Parallel Programming and Debugging with CUDA C

Geoff Gerfin

Sr. System Software Engineer

NVIDIA

CUDA - NVIDIA's Architecture for GPU Computing

Broad Adoption

- Over 250M installed CUDA-enabled GPUs
- Over 650k CUDA Toolkit downloads in last 2 Yrs
- Windows, Linux and MacOS Platforms supported
- GPU Computing spans HPC to Consumer
- 350+ Universities teaching GPU Computing on the CUDA Architecture

GPU Computing Applications

CUDA C/C++
- Over 100k developers
- Running in Production since 2008
- SDK + Libs + Visual Profiler and Debugger

OpenCL
- Commercial OpenCL Conformant Driver
- Public Availability across all CUDA Architecture GPU's
- SDK + Visual Profiler

Direct Compute
- Microsoft API for GPU Computing
- Supports all CUDA-Architecture GPUs (DX10 and DX11)

Fortran
- PGI Accelerator
- PGI CUDA Fortran

Python, Java, .NET, …
- PyCUDA
- GPU.NET
- jCUDA

NVIDIA GPU
with the CUDA Parallel Computing Architecture

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

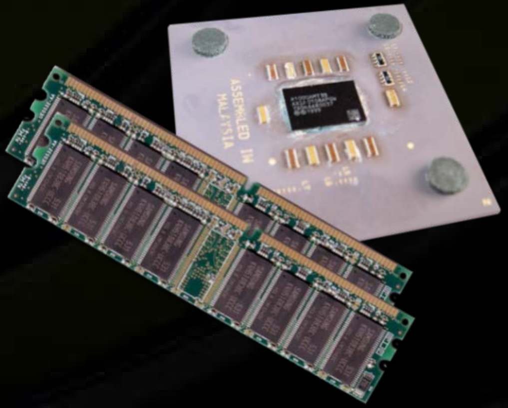
CUDA C

- **What will you learn today?**
  - **Write and launch CUDA C kernels**
  - **Manage GPU memory**
  - **Run parallel kernels in CUDA C**
  - **Parallel communication and synchronization**
  - **Debug with cuda-gdb, the Linux CUDA debugger**

CUDA C: The Basics

- Terminology
  - ***Host*** – The CPU and its memory (host memory)
  - ***Device*** – The GPU and its memory (device memory)

Host



Device

Note: *figure not to scale*

Hello, World!

```
int main( void ) {
    printf( "Hello, World!\n" );
    return 0;
}
```

- This basic program is just standard C that runs on the *host*
- NVIDIA's compiler, `nvcc`, will not complain about CUDA programs with no *device* code
- At its simplest, CUDA C is just C!

A Simple Example

- A simple kernel to add two integers:

```
__global__ void add( int *a, int *b, int *c ) {
    *c = *a + *b;
}
```

- CUDA C keyword `__global__` indicates that the `add()` function
  - Runs on the *device*
  - Called from *host* code

A Simple Example

- **Notice that we use pointers for our variables**

```
__global__ void add( int *a, int *b, int *c) {
    *c = *a + *b;
}
```

A Simple Example

- **Notice that we use pointers for our variables**

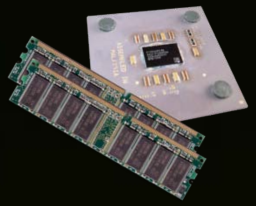
```
__global__ void add( int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- `add()` **runs on the device…so a, b, and c must point to device memory**

- **How do we allocate device memory?**

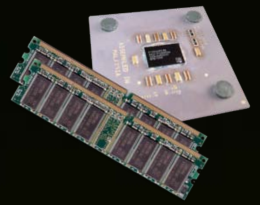
Memory Management

- Host and device memory are distinct entities
  - Device pointers point to GPU memory
    - May be passed to and from host code
    - May not be dereferenced from host code
  - Host pointers point to CPU memory
    - May be passed to and from device code
    - May not be dereferenced from device code



Memory Management

- Host and device memory are distinct entities
  - Device pointers point to GPU memory
    - May be passed to and from host code
    - May not be dereferenced from host code
  - Host pointers point to CPU memory
    - May be passed to and from device code
    - May not be dereferenced from device code



- Basic CUDA API for dealing with device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to their C equivalents, `malloc()`, `free()`, `memcpy()`

A Simple Example: add()

- **Using our** add() **kernel:**

```
__global__ void add( int *a, int *b, int *c ) {
    *c = *a + *b;
}
```

- **Let's take a look at** main()...

A Simple Example: `main()`

```
int main( void ) {
    int a, b, c;                    // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;     // device copies of a, b, c
    int size = sizeof( int );       // we need space for an integer
```

A Simple Example: `main()`

```
int main( void ) {
    int a, b, c;                    // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;     // device copies of a, b, c
    int size = sizeof( int );       // we need space for an integer

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );
```

A Simple Example: `main()`

```
int main( void ) {
    int a, b, c;                    // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;     // device copies of a, b, c
    int size = sizeof( int );       // we need space for an integer

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = 2;
    b = 7;
```

A Simple Example: `main()` (cont.)

