

**nVIDIA®**

## **CUDA Parallel Programming Model**

Scalable Parallel Programming with CUDA

# Some Design Goals



- **Scale to 100s of cores, 1000s of parallel threads**
- **Let programmers focus on parallel algorithms**
  - *not* mechanics of a parallel programming language.
- **Enable heterogeneous systems (i.e., CPU+GPU)**
  - CPU & GPU are separate devices with separate DRAMs

# Key Parallel Abstractions in CUDA



- **Hierarchy of concurrent threads**
- **Lightweight synchronization primitives**
- **Shared memory model for cooperating threads**

# Hierarchy of concurrent threads



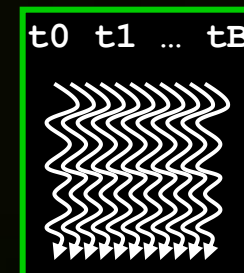
- Parallel **kernels** composed of many threads
  - all threads execute the same sequential program

Thread  $t$



- Threads are grouped into **thread blocks**
  - threads in the same block can cooperate

Block  $b$



- Threads/blocks have unique IDs

# Example: Vector Addition Kernel



## Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

```
int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

# Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

# Synchronization of blocks



- Threads within block may synchronize with **barriers**

```
... Step 1 ...  
__syncthreads ();  
... Step 2 ...
```

- Blocks **coordinate** via atomic memory operations

- e.g., increment shared queue pointer with **atomicInc()**

- Implicit barrier between **dependent kernels**

```
vec_minus<<<nblocks, blksize>>>(a, b, c);  
vec_dot<<<nblocks, blksize>>>(c, c);
```

# What is a thread?



- **Independent thread of execution**
  - has its own PC, variables (registers), processor state, etc.
  - no implication about how threads are scheduled
  
- **CUDA threads might be **physical** threads**
  - as on NVIDIA GPUs
  
- **CUDA threads might be **virtual** threads**
  - might pick 1 block = 1 physical thread on multicore CPU

# What is a thread block?

- Thread block = **virtualized multiprocessor**
  - freely choose processors to fit data
  - freely customize for each kernel launch
- Thread block = a (data) **parallel task**
  - all blocks in kernel have the same entry point
  - but may execute any code they want
- Thread blocks of kernel must be **independent** tasks
  - program valid for *any interleaving* of block executions

# Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: **OK**
  - shared lock: **BAD** ... can easily deadlock
- Independence requirement gives **scalability**

# Levels of parallelism

- **Thread parallelism**
  - each thread is an independent thread of execution
- **Data parallelism**
  - across threads in a block
  - across blocks in a kernel
- **Task parallelism**
  - different blocks are independent
  - independent kernels

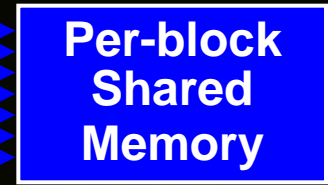
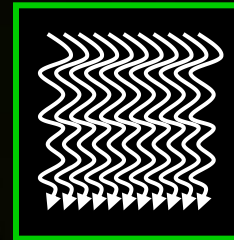
# Memory model



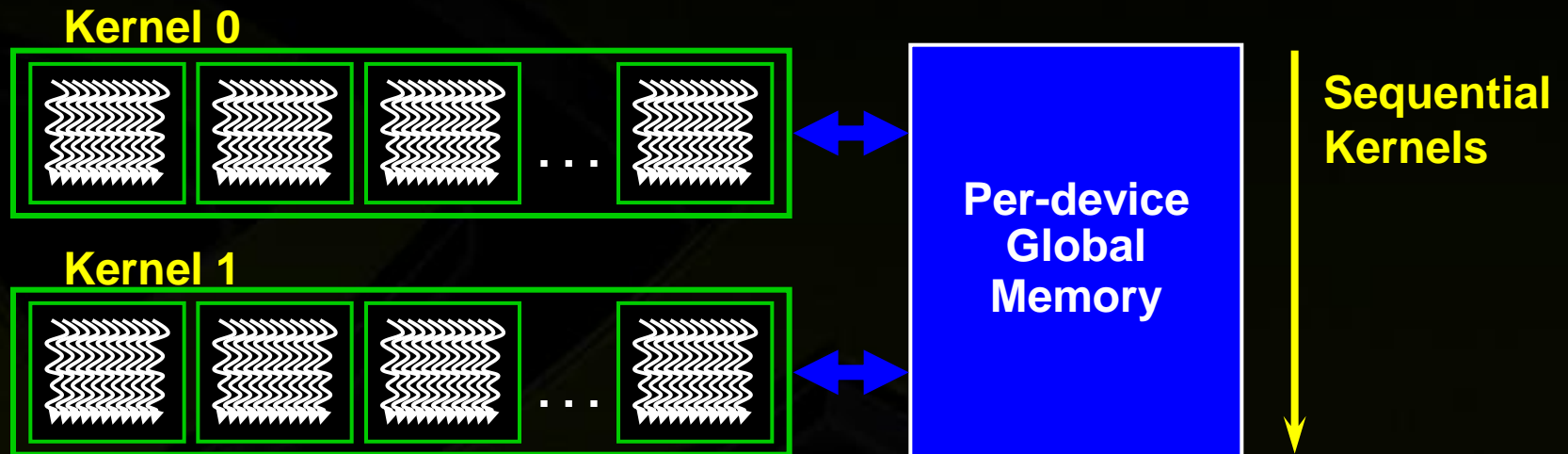
**Thread**



**Block**



# Memory model



# Memory model

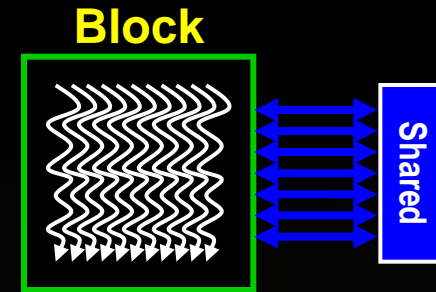


# Using per-block shared memory



- Variables shared across block

