.3 is found, use it
        if (props.major > 1 || (props.major == 1 && props.minor >= 3)) break;
    }
    if (device == devCount) {
        printf("No device above 1.2 compute capability detected.\n");
        exit(EXIT_FAILURE);
    }
    else cudaSetDevice(device);
}
```

Memory Management

function	description
cudaMalloc()	Allocates memory on the GPU
cudaMallocPitch()	Allocates memory on the GPU device for 2D arrays, may pad the allocated memory to ensure alignment requirements
cudaFree()	Frees the memory allocated on the GPU
cudaMallocArray()	Allocates an array on the GPU
cudaFreeArray()	Frees an array allocated on the GPU
cudaMallocHost()	Allocates page-locked memory on the host
cudaFreeHost()	Frees page-locked memory in the host

Memory Management (Cont.)

function	description
`cudaMemset()`	Initializes or sets GPU memory to a value
`cudaMemCpy()`	Copies data between host and the device
`cudaMemcpyToArray()`	
`cudaMemcpyFromArray()`	
`cudaMemcpyArrayToArray()`	
`cudaMemcpyToSymbol()`	
`cudaMemcpyFromSymbol()`	
`cudaGetSymbolAddress()`	Finds the address associated with a CUDA symbol
`cudaGetSymbolSize()`	Finds the size of the object associated with a CUDA symbol

Example

```
main()
{ …

float *devPtrA, *devPtrB;

cudaMalloc((void**)&devPtrA, N * sizeof(float));
cudaMemcpy(devPtrA, A, N * sizeof(float),  cudaMemcpyHostToDevice);
cudaMalloc((void**)&devPtrB, N * sizeof(float));
cudaMemset(evPtrB, 0, N * sizeof(float));

// call kernel
myKernel<<<…>>>(devPtrA, devPtrB, N);

cudaMemcpy(B, devPtrB, N * sizeof(float),  cudaMemcpyDeviceToHost);

cudaFree(devPtrA);
cudaFree(devPtrB);
… }
```

Error Handling

All CUDA runtime API functions return an error code. The runtime maintains an error variable for each host thread that is overwritten by the error code every time an error concurs.

function	description
`cudaGetLastError()`	Returns error variable and resets it to cudaSuccess
`cudaGetErrorString()`	Returns the message string from an error code

```
cudaError_t err = cudaGetLastError();
if (cudaSuccess != err)  {
        fprintf(stderr, "CUDA error: %s.\n", cudaGetErrorString( err) );
        exit(EXIT_FAILURE);
}
```

Sum reduction kernel example

- Source is in ~/tutorial/src4
  - sum.c – reference C implementation
  - makefile – make file
  - sum.cu.reference – CUDA implementation for reference

Sum reduction

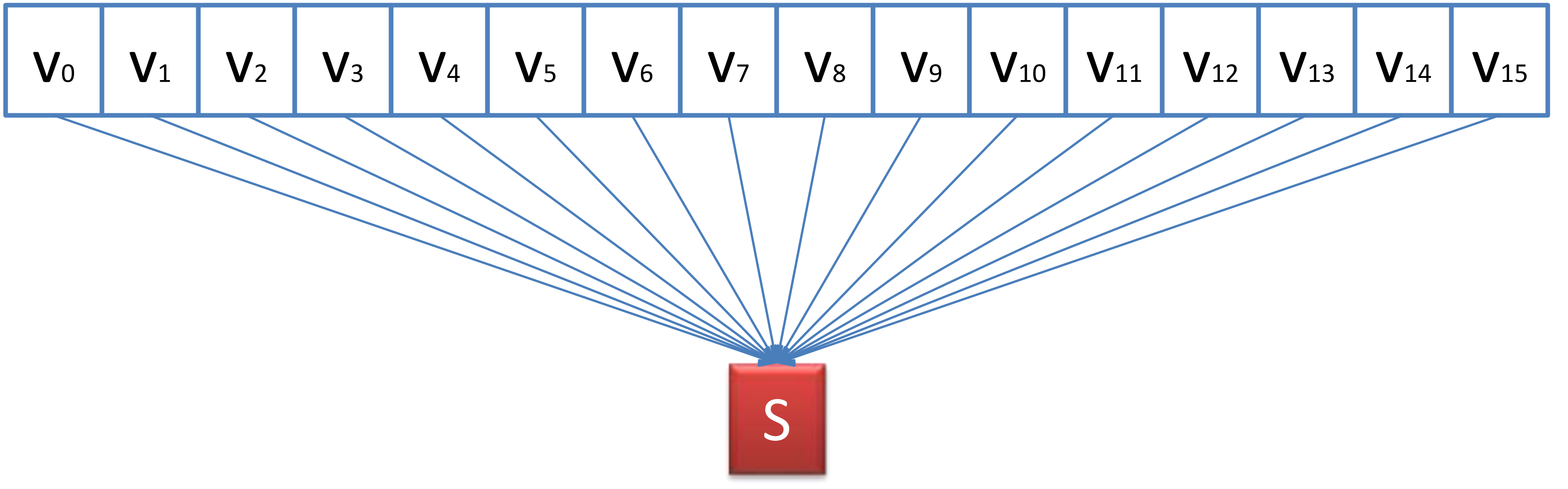
```
int  main(int argc, char **argv)
{
   int i, N = 2097152;  // vector size
   double *A, s = 0.0f;

