# CUDA

Sathish Vadhiyar
High Performance Computing

# Hierarchical Parallelism

- Parallel computations arranged as grids
- One grid executes after another
- Grid consists of blocks
- Blocks assigned to SM. A single block assigned to a single SM. Multiple blocks can be assigned to a SM.
  - Max thread blocks executed concurrently per SM = 16

# Hierarchical Parallelism

- Block consists of elements
- Elements computed by threads
  - Max threads per thread block = 1024
- A thread executes on a GPU core

# Hierarchical Parallelism

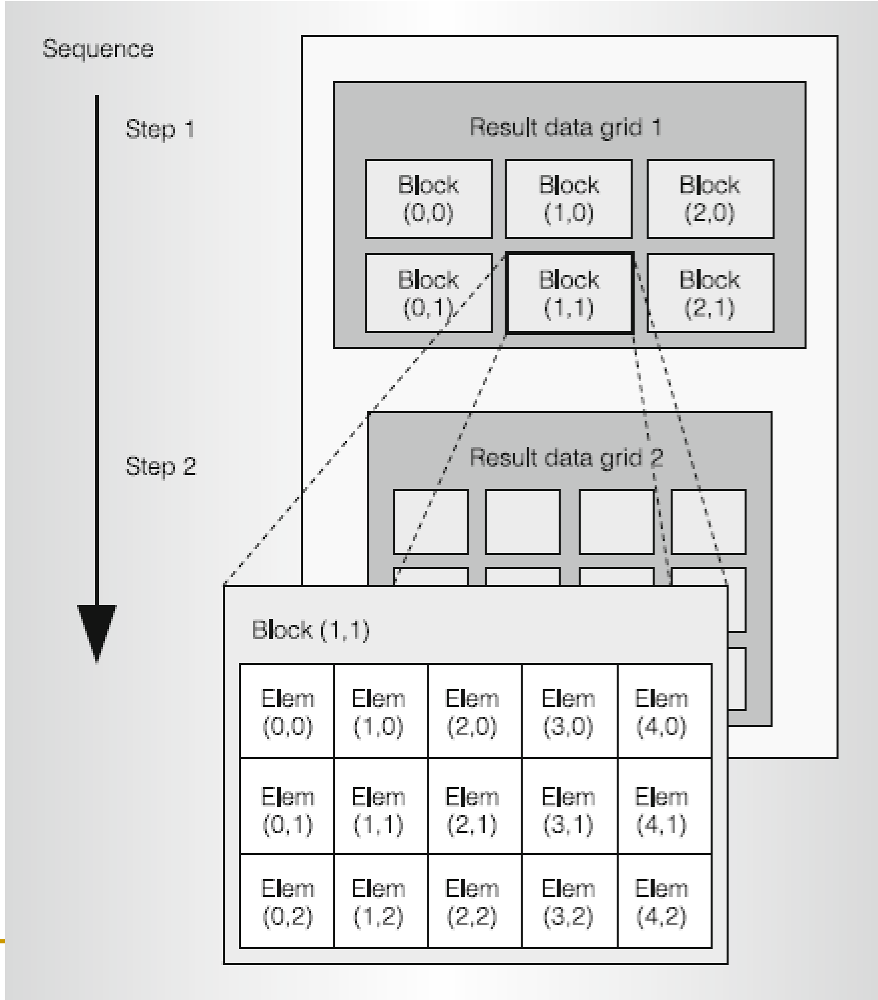



Figure 5. Decomposing result data into a grid of blocks partitioned into elements to be computed in parallel.