```
    // copy inputs to device
    cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );

}
```

A Simple Example: `main()` (cont.)

```
// copy inputs to device
cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );

// launch add() kernel on GPU, passing parameters
add<<< 1, 1 >>>( dev_a, dev_b, dev_c );
```

A Simple Example: `main()` (cont.)

```
    // copy inputs to device
    cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );

    // launch add() kernel on GPU, passing parameters
    add<<< 1, 1 >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost );

}
```

A Simple Example: `main()` (cont.)

```
// copy inputs to device
cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );

// launch add() kernel on GPU, passing parameters
add<<< 1, 1 >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost );

cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
```

A Simple Example: `main()` (cont.)

```
    // copy inputs to device
    cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );

    // launch add() kernel on GPU, passing parameters
    add<<< 1, 1 >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost );

    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Parallel Programming in CUDA C

- But wait…GPU computing is about massive parallelism
- So how do we run code in parallel on the device?

Parallel Programming in CUDA C

- But wait…GPU computing is about massive parallelism
- So how do we run code in parallel on the device?
- Solution lies in the parameters between the triple angle brackets:

Parallel Programming in CUDA C

- **But wait…GPU computing is about massive parallelism**
- **So how do we run code in parallel on the device?**
- **Solution lies in the parameters between the triple angle brackets:**

```
add<<< 1, 1 >>>( dev_a, dev_b, dev_c );

add<<< N, 1 >>>( dev_a, dev_b, dev_c );
```

- **Instead of executing `add()` once, `add()` executed N times in parallel**

Parallel Programming in CUDA C

- **With** `add()` **running in parallel, let's do vector addition**
- **Terminology: Each parallel invocation of** `add()` **referred to as a** ***block***

Parallel Programming in CUDA C

- **With `add()` running in parallel, let's do vector addition**
- **Terminology: Each parallel invocation of `add()` referred to as a *block***
- **Kernel can refer to its block's index with variable `blockIdx.x`**
- **Each block adds a value from `a[]` and `b[]`, storing the result in `c[]`:**

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- **By using `blockIdx.x` to index arrays, each block handles different indices**

Parallel Programming in CUDA C

- **We write this code:**

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

Parallel Programming in CUDA C

- **We write this code:**

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- **This is what runs in parallel on the device:**

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

Parallel Addition: add()

- **Using our newly parallelized** add() **kernel:**

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- **Let's take a look at** main()**…**

Parallel Addition: `main()`

```
#define N  512
int main( void ) {
    int *a, *b, *c;                     // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;         // device copies of a, b, c
```

Parallel Addition: `main()`

```
#define N  512
int main( void ) {
    int *a, *b, *c;                          // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;              // device copies of a, b, c
    int size = N * sizeof( int );            // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition: `main()` (cont.)

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );
```

Parallel Addition: `main()` (cont.)

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch add() kernel with N parallel blocks
add<<< N, 1 >>>( dev_a, dev_b, dev_c );
```

Parallel Addition: `main()` (cont.)

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch add() kernel with N parallel blocks
add<<< N, 1 >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );
```

Parallel Addition: `main()` (cont.)

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch add() kernel with N parallel blocks
    add<<< N, 1 >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Threads

- **Terminology: A block can be split into parallel *threads***

- **Let's change vector addition to use parallel threads instead of parallel blocks:**

```
__global__ void add( int *a, int *b, int *c ) {
    c[ blockIdx.x ] = a[ blockIdx.x ] + b[ blockIdx.x ];
}
```

Threads

- Terminology: A block can be split into parallel *threads*

- Let's change vector addition to use parallel threads instead of parallel blocks:

```
__global__ void add( int *a, int *b, int *c ) {
    c[ threadIdx.x ] = a[ threadIdx.x ] + b[ threadIdx.x ];
}
```

- We use `threadIdx.x` instead of `blockIdx.x` in `add()`

Threads

- Terminology: A block can be split into parallel *threads*
- Let's change vector addition to use parallel threads instead of parallel blocks:

```
__global__ void add( int *a, int *b, int *c ) {
    c[ threadIdx.x ] = a[ threadIdx.x ] + b[ threadIdx.x ];
}
```

- We use `threadIdx.x` instead of `blockIdx.x` in `add()`
- `main()` will require one change as well…

Parallel Addition (Threads): `main()`

```
#define N  512
int main( void ) {
    int *a, *b, *c;                   // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;       // device copies of a, b, c
    int size = N * sizeof( int );     // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition (Threads): `main()` (cont.)

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );
```

Parallel Addition (Threads): `main()` (cont.)