```
__shared__ int *begin, *end;
```



- Scratchpad memory

```
__shared__ int scratch[blocksize];  
scratch[threadIdx.x] = begin[threadIdx.x];  
// ... compute on scratch values ...  
begin[threadIdx.x] = scratch[threadIdx.x];
```

- Communicating values between threads

```
scratch[threadIdx.x] = begin[threadIdx.x];  
__syncthreads();  
int left = scratch[threadIdx.x - 1];
```

# CUDA: Minimal extensions to C/C++



- Declaration specifiers to indicate where things live

```
__global__ void KernelFunc(...); // kernel callable from host  
__device__ void DeviceFunc(...); // function callable on device  
__device__ int GlobalVar; // variable in device memory  
__shared__ int SharedVar; // in per-block shared memory
```

- Extend function invocation syntax for parallel kernel launch

```
KernelFunc<<<500, 128>>>(...); // 500 blocks, 128 threads each
```

- Special variables for thread identification in kernels

```
dim3 threadIdx; dim3 blockIdx; dim3 blockDim;
```

- Intrinsics that expose specific operations in kernel code

```
__syncthreads (); // barrier synchronization
```

# CUDA: GPU math and runtime libs



- **Standard mathematical functions**

`sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.

- **Atomic memory operations**

`atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.

- **Texture accesses in kernels**

```
texture<float,2> my_texture; // declare texture reference
```

```
float4 texel = texfetch(my_texture, u, v);
```

# CUDA: Runtime support



- **Explicit memory allocation returns pointers to GPU memory**  
`cudaMalloc()`, `cudaFree()`
- **Explicit memory copy for host ↔ device, device ↔ device**  
`cudaMemcpy()`, `cudaMemcpy2D()`, ...
- **Texture management**  
`cudaBindTexture()`, `cudaBindTextureToArray()`, ...
- **OpenGL & DirectX interoperability**  
`cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, ...

# Example: Vector Addition Kernel