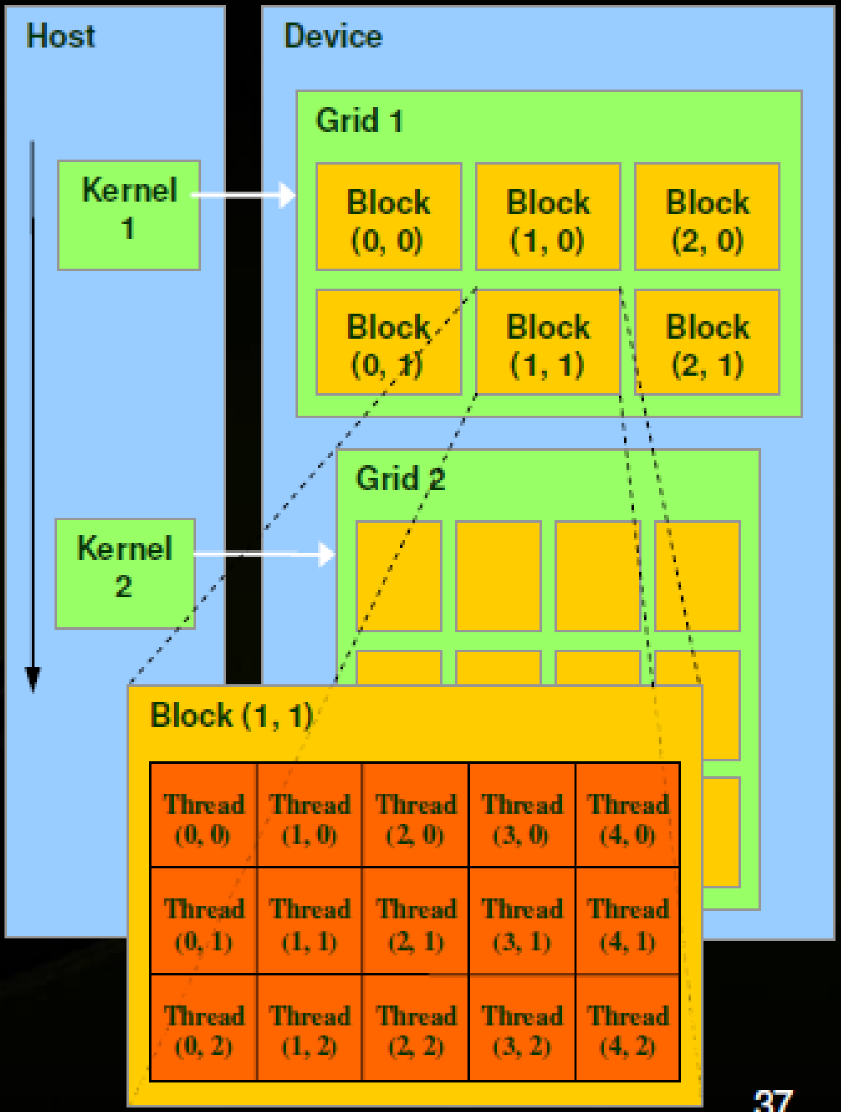
# Thread Blocks

- Thread block – an array of concurrent threads that execute the same program and can cooperate to compute the result
- Has shape and dimensions (1d, 2d or 3d) for threads
- A thread ID has corresponding 1,2 or 3d indices
- Threads of a thread block share memory

# CUDA Programming Model

A kernel is executed by a grid of thread blocks

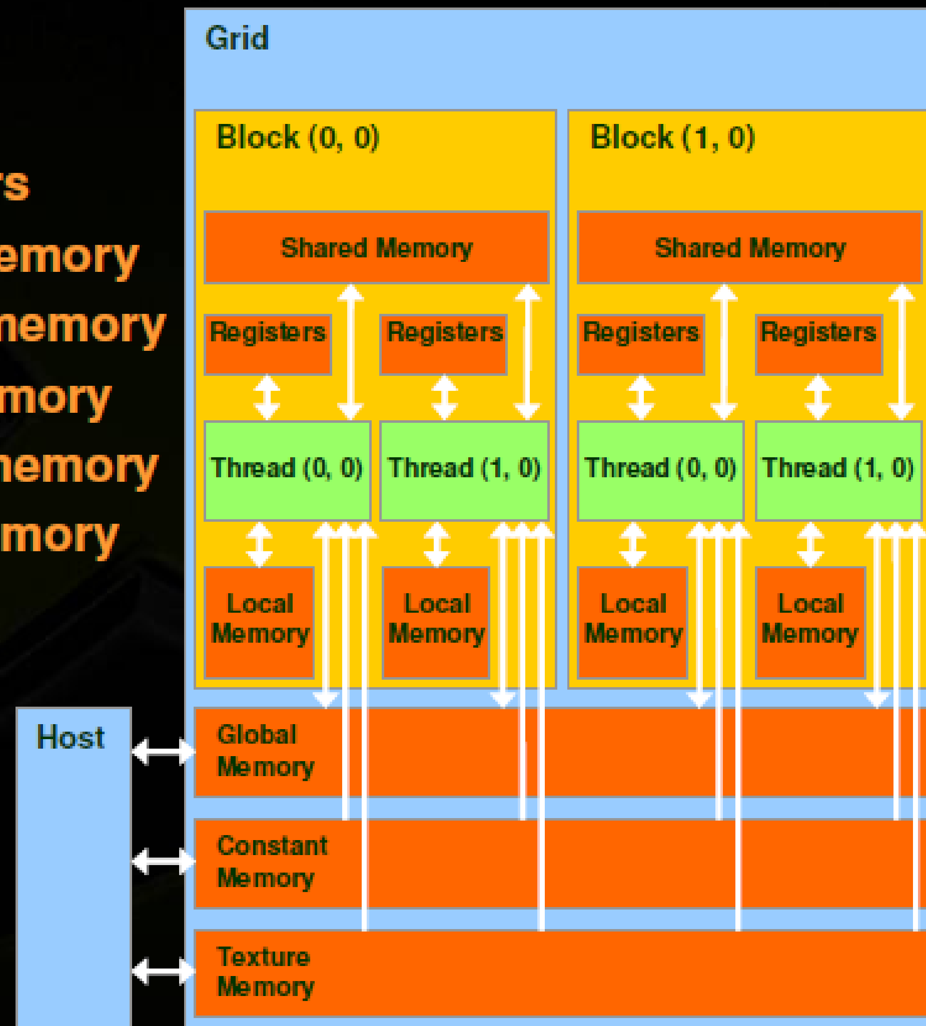
- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate



# CUDA Memory Spaces

- Each thread can:
  - Read/write per-thread **registers**
  - Read/write per-thread **local memory**
  - Read/write per-block **shared memory**
  - Read/write per-grid **global memory**
  - Read only per-grid **constant memory**
  - Read only per-grid **texture memory**

- The host can read/write global, constant, and texture memory (stored in DRAM)



# CUDA Memory Spaces

- Global and Shared Memory introduced before
  - Most important, commonly used
- Local, Constant, and Texture for convenience/performance
  - Local: automatic array variables allocated there by compiler
  - Constant: useful for uniformly-accessed read-only data
    - Cached (see programming guide)
  - Texture: useful for spatially coherent random-access read-only data
    - Cached (see programming guide)
    - Provides address clamping and wrapping

| Memory | Location | Cached | Access | Scope ("Who?") |
|---|---|---|---|---|
| Local | Off-chip | No | Read/write | One thread |
| Shared | On-chip | N/A | Read/write | All threads in a block |
| Global | Off-chip | No | Read/write | All threads + host |
| Constant | Off-chip | Yes | Read | All threads + host |
| Texture | Off-chip | Yes | Read | All threads + host |

# CUDA Programming Language

- Programming language for threaded parallelism for GPUs
- Minimal extension of C
- A serial program that calls parallel kernels
- Serial code executes on CPU
- Parallel kernels executed across a set of parallel threads on the GPU
- Programmer organizes threads into a hierarchy of thread blocks and grids

# CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as **kernels**
  - One **kernel** is executed at a time
  - Many threads execute each **kernel**
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions:
*Device* = GPU; *Host* = CPU
*Kernel* = function that runs on the device

# CUDA C

- Built-in variables:
  - threadIdx.{x,y,z} – thread ID within a block
  - blockIDx.{x,y,z} – block ID within a grid
  - blockDim.{x,y,z} – number of threads within a block
  - gridDim.{x,y,z} – number of blocks within a grid
- kernel<<<nBlocks,nThreads>>>(args)
  - Invokes a parallel kernel function on a grid of nBlocks where each block instantiates nThreads concurrent threads

# Example: Summing Up

kernel function

```c
void addMatrix
     (float *a, float *b, float *c, int N)
{
     int i, j, idx;
     for (i = 0; i < N; i++) {
          for (j = 0; j < N; j++) {
               idx = i + j*N;
               c[idx] = a[idx] + b[idx];
          }
     }
}
void  main()
{
     . . .
     addMatrix(a, b, c, N);
}
```

(a)

```c
__global__ void addMatrixG
     (float *a, float *b, float *c, int N)
{
     int i = blockIdx.x*blockDim.x + threadIdx.x;
     int j = blockIdx.y*blockDim.y + threadIdx.y;
     int idx = i + j*N;
     if (i < N && j < N)
          c[idx] = a[idx] + b[idx];
}

void  main()
{
     dim3 dimBlock (blocksize, blocksize);
     dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
     addMatrixG<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

(b)

Figure 8. Serial C (a) and CUDA C (b) examples of programs that add arrays.

grid of kernels

# Variable Qualifiers (GPU code)

- __device__
  - stored in device memory (large, high latency, no cache)
  - Allocated with cudaMalloc (__device__ qualifier implied)
  - accessible by all threads
  - lifetime: application
- __constant__
  - same as __device__, but cached and read-only by GPU
  - written by CPU via cudaMemcpyToSymbol(...) call
  - lifetime: application
- __shared__
  - stored in on-chip shared memory (very low latency)
  - accessible by all threads in the same thread block
  - lifetime: kernel launch
- Unqualified variables:
  - scalars and built-in vector types are stored in registers
  - arrays of more than 4 elements stored in device memory

# General CUDA Steps

1. Copy data from CPU to GPU
2. Compute on GPU
3. Copy data back from GPU to CPU

- By default, execution on host doesn’t wait for kernel to finish
- General rules:
  - Minimize data transfer between CPU & GPU
  - Maximize number of threads on GPU

# CUDA Elements

- cudaMalloc – for allocating memory in device
- cudaMemCopy – for copying data to allocated memory from host to device, and from device to host
- cudaFree – freeing allocated memory
- void syncthreads__() – synchronizing all threads in a block like barrier

# EXAMPLE 1: MATRIX VECTOR MULTIPLICATION

# Kernel