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch add() kernel with N parallel threads
add<<< 1, N >>>( dev_a, dev_b, dev_c );
```

Parallel Addition (Threads): `main()` (cont.)

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch add() kernel with N parallel threads
    add<<< 1, N >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Why Bother With Threads?

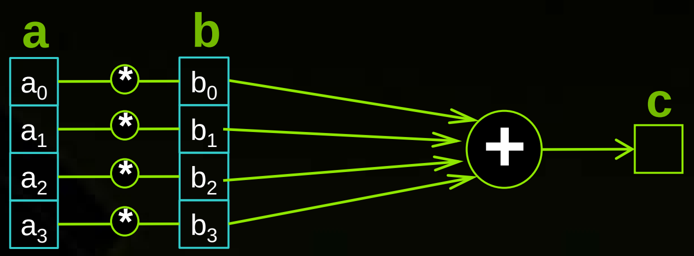
- Threads seem unnecessary
  - Added a level of abstraction and complexity
  - What did we gain?

Why Bother With Threads?

- Threads seem unnecessary
  - Added a level of abstraction and complexity
  - What did we gain?
- Unlike parallel blocks, parallel threads have mechanisms to:
  - Communicate
  - Synchronize
- Let’s see how…

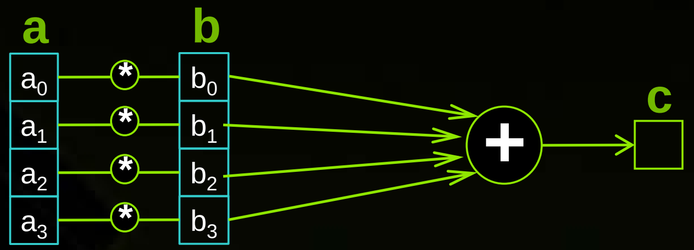
Dot Product

- Unlike vector addition, dot product is a *reduction* from vectors to a scalar



Dot Product

- Unlike vector addition, dot product is a *reduction* from vectors to a scalar



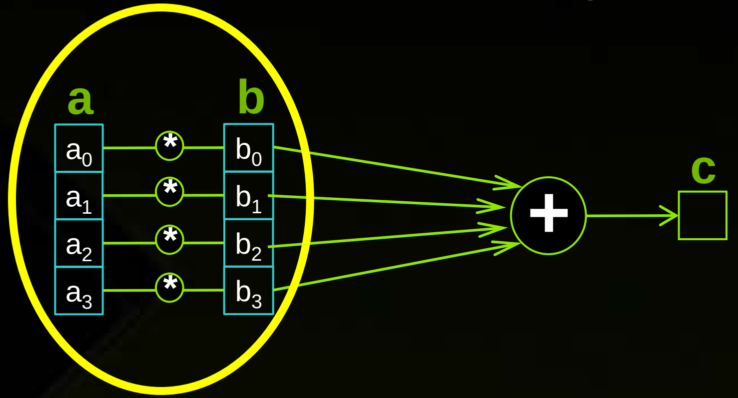
$$c = \vec{a} \cdot \vec{b}$$

$$c = (a_0, a_1, a_2, a_3) \cdot (b_0, b_1, b_2, b_3)$$

$$c = a_0\, b_0 + a_1\, b_1 + a_2\, b_2 + a_3\, b_3$$

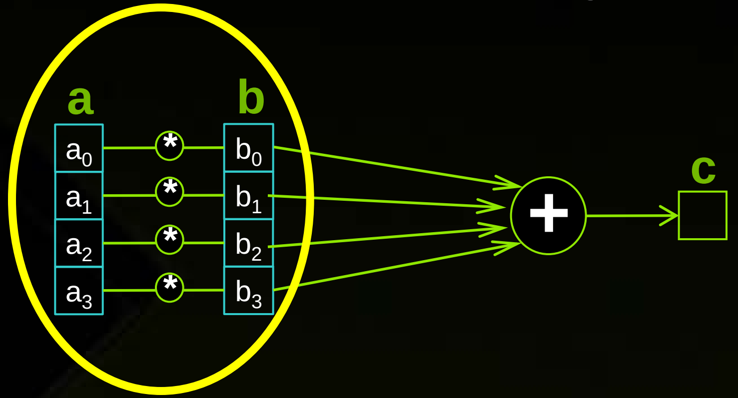
Dot Product

- **Parallel threads have no problem computing the pairwise products:**



Dot Product

- Parallel threads have no problem computing the pairwise products:

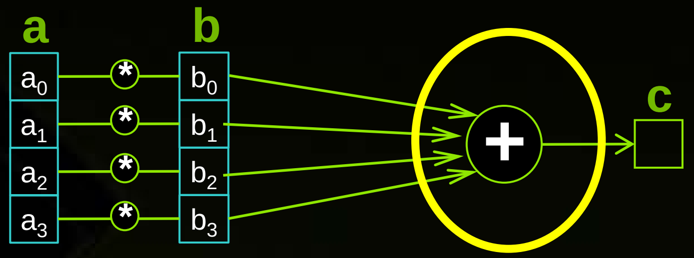


- So we can start a dot product CUDA kernel by doing just that:

```
__global__ void dot( int *a, int *b, int *c )   {
    // Each thread computes a pairwise product
    int temp = a[threadIdx.x] * b[threadIdx.x];
```

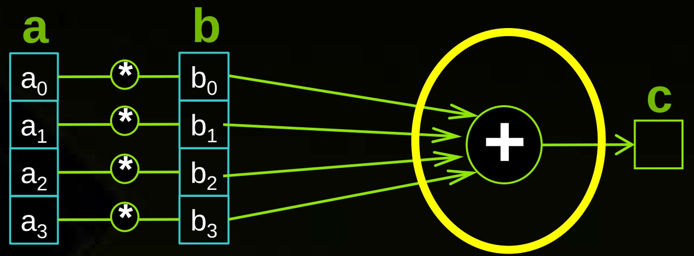
Dot Product

- **But we need to share data between threads to compute the final sum:**



Dot Product

- **But we need to share data between threads to compute the final sum:**




```
__global__ void dot( int *a, int *b, int *c )   {
   // Each thread computes a pairwise product
   int temp = a[threadIdx.x] * b[threadIdx.x];

   // Can’t compute the final sum
   // Each thread’s copy of ‘temp’ is private
}
```

Sharing Data Between Threads

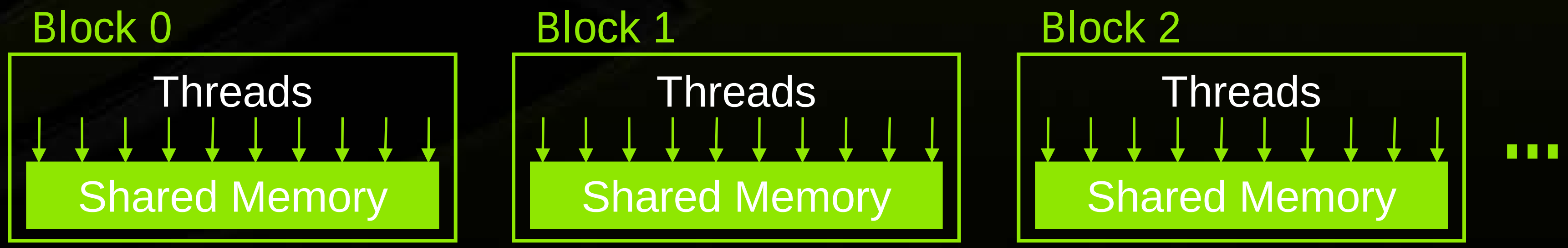
- Terminology: A block of threads shares memory called…

Sharing Data Between Threads

- **Terminology: A block of threads shares memory called…*shared memory***

Sharing Data Between Threads

- **Terminology: A block of threads shares memory called…*shared memory***
- **Extremely fast, on-chip memory (user-managed cache)**
- **Declared with the `__shared__` CUDA keyword**
- **Not visible to threads in other blocks running in parallel**



Parallel Dot Product: `dot()`

- **We perform parallel multiplication, serial addition:**

```
#define N 512
__global__ void dot( int *a, int *b, int *c )   {
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
```

Parallel Dot Product: `dot()`

- **We perform parallel multiplication, serial addition:**

```
#define N 512
__global__ void dot( int *a, int *b, int *c )   {
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // Thread 0 sums the pairwise products
    if ( 0 == threadIdx.x ) {
        int sum = 0;
        for (int i = 0; i < N; i++)
            sum += temp[i];
        *c = sum;
    }
}
```

Parallel Dot Product Recap

- We perform parallel, pairwise multiplications

Parallel Dot Product Recap

- We perform parallel, pairwise multiplications

- Shared memory stores each thread’s result

Parallel Dot Product Recap

- We perform parallel, pairwise multiplications
- Shared memory stores each thread's result
- We sum these pairwise products from a single thread
- Sounds good…

Parallel Dot Product Recap

- We perform parallel, pairwise multiplications
- Shared memory stores each thread's result
- We sum these pairwise products from a single thread
- Sounds good… but we've made a huge mistake

Enter the Debugger

- We will demonstrate how *cuda-gdb* can be used to find a bug in our `dot()` kernel

Enter the Debugger

- We will demonstrate how *cuda-gdb* can be used to find a bug in our `dot()` kernel
- The debugger follows CUDA language semantics when advancing program execution:

Enter the Debugger

- We will demonstrate how *cuda-gdb* can be used to find a bug in our `dot()` kernel

- The debugger follows CUDA language semantics when advancing program execution:
  - When single-stepping a CUDA thread, the entire *warp* it belongs to will single-step
  - A warp is a group of 32 CUDA threads

Enter the Debugger

- We will demonstrate how *cuda-gdb* can be used to find a bug in our `dot()` kernel

- The debugger follows CUDA language semantics when advancing program execution:
  - When single-stepping a CUDA thread, the entire *warp* it belongs to will single-step
  - A warp is a group of 32 CUDA threads

- Simply tracking how the program advances can reveal synchronization issues

Debugging with cuda-gdb