   A = (double*)malloc(N * sizeof(double));

   // generate random data
   for (i = 0; i < N; i++)
      A[i] = (double)rand()/RAND_MAX;

   s = sum(A, N);  // call compute kernel

   printf("sum=%.2f\n", s);

   free(A);  // free allocated memory
}
```

$$S = \sum_{k=0}^{n} v_k$$

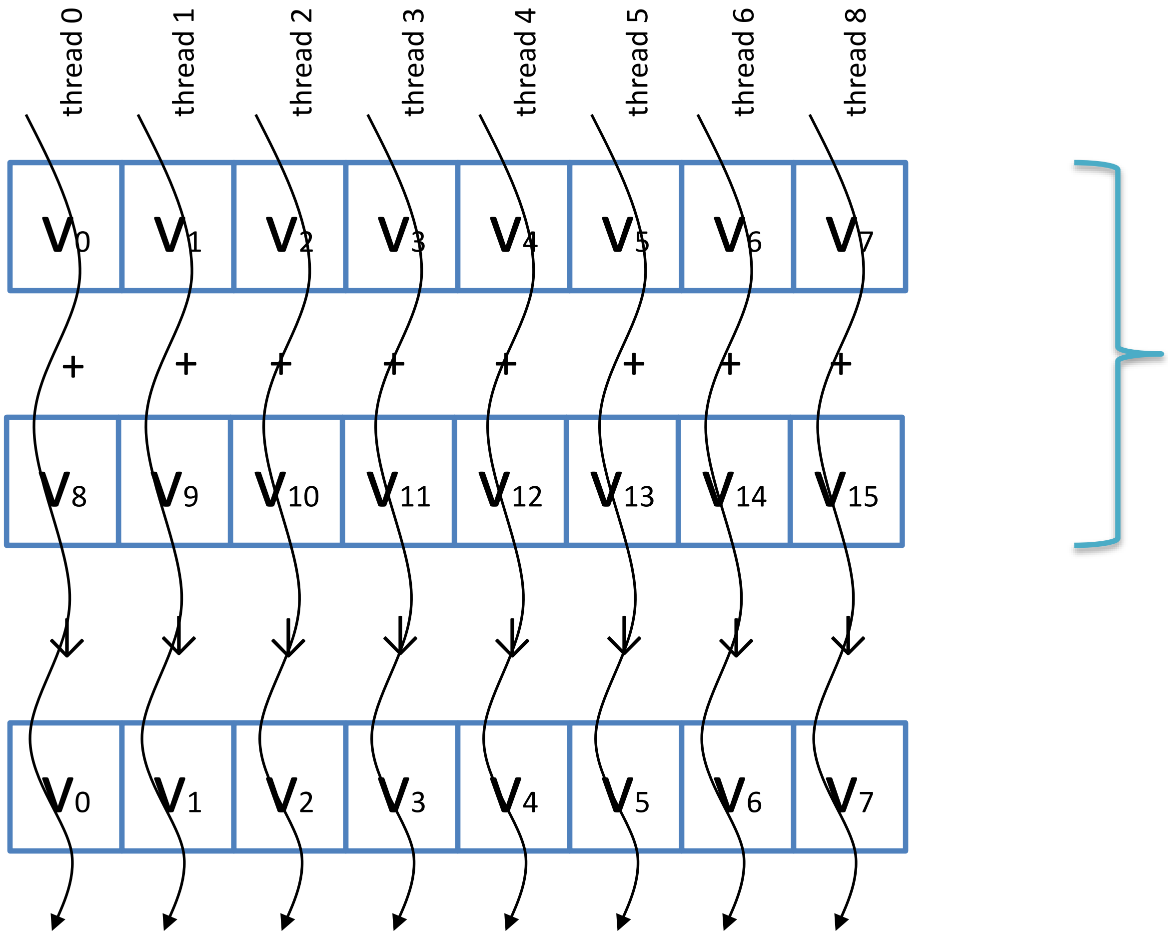
```
double sum(double* v, int n)
{
   int i;
   double s = 0.0f;

   for (i = 0; i < n; i++)
      s += v[i];