```
__global__ void matvec_mul(int m, int n, double *A, double *x,
    double *y)
{
  int row, col;
  double sum;

  row =blockIdx.x*blockDim.x+threadIdx.x;

  sum=0;
  if(row < m){
    for(col=0; col<n; col++){
      sum += A[row*n+col]*x[col];
    }
  }
  y[row]=sum;

}
```

# Host Program

```
int main(int argc, char** argv){
...
size_t size_A, size_x, size_y;
double *A, *x, *y;
double *dA, *dx, *dy;
...

  m = ... /* rows */; n = ... /* cols */

  size_A = sizeof(double)*m*n;  size_x = sizeof(double)*n;
       size_y = sizeof(double)*m;

  A = (double*)malloc(size_A);  x = (double*)malloc(size_x);  y =
       (double*)malloc(size_y);

  /* Allocate on the device memory */
  cudaMalloc((void **) &dA, size_A);
  cudaMalloc((void **) &dx, size_x);
  cudaMalloc((void **) &dy, size_y);

  /* Initialize A and x */
  /* Initialize y */
  for(i=0; i<m; i++) y[i] = 0;

```

# Host Program

```
  /* Copy A and x to the device */
  cudaMemcpy(dA, A, size_A , cudaMemcpyHostToDevice);
  cudaMemcpy(dx , x, size_x , cudaMemcpyHostToDevice);

  numThreadsPerBlock = 1024;  numBlocks = m/numThreadsPerBlock;

  dim3 dimGrid(numBlocks);
  dim3 dimBlock(numThreadsPerBlock);
  matvec_mul<<< dimGrid , dimBlock >>>(m,n,dA,dx,dy);

  cudaMemcpy(y, dy, size_y , cudaMemcpyDeviceToHost);

  cudaFree(dA); cudaFree(dx); cudaFree(dy);

  free(A); free(x); free(y);

}
```

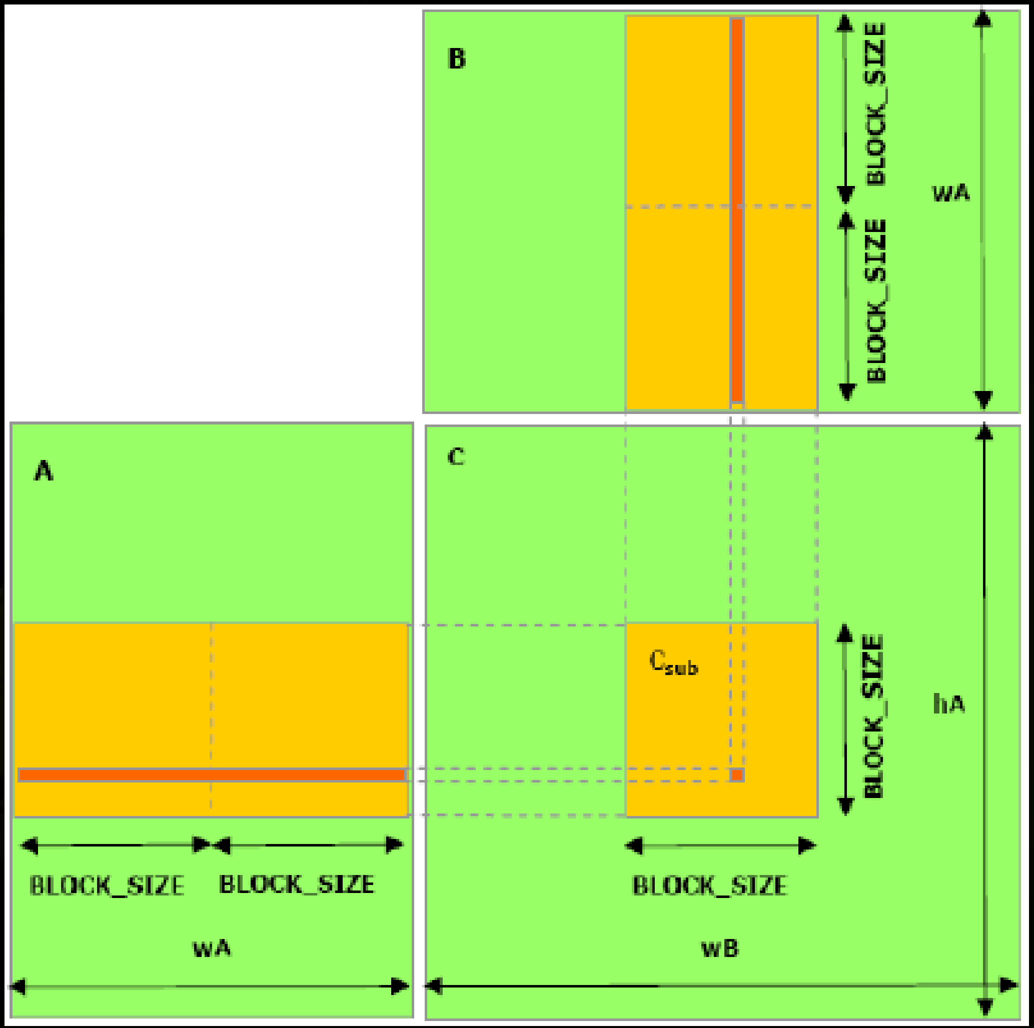
# EXAMPLE 1, VERSION 2: ACCESS FROM SHARED MEMORY