```
#define N 512
__global__ void dot( int *a, int *b, int *c )   {
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // Thread 0 sums the pairwise products
    if ( 0 == threadIdx.x ) {
        int sum = 0;
        for (int i = 0; i < N; i++)
            sum += temp[i];
        *c = sum;
    }
}
```

(cuda-gdb)

Debugging with cuda-gdb

```
#define N 512
__global__ void dot( int *a, int *b, int *c )   {
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // Thread 0 sums the pairwise products
    if ( 0 == threadIdx.x ) {
        int sum = 0;
        for (int i = 0; i < N; i++)
            sum += temp[i];
        *c = sum;
    }
}
```

(cuda-gdb) break dot

Debugging with cuda-gdb

```
#define N 512
__global__ void dot( int *a, int *b, int *c )   {
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // Thread 0 sums the pairwise products
    if ( 0 == threadIdx.x ) {
        int sum = 0;
        for (int i = 0; i < N; i++)
            sum += temp[i];
        *c = sum;
    }
}
```

(cuda-gdb) run

Debugging with cuda-gdb

```
1  #define N 512
2  __global__ void dot( int *a, int *b, int *c )   {
3      // Shared memory for results of multiplication
4      __shared__ int temp[N];
5      temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
6
7      // Thread 0 sums the pairwise products
8      if ( 0 == threadIdx.x ) {
9          int sum = 0;
10         for (int i = 0; i < N; i++)
11             sum += temp[i];
12         *c = sum;
13     }
14 }
```

(cuda-gdb) info cuda threads

<<<(0,0),(0,0,0)>>> … <<<(0,0),(511,0,0)>>> at dotproduct.cu:5

Debugging with cuda-gdb

```
#define N 512
__global__ void dot( int *a, int *b, int *c )   {
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // Thread 0 sums the pairwise products
    if ( 0 == threadIdx.x ) {
        int sum = 0;
        for (int i = 0; i < N; i++)
            sum += temp[i];
        *c = sum;
    }
}
```

(cuda-gdb) next

Debugging with cuda-gdb

```
#define N 512
__global__ void dot( int *a, int *b, int *c )   {
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // Thread 0 sums the pairwise products
    if ( 0 == threadIdx.x ) {
        int sum = 0;
        for (int i = 0; i < N; i++)
            sum += temp[i];
        *c = sum;
    }
}
```

(cuda-gdb) next

Debugging with cuda-gdb

```
#define N 512
__global__ void dot( int *a, int *b, int *c )   {
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // Thread 0 sums the pairwise products
    if ( 0 == threadIdx.x ) {
        int sum = 0;
        for (int i = 0; i < N; i++)
            sum += temp[i];
        *c = sum;
    }
}
```

(cuda-gdb) next

Debugging with cuda-gdb

```
#define N 512
__global__ void dot( int *a, int *b, int *c )   {
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // Thread 0 sums the pairwise products
    if ( 0 == threadIdx.x ) {
        int sum = 0;
        for (int i = 0; i < N; i++)
            sum += temp[i];
        *c = sum;
    }
}
```

(cuda-gdb) next

Debugging with cuda-gdb

```
1  #define N 512
2  __global__ void dot( int *a, int *b, int *c )   {
3      // Shared memory for results of multiplication
4      __shared__ int temp[N];
5      temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
6
7      // Thread 0 sums the pairwise products
8      if ( 0 == threadIdx.x ) {
9          int sum = 0;
10         for (int i = 0; i < N; i++)
11             sum += temp[i];
12         *c = sum;
13     }
14 }
```

```
(cuda-gdb) next

<<<(0,0),(0,0,0)>>> … <<<(0,0),(0,0,0)>>> at dotproduct.cu:11
<<<(0,0),(1,0,0)>>> … <<<(0,0),(31,0,0)>>> at dotproduct.cu:14
<<<(0,0),(32,0,0)>>> … <<<(0,0),(511,0,0)>>> at dotproduct.cu:5
```

Debugging with cuda-gdb