```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

# Example: Host code for `vecAdd`

```
// allocate and initialize host (CPU) memory
```

```
float *h_A = ..., *h_B = ...;
```

```
// allocate device (GPU) memory
```

```
float *d_A, *d_B, *d_C;
```

```
cudaMalloc( (void**) &d_A, N * sizeof(float));
```

```
cudaMalloc( (void**) &d_B, N * sizeof(float));
```

```
cudaMalloc( (void**) &d_C, N * sizeof(float));
```

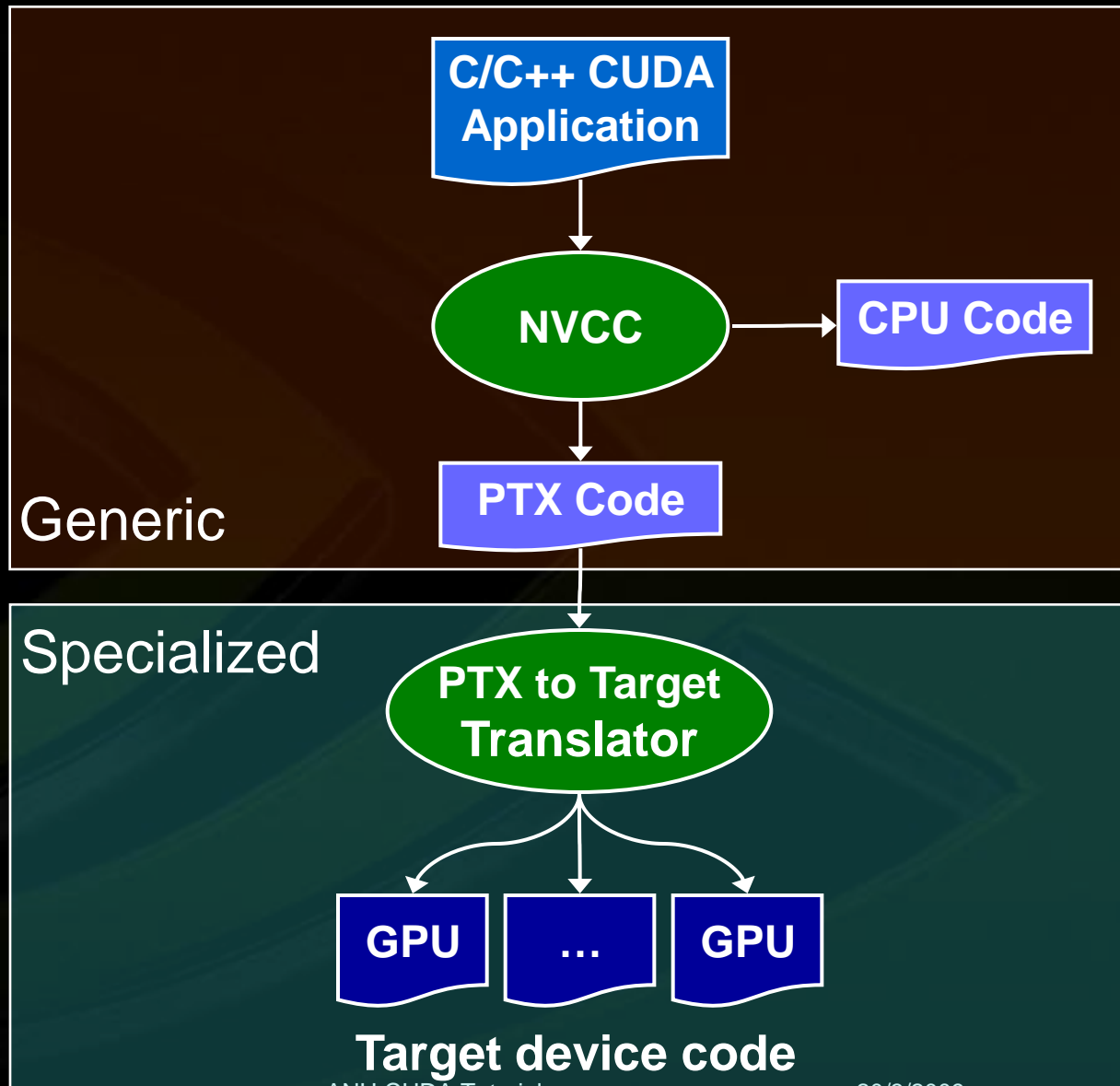
```
// copy host memory to device
```

```
cudaMemcpy( d_A, h_A, N * sizeof(float),  
            cudaMemcpyHostToDevice );
```

```
cudaMemcpy( d_B, h_B, N * sizeof(float),  
            cudaMemcpyHostToDevice );
```

```
// execute the kernel on N/256 blocks of 256 threads each  
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

# Compiling CUDA for GPUs



# Summing Up



- **CUDA = C + a few simple extensions**
  - makes it easy to start writing basic parallel programs
- **Three key abstractions:**
  1. hierarchy of parallel threads