   return s;
}
```

Where do we find parallelism?

$$S = \sum_{k=0}^{15} v_k$$



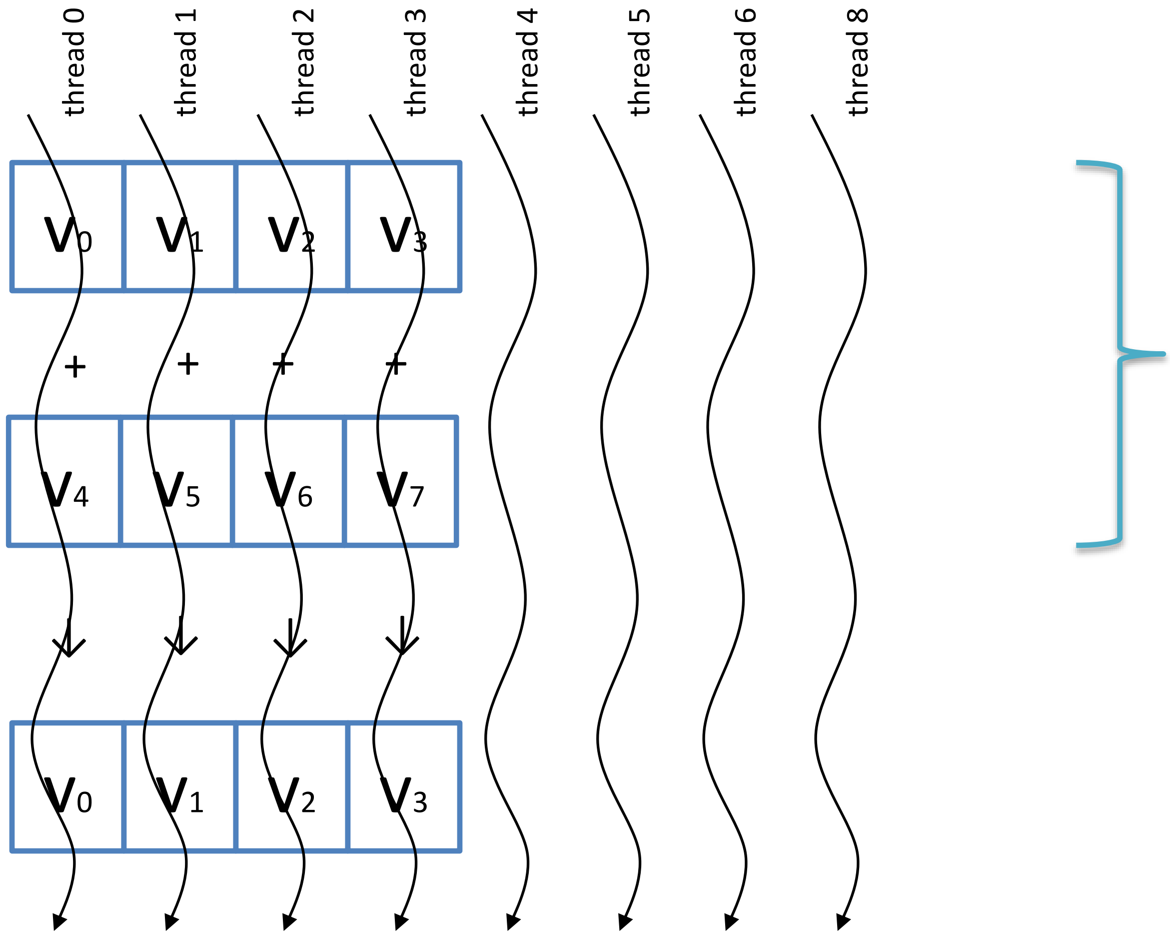
Where do we find parallelism?



N/2 additions can be done independently

V. Kindratenko, **Introduction to GPU Programming (part III)**, December 2010, The American University in Cairo, Egypt

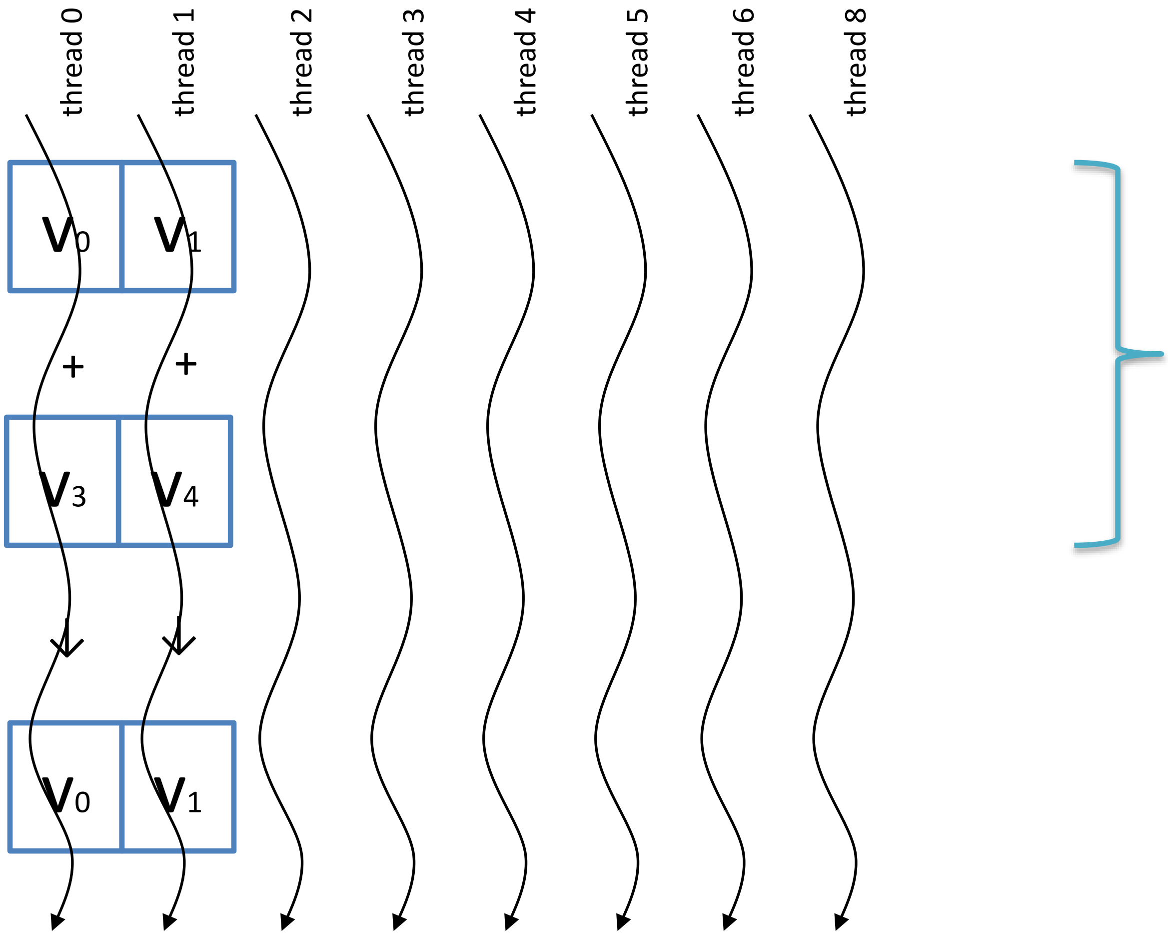
Where do we find parallelism?



N/4 additions can be done independently