```
1   #define N 512
2   __global__ void dot( int *a, int *b, int *c )   {
3       // Shared memory for results of multiplication
4       __shared__ int temp[N];
5       temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
6       __syncthreads();
7       // Thread 0 sums the pairwise products
8       if ( 0 == threadIdx.x ) {
9           int sum = 0;
10          for (int i = 0; i < N; i++)
11              sum += temp[i];
12          *c = sum;
13      }
14  }
```

Threads 32 through 511 did not write out their results yet. To fix this bug, we need to synchronize all threads in this block.

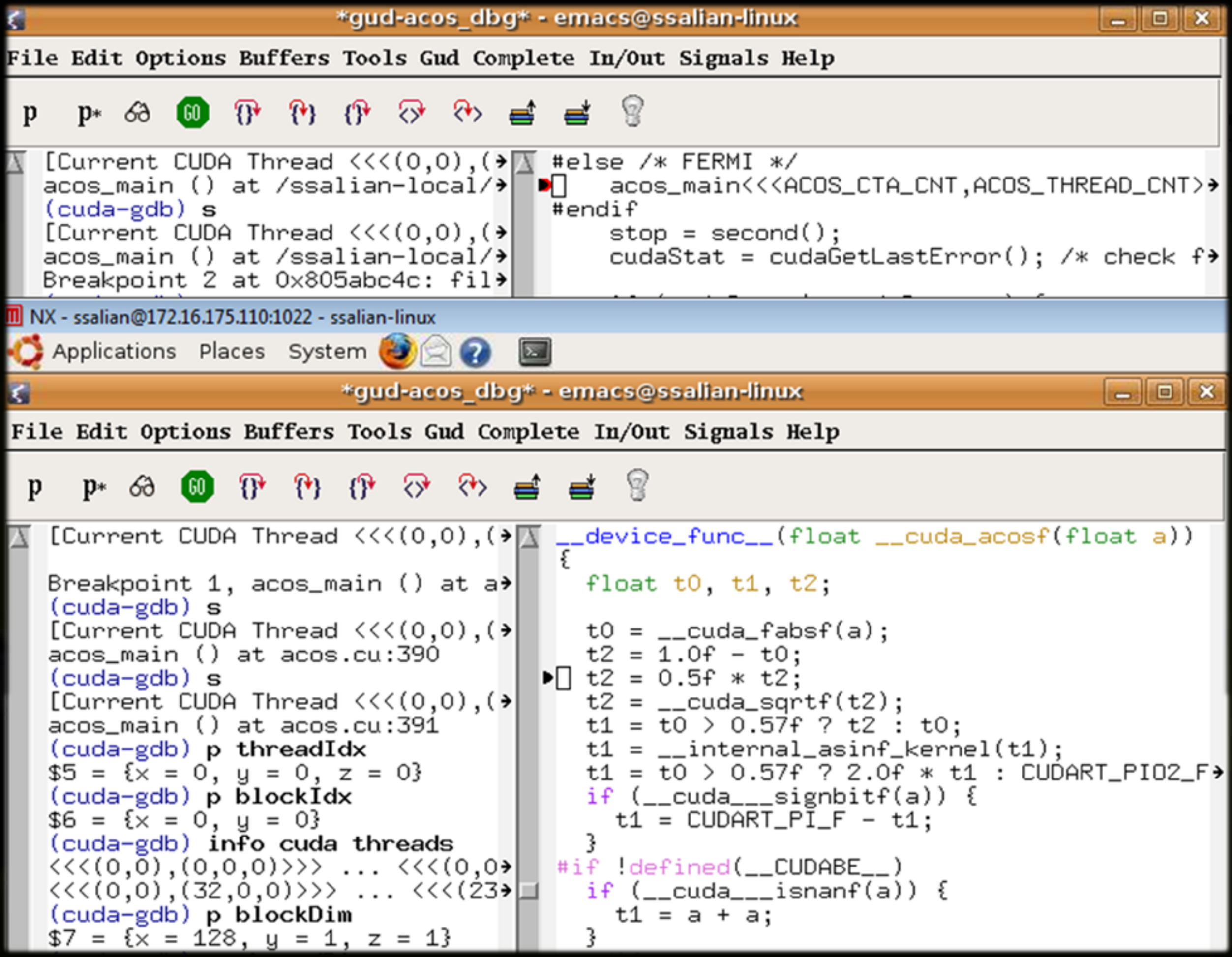
```
(cuda-gdb)  next

<<<(0,0),(0,0,0)>>> … <<<(0,0),(0,0,0)>>>  at dotproduct.cu:11
<<<(0,0),(1,0,0)>>> … <<<(0,0),(31,0,0)>>>  at dotproduct.cu:14
<<<(0,0),(32,0,0)>>> … <<<(0,0),(511,0,0)>>>  at dotproduct.cu:5
```

NVIDIA cuda-gdb

CUDA debugging integrated into GDB on Linux

- Supported on 32bit and 64bit systems
- Seamlessly debug both the host/CPU and device/GPU code
- Set breakpoints on any source line or symbol name
- Access and print all CUDA memory allocs, local, global, constant and shared vars

Included in the CUDA Toolkit



Applications Places System

File Edit View Terminal Help