  2. corresponding levels of synchronization
  3. corresponding memory spaces
- **Supports massive parallelism of manycore GPUs**

# Questions?

[mharris@nvidia.com](mailto:mharris@nvidia.com)

<http://www.nvidia.com/CUDA>

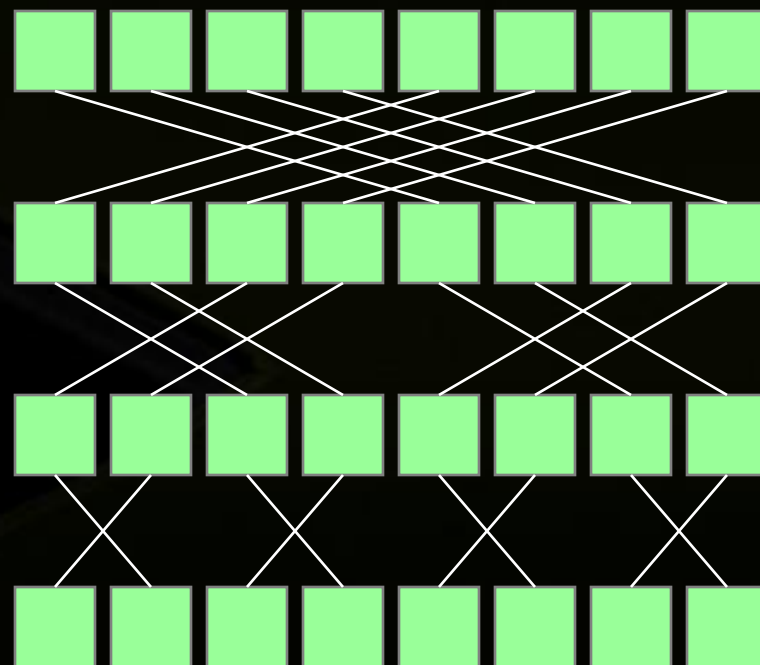
# Example: Parallel Reduction

- Summing up a sequence with 1 thread:

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

- Parallel reduction builds a summation tree

- each thread holds 1 element
- stepwise partial sums
- N threads need  $\log N$  steps
- one possible approach:  
Butterfly pattern



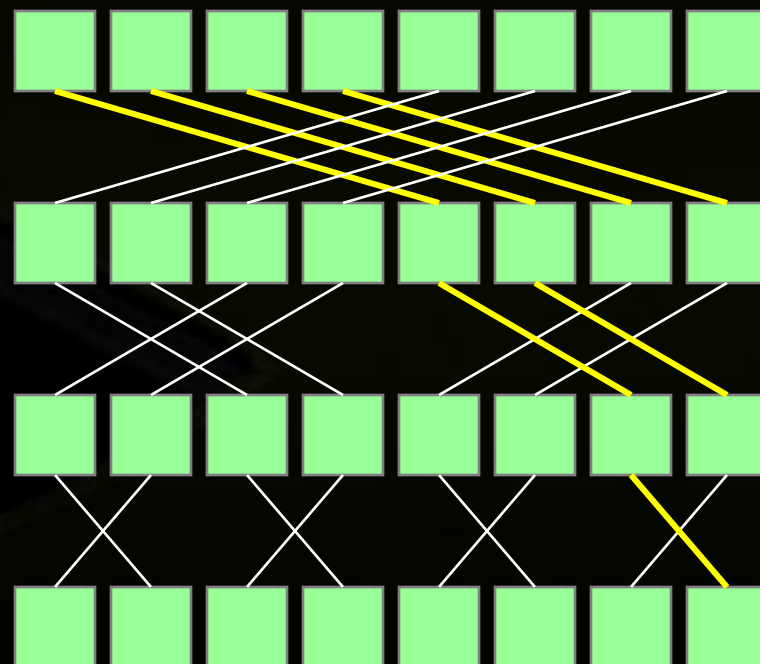
# Example: Parallel Reduction

- Summing up a sequence with 1 thread:

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

- Parallel reduction builds a summation tree

- each thread holds 1 element
- stepwise partial sums
- N threads need  $\log N$  steps
- one possible approach:  
Butterfly pattern



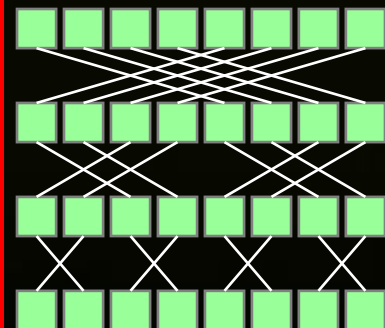
# Parallel Reduction for 1 Block