V. Kindratenko, **Introduction to GPU Programming (part III)**, December 2010, The American University in Cairo, Egypt

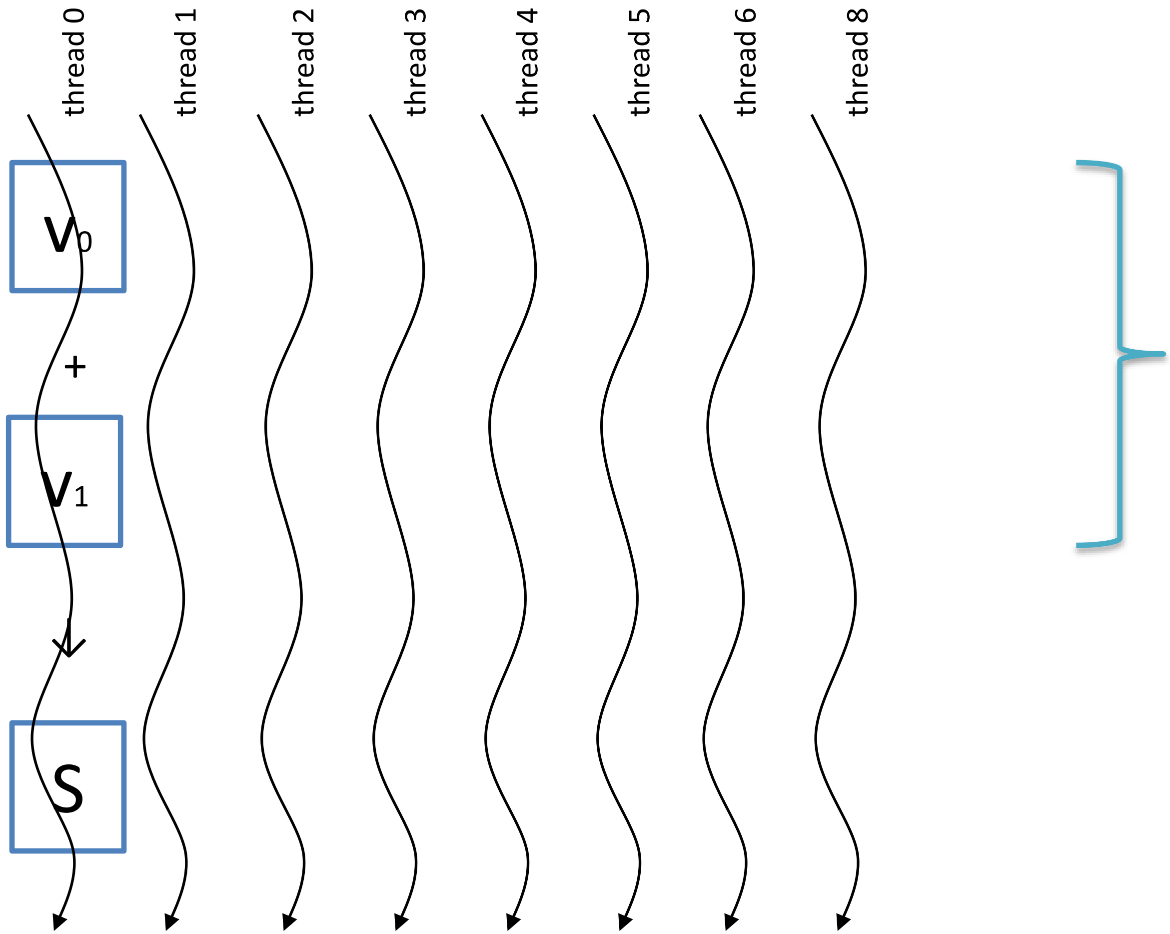
Where do we find parallelism?



N/8 additions can be done independently

V. Kindratenko, **Introduction to GPU Programming (part III)**, December 2010, The American University in Cairo, Egypt

Where do we find parallelism?



N/16 additions can be done independently

GPU kernel for N<=1024