```
linux64:~/demo2010$ ./ptrchecktest
unspecified launch failure : 79
linux64:~/demo2010$ cuda-memcheck ./ptrchecktest
========= CUDA-MEMCHECK
unspecified launch failure : 79
========= Invalid __global__ read of size 4
=========     at 0x00000158 in ptrchecktest.cu:27:kernel2
=========     by thread (0,0,0) in block (0,0)
=========     Address 0xfd00000001 is misaligned
=========
========= ERROR SUMMARY: 1 error
linux64:~/demo2010$ cuda-memcheck --continue ./ptrchecktest
========= CUDA-MEMCHECK
Checking...
Done
Checking...
Error: 3 (0)
Done
Checking...
Error: 1 (0)
Error: 3 (0)
Error: 5 (0)
Error: 7 (0)
Done
========= Invalid __global__ read of size 4
=========     at 0x00000158 in ptrchecktest.cu:27:kernel2
=========     by thread (0,0,0) in block (0,0)
=========     Address 0xfd00000001 is misaligned
=========
========= Invalid __global__ read of size 4
=========     at 0x00000198 in ptrchecktest.cu:18:kernel1
=========     by thread (3,0,0) in block (5,0)
=========     Address 0xfd00000028 is out of bounds
=========
========= Invalid __global__ write of size 8
=========     at 0x000001d0 in ptrchecktest.cu:38:kernel3
=========     by thread (1,0,0) in block (8,0)
=========     Address 0xfd00000204 is misaligned
=========
========= Invalid __global__ write of size 4
=========     at 0x000000f0 in ptrchecktest.cu:44:kernel4
=========     by thread (63,0,0) in block (22,0)
=========     Address 0x00000000 is out of bounds
=========
========= ERROR SUMMARY: 4 errors
```

cuda-memcheck

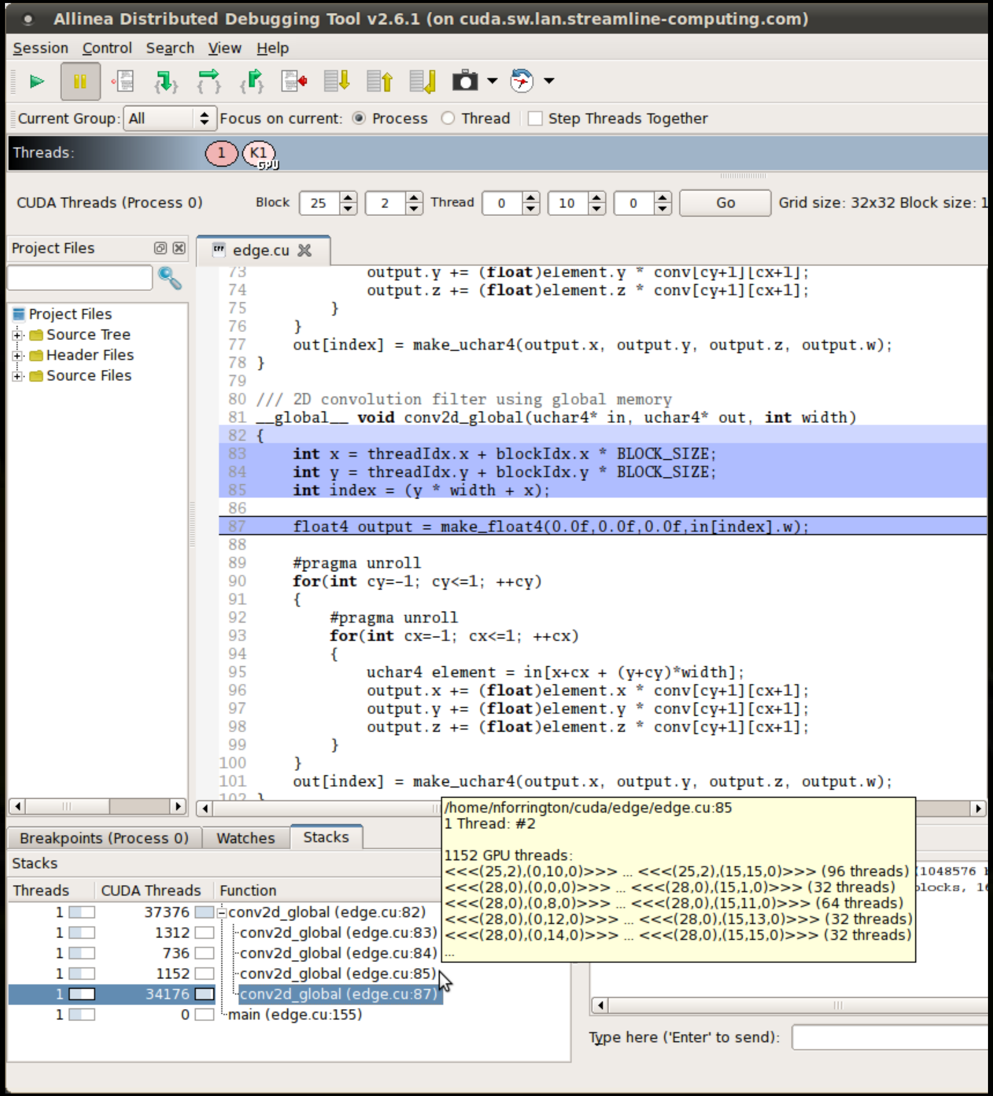
- **Detect memory and threading errors**
  - OOB memory accesses
  - Misaligned memory accesses