```
__global__ void matvec_mul(int m, int n, double *A, double *x,
    double *y)
{
  int row, col;
  double sum;

  __shared__ int sx[BLOCK_SIZE];

  sx[threadIdx.x] = x[threadIdx.x];
  __syncthreads();

  row =blockIdx.x*blockDim.x+threadIdx.x;

  sum=0;
  if(row < m){
    for(col=0; col<n; col++){
      sum += A[row*n+col]*sx[col];
    }
  }
  y[row]=sum;

}
```

# EXAMPLE 2: MATRIX MULTIPLICATION

# Matrix Multiplication Example

- Computing the product *C* of two matrices:
  *A : (wA, hA)*
  *B : (wB, wA).*
- Each thread block computes one square sub-matrix *Csub* of *C*;
- Each thread within the block computes one element of *Csub*.



# Example 1: Matrix Multiplication

## Host matrix multiplication code

```
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
   int size;
   // Load Input matrices A and B to the device
   float* Ad;
   size = hA * wA * sizeof(float);
   cudaMalloc((void**)&Ad, size);
   cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
   .
   // Allocate memory for output matrix C on the device
   float* Cd;
   size = hA * wB * sizeof(float);
   cudaMalloc((void**)&Cd, size);
```

# Example 1

```
// Compute the execution configuration assuming
// the matrix dimensions are multiples of BLOCK_SIZE
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);
// Launch the device computation
Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);
// Read Ouput matrix C from the device
cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(Ad);
.
}
```

# Example 1

## Device matrix multiplication function

```
__global__ void Muld ( float* A, float* B, int wA, int wB, float* C)
{
  // Setup aBegin, aEnd, aStep    bBegin, bStep based on Block index and Block size

 // The element of the block sub-matrix that is computed by the thread
   float Csub = 0;
 // Loop over all the sub-matrices of A and B required to compute the block sub-matrix
 for (int a = aBegin, b = bBegin;  a <= aEnd;  a += aStep, b += bStep) {

  // Shared memory for the sub-matrices of A and B
  __shared__  float As [ BLOCK_SIZE ] [ BLOCK_SIZE ];
  __shared__  float Bs [ BLOCK_SIZE ] [ BLOCK_SIZE ];
```

NVIDIA

# Example 1

```
// Load the matrices from global memory to shared memory; each thread loads one element of each matrix
As [ ty ] [ tx ] = A [ a + wA * ty + tx ];
Bs [ ty ] [ tx ] = B [ b + wB * ty + tx ];

// Synchronize to make sure the matrices are loaded
__syncthreads();

// Multiply the two matrices together; each thread computes one element/ of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += As[ty][k] * Bs[k][tx];
```

# Example 1

```
// Synchronize to make sure that the preceding computation is done before loading two new
// sub-matrices of A and B in the next iteration
__syncthreads();
}

// Write the block sub-matrix to global memory; each thread writes one element
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
}
```

# EXAMPLE 2: REDUCTION

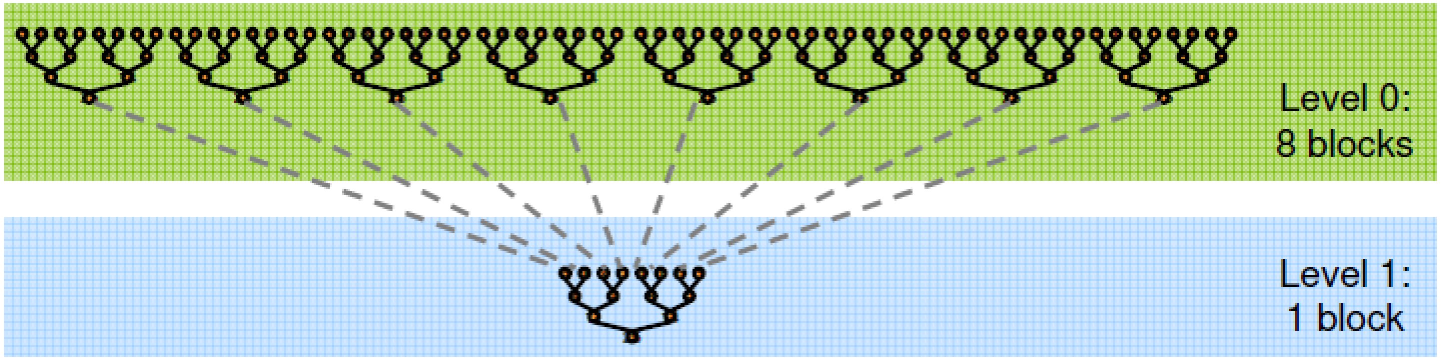
# Example: Reduction

- Tree based approach used within each thread block
- In this case, partial results need to be communicated across thread blocks
- Hence, global synchronization needed across thread blocks

# Reduction

- But CUDA does not have global synchronization –
  - expensive to build in hardware for large number of GPU cores
- Solution
  - Decompose into multiple kernels
  - Kernel launch serves as a global synchronization point

# Illustration



# Host Code