```
// INPUT: Thread i holds value x_i  
int i = threadIdx.x;  
__shared__ int sum[blocksize];
```

```
// One thread per element  
sum[i] = x_i; __syncthreads();
```

```
for(int bit=blocksize/2; bit>0; bit/=2)  
{  
    int t=sum[i]+sum[i^bit]; __syncthreads();  
    sum[i]=t; __syncthreads();  
}
```

```
// OUTPUT: Every thread now holds sum in sum[i]
```



# Example: Serial SAXPY routine

**Serial program:** compute  $y = \alpha x + y$  with a loop

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = a*x[i] + y[i];
}
```

**Serial execution:** call a function

```
saxpy_serial(n, 2.0, x, y);
```

# Example: Parallel SAXPY routine

**Parallel program:** compute with 1 thread per element

```
__global__  
void saxpy_parallel(int n, float a, float *x, float *y)  
{  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
  
    if( i<n )    y[i] = a*x[i] + y[i];  
}
```

**Parallel execution:** launch a kernel

```
uint size      = 256;           // threads per block  
uint blocks    = (n + size-1) / size; // blocks needed  
  
saxpy_parallel<<<blocks, size>>>(n, 2.0, x, y);
```

# SAXPY in PTX 1.0 ISA



```
cvt.u32.u16    $blockid, %ctaid.x; // Calculate i from thread/block IDs
cvt.u32.u16    $blocksize, %ntid.x;
cvt.u32.u16    $tid, %tid.x;
mad24.lo.u32   $i, $blockid, $blocksize, $tid;
ld.param.u32   $n, [N]; // Nothing to do if n ≤ i
setp.le.u32    $p1, $n, $i;
@$p1 bra      $L_finish;

mul.lo.u32     $offset, $i, 4; // Load y[i]
ld.param.u32   $yaddr, [Y];
add.u32        $yaddr, $yaddr, $offset;
ld.global.f32  $y_i, [$yaddr+0];
ld.param.u32   $xaddr, [X]; // Load x[i]
add.u32        $xaddr, $xaddr, $offset;
ld.global.f32  $x_i, [$xaddr+0];

ld.param.f32   $alpha, [ALPHA]; // Compute and store alpha*x[i] + y[i]
mad.f32        $y_i, $alpha, $x_i, $y_i;
st.global.f32  [$yaddr+0], $y_i;

$L_finish:    exit;
```