```
__global__ void sum (double *v)
{
    unsigned int t = threadIdx.x;
    unsigned int stride;

    for (stride = blockDim.x >> 1; stride > 0; stride >>= 1)
    {
        __syncthreads();
        if (t < stride)
            v[t] += v[t+stride];
    }
}
```

sum<<<1, N/2>>>(a);

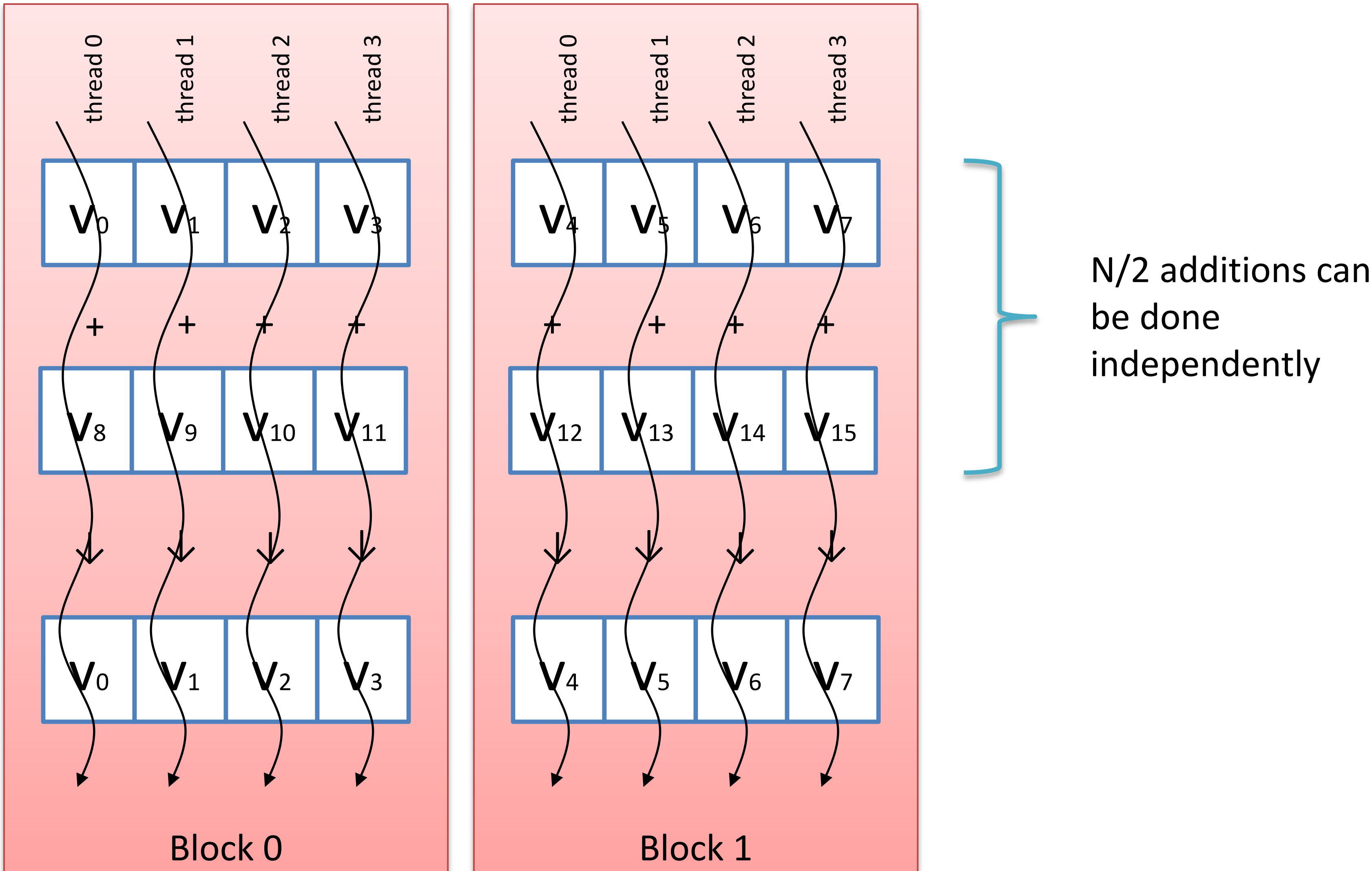
The rest of the code

```
double *devPtrA; // allocate memory, copy data
cudaMalloc((void**)&devPtrA, N * sizeof(double));
cudaMemcpy(devPtrA, A, N * sizeof(double),  cudaMemcpyHostToDevice);

sum<<<1, N/2>>>(devPtrA); // call compute kernel

cudaError_t err = cudaGetLastError(); // check for errors
if (cudaSuccess != err)
{
   fprintf(stderr, "CUDA error: %s.\n", cudaGetErrorString( err) );
   exit(EXIT_FAILURE);
}