```
int main(){

int* h_idata, h_odata; /* host data*/
Int *d_idata, d_odata; /* device data*/

/* copying inputs to device memory */
cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice) ;
cudaMemcpy(d_odata, h_idata, numBlocks*sizeof(int),
    cudaMemcpyHostToDevice) ;

int numThreadsperBlock = (n < maxThreadsperBlock) ? n : maxThreadsperBlock;
int numBlocks = n / numThreadsperBlock;
dim3 dimBlock(numThreads, 1, 1); dim3 dimGrid(numBlocks, 1, 1);

reduce<<< dimGrid, dimBlock >>>(d_idata, d_odata);
```

# Host Code

```
int s=numBlocks;
while(s > 1) {
        numThreadsperBlock = (s< maxThreadsperBlock) ? s :
    maxThreadsperBlock;    numBlocks = s / numThreadsperBlock;
        dimBlock(numThreads, 1, 1);     dimGrid(numBlocks, 1, 1);
         reduce<<< dimGrid, dimBlock, smemSize >>>(d_idata,
    d_odata);
         s = s / numThreadsperBlock;
}

}
```

# Device Code

```
__global__ void reduce(int *g_idata, int *g_odata)
{
    extern __shared__ int sdata[];

    // load shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
      if ((tid % (2*s)) == 0)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

- For more information…
- CUDA SDK code samples – NVIDIA - http://www.nvidia.com/object/cuda_get_samples.html

# BACKUP

# EXAMPLE 3: SCAN

# Example: Scan or Parallel-prefix sum

**Definition:** *The* all-prefix-sums *operation takes a binary associative operator* $\oplus$, *and an array of n elements*