- **Windows, Linux, and Mac OSX**
- **Usage**
  - Standalone: cuda-memcheck <app>
  - cuda-gdb: set cuda memcheck on

Included in the CUDA Toolkit

Parallel Nsight for Visual Studio

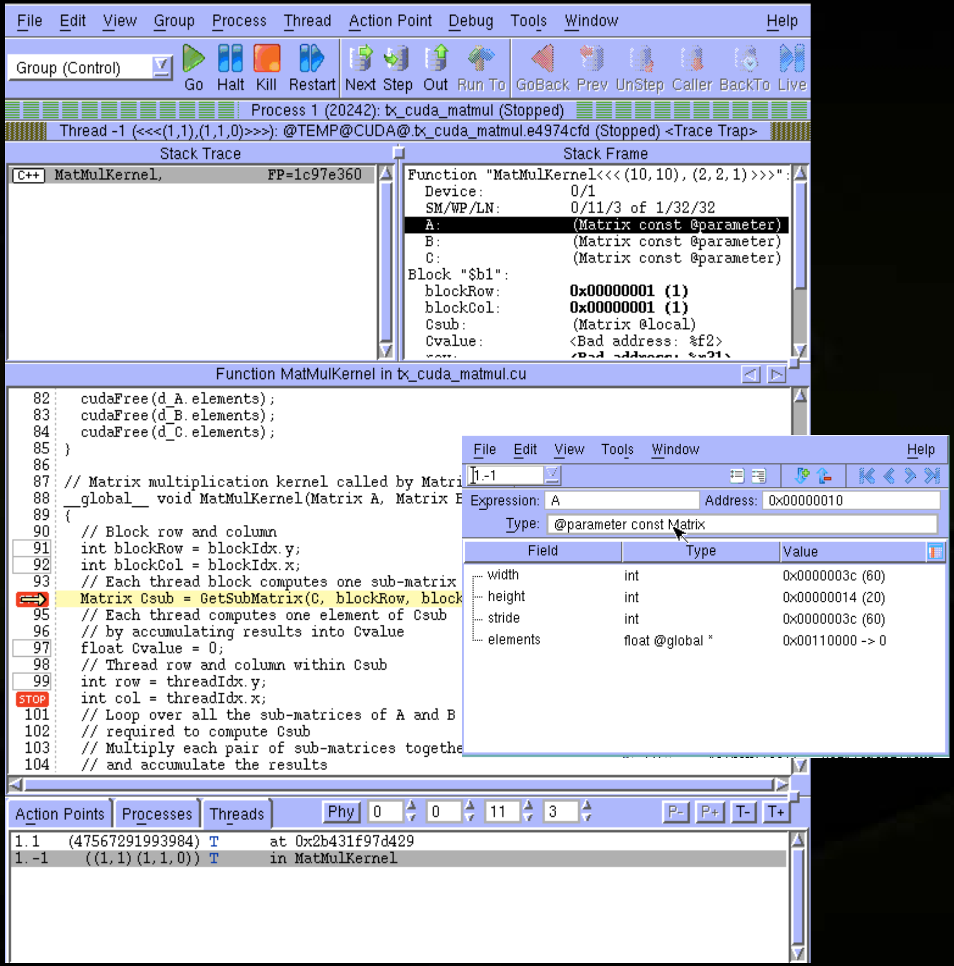
- **Integrated development for CPU and GPU**

Allinea DDT Debugger



- Fermi and Tesla support
- cuda-memcheck support for memory errors
- Combined MPI and CUDA support
- Stop on kernel launch feature
- Kernel thread control, evaluation and breakpoints
  - Identify thread counts, ranges and CPU/GPU threads easily
- Multi-Dimensional Array Viewer (MDA)
  - 3D Data Visualization
- Coming soon: multiple GPU device support

TotalView Debugger



- Full visibility of both Linux threads and GPU device threads
  - Device threads shown as part of the parent Unix process
  - Correctly handle all the differences between the CPU and GPU
- Fully represent the hierarchical memory
  - Display data at any level (registers, local, block, global or host memory)
  - Making it clear where data resides with type qualification
- Thread and Block Coordinates
  - Built in runtime variables display threads in a warp, block and thread dimensions and indexes
  - Displayed on the interface in the status bar, thread tab and stack frame
- Device thread control
  - Warps advance synchronously
- Handles CUDA function inlining
  - Step into or over inlined functions
- Reports memory access errors
  - CUDA memcheck
- Can be used with MPI

Questions?

- Latest CUDA Toolkit and Driver
  - http://www.nvidia.com/getcuda
- Additional Resources on CUDA from GTC 2010
  - http://www.nvidia.com/gtc
- PGI CUDA C for Multi-Core x86 Processors
  - Wednesday, 11/17 @ 1:00pm