// get results, free memory
cudaMemcpy(&s, devPtrA, sizeof(double),  cudaMemcpyDeviceToHost);
cudaFree(devPtrA);
```

Problems with this implementation

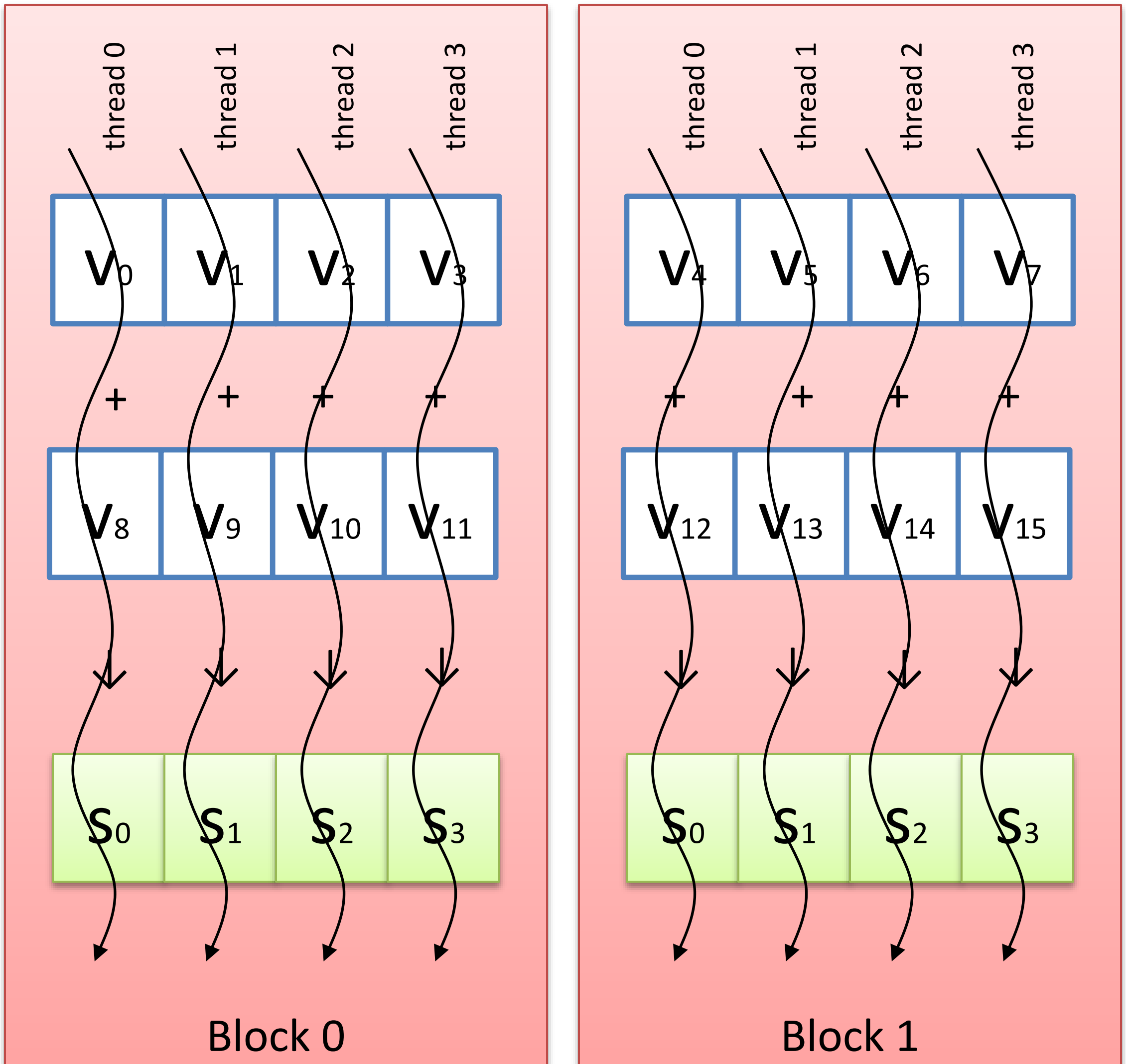
- N <= 1024
  - A thread block may not have more than 512 threads
- Inefficient
  - Data is stored in global memory which has very high access latency
- N must be a power of 2

Expanding to multiple thread blocks



V. Kindratenko, **Introduction to GPU Programming (part III)**, December 2010, The American University in Cairo, Egypt