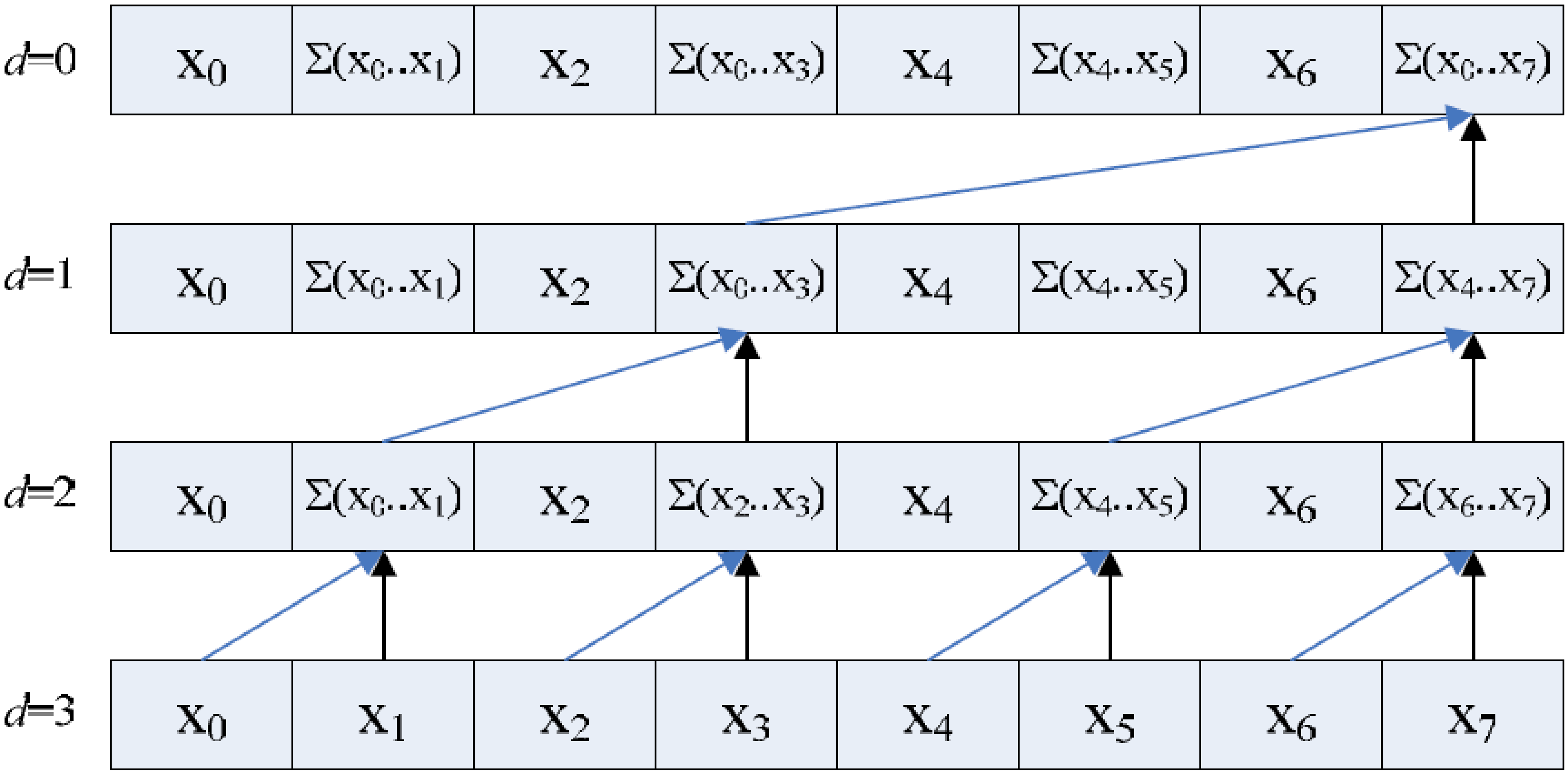
$$[a_0, a_1, \ldots, a_{n-1}],$$

*and returns the array*

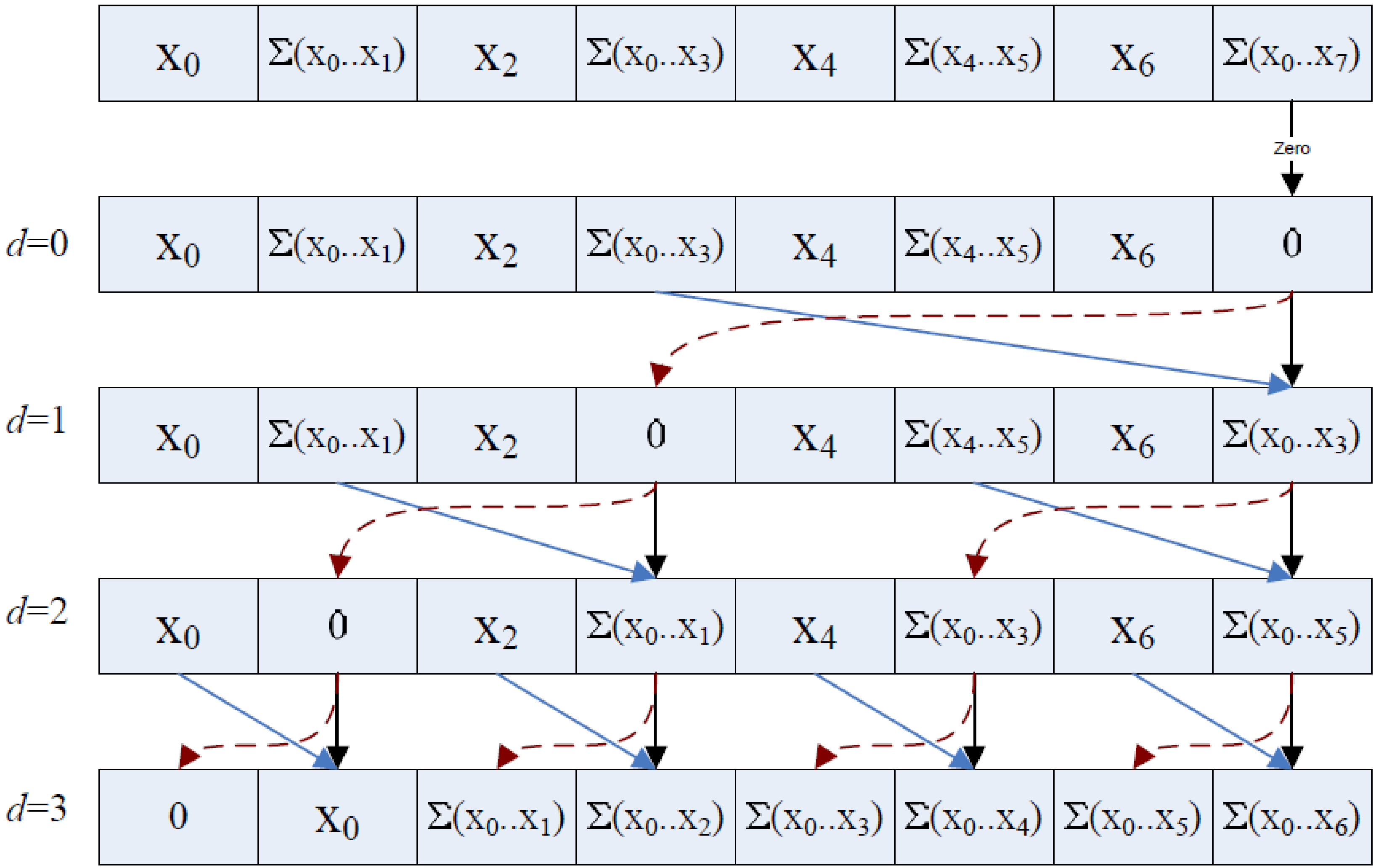
$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})].$$