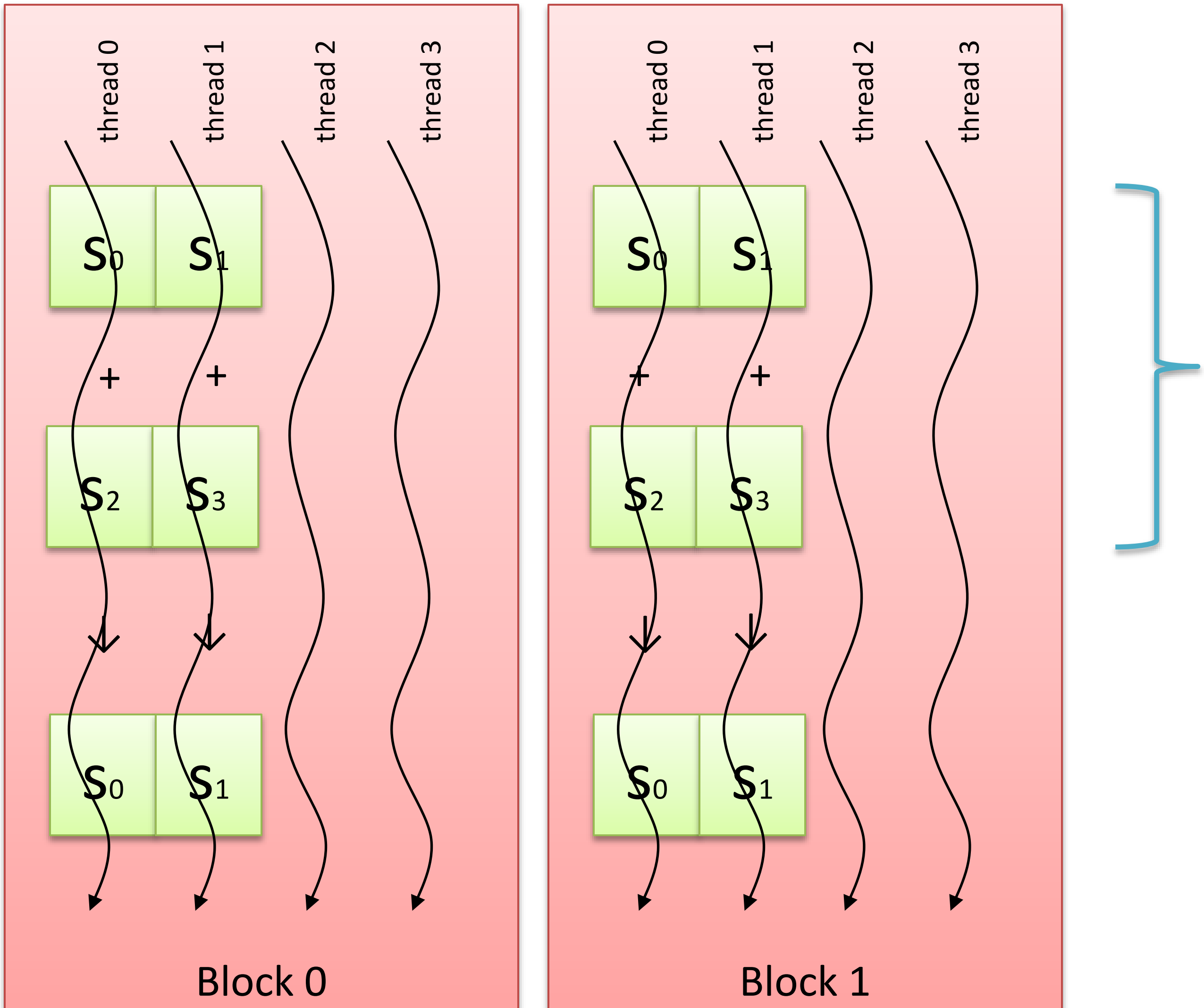
Eliminating global memory access latency



N/2 additions can be done independently

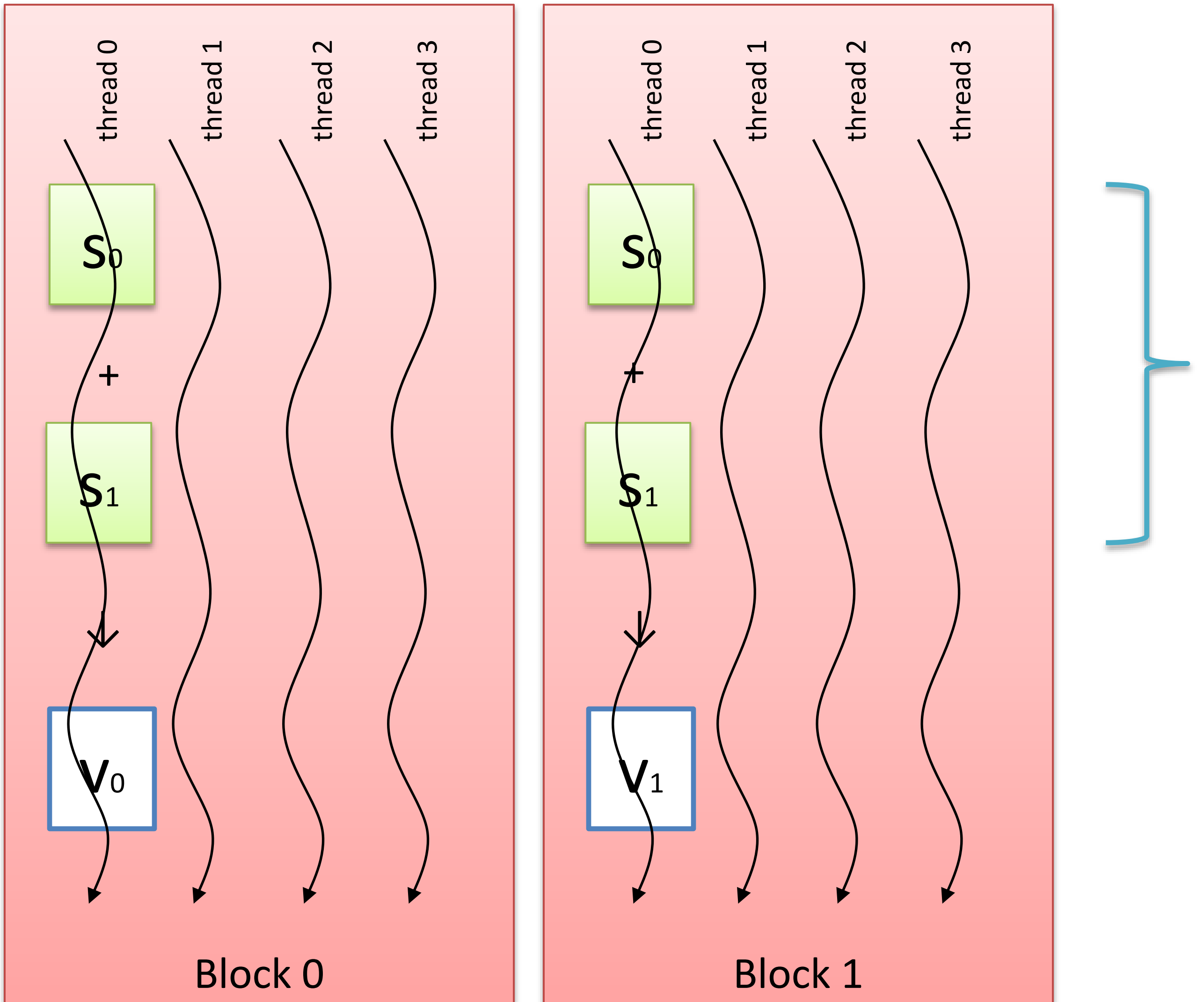
Store partial sums in the per-block shared memory

Expanding to multiple thread blocks



N/4 additions can be done independently

Expanding to multiple thread blocks



N/8 additions can be done independently

V. Kindratenko, **Introduction to GPU Programming (part III)**, December 2010, The American University in Cairo, Egypt

Final sum reduction kernel

```
__global__ void sum(double *v)
{
   extern double __shared__ sd[];
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;

   sd[tid] = v[i] + v[i+blockDim.x];
   __syncthreads();

   for (unsigned int s = blockDim.x/2; s > 0; s >>= 1)
   {
      if (tid < s)
         sd[tid] += sd[tid + s];
      __syncthreads();
   }

   if (tid == 0) v[blockIdx.x] = sd[0];
}
```

perform first level of reduction, reading from global memory, writing to shared memory

do reduction in shared memory

write result for this block to global mem

Are we done yet?

- We started with this

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}

- And ended with this

v_0	v_1

- where v_0 and v_1 are partial sums computed by individual thread blocks, stored in global memory, and they still need to be added
- The final addition can be done by running the same kernel on this reduced data set

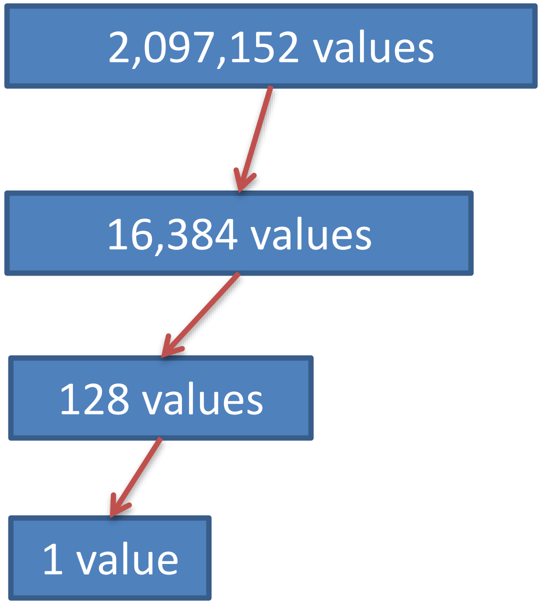
Modified host code

```
int threads = 64;
int old_blocks, blocks = N / threads / 2;
blocks = (blocks == 0) ? 1 : blocks;
old_blocks = blocks;

while (blocks > 0)   // call compute kernel
{
   sum<<<blocks, threads, threads*sizeof(double)>>>(devPtrA);
   old_blocks = blocks;
   blocks = blocks / threads / 2;
};

if (blocks == 0 && old_blocks != 1)   // final kernel call, if still needed
      sum<<<1, old_blocks/2, old_blocks/2*sizeof(double)>>>(devPtrA);
```

Example run

- [kindr@ac src4]$ ./sum_cpu
- Running CPU sum for 2097152 elements
- sum=1048443.09
- sec = 0.006771   GFLOPS = 0.309

- [kindr@ac src4]$ ./sum_gpu
- Running GPU sum for 2097152 elements
- Grid/thread dims are (16384), (64)
- Grid/thread dims are (128), (64)
- Grid/thread dims are (1), (64)
- sum=1048443.09
- sec = 0.000389   GFLOPS = 5.391



Lab/Homework Exercises

- Exercise 2: Modify reduction example to eliminate multiple calls to the kernel
  - hint: use atomic add