- Using binary tree
- An upward reduction phase (reduce phase or up-sweep phase)
  - Traversing tree from leaves to root forming partial sums at internal nodes
- Down-sweep phase
  - Traversing from root to leaves using partial sums computed in reduction phase

# Up Sweep


$d$=0
$x_0$
$\Sigma(x_0..x_1)$
$x_2$
$\Sigma(x_0..x_3)$
$x_4$
$\Sigma(x_4..x_5)$
$x_6$
$\Sigma(x_0..x_7)$
$d$=1
$x_0$
$\Sigma(x_0..x_1)$
$x_2$
$\Sigma(x_0..x_3)$
$x_4$
$\Sigma(x_4..x_5)$
$x_6$
$\Sigma(x_4..x_7)$
$d$=2
$x_0$
$\Sigma(x_0..x_1)$
$x_2$
$\Sigma(x_2..x_3)$
$x_4$
$\Sigma(x_4..x_5)$
$x_6$
$\Sigma(x_6..x_7)$
$d$=3
$x_0$
$x_1$
$x_2$
$x_3$
$x_4$
$x_5$
$x_6$
$x_7$

# Down Sweep



x0
Σ(x0..x1)
x2
Σ(x0..x3)
x4
Σ(x4..x5)
x6
Σ(x0..x7)
Zero
d=0
x0
Σ(x0..x1)
x2
Σ(x0..x3)
x4
Σ(x4..x5)
x6
0
d=1
x0
Σ(x0..x1)
x2
0
x4
Σ(x4..x5)
x6
Σ(x0..x3)
d=2
x0
0
x2
Σ(x0..x1)
x4
Σ(x0..x3)
x6
Σ(x0..x5)
d=3
0
x0
Σ(x0..x1)
Σ(x0..x2)
Σ(x0..x3)
Σ(x0..x4)
Σ(x0..x5)
Σ(x0..x6)

# Host Code

- int main(){
- const unsigned int num_threads = num_elements / 2;
- /* cudaMalloc d_idata and d_odata */
- cudaMemcpy( d_idata, h_data, mem_size, cudaMemcpyHostToDevice) );
- dim3 grid(256, 1, 1); dim3 threads(num_threads, 1, 1);
- scan<<< grid, threads>>> (d_odata, d_idata);
- cudaMemcpy( h_data, d_odata[i], sizeof(float) * num_elements, cudaMemcpyDeviceToHost
- /* cudaFree d_idata and d_odata */
- }

# Device Code

```
__global__ void scan_workefficient(float *g_odata, float *g_idata, int n)
{
    // Dynamically allocated shared memory for scan kernels
    extern  __shared__  float temp[];

    int thid = threadIdx.x;    int offset = 1;

    // Cache the computational window in shared memory
    temp[2*thid]   = g_idata[2*thid];
    temp[2*thid+1] = g_idata[2*thid+1];

    // build the sum in place up the tree
    for (int d = n>>1; d > 0; d >>= 1)
    {
        __syncthreads();

        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            temp[bi] += temp[ai];
        }

        offset *= 2;
    }
```

# Device Code

```
// scan back down the tree

// clear the last element
if (thid == 0)        temp[n - 1] = 0;
// traverse down the tree building the scan in place
for (int d = 1; d < n; d *= 2)
{
    offset >>= 1;
    __syncthreads();
    if (thid < d)
    {
        int ai = offset*(2*thid+1)-1;
        int bi = offset*(2*thid+2)-1;

        float t   = temp[ai];
        temp[ai]  = temp[bi];
        temp[bi] += t;
    }
}

__syncthreads();

// write results to global memory
g_odata[2*thid]   = temp[2*thid];    g_odata[2*thid+1] = temp[2*thid+1];
}
```