Really Fast Introduction
to CUDA and CUDA C

Jul 2013

Dale Southard, NVIDIA
dsouthard@nvidia.com
About the Speaker

[Dale] is a senior solution architect with NVIDIA (I fix things). I primarily cover HPC in Gov/Edu/Research and cloud computing. In the past I was a HW architect in the LLNL systems group designing the vis/post-processing solutions.
The World Leader in Parallel Processing
Why CUDA
CUDA Accelerates Computing

Choose the right processor for the right task

CPU

Several sequential cores

CUDA GPU

Thousands of parallel cores
Power is THE problem

Underlying changes in silicon technology scaling mean the “free ride” of the past several decades is over. Power will be the dominant constraint in computing.
Estimated amount of power for an exascale system using today’s x86 processors, equivalent to the maximum output of the Hoover Dam.

DATA: U.S. Dept. of Energy
CPU
>1000 pJ/Instruction
Optimized for Latency
Caches

GPU
<200 pJ/Instruction
Optimized for Throughput
Explicit Management of On-chip Memory
The World’s Most Energy Efficient Supercomputer

3208 MFLOPS/Watt

128 Tesla K20 Accelerators

$100k Energy Savings / Yr

300 Tons of CO₂ Saved / Yr

CINECA Eurora

“Liquid-Cooled” Eurotech Aurora Tigon

Greener than Xeon Phi, Xeon CPU
Running NAMD version 2.9

The blue node contains Dual E5-2687W CPUs (150W each, 8 Cores per CPU).

The green nodes contain Dual E5-2687W CPUs (8 Cores per CPU) and 2x NVIDIA K10, K20, or K20X GPUs (235W each).

Energy Expended $= \text{Power} \times \text{Time}$

Cut down energy usage by $\frac{1}{2}$ with GPUs
### Tesla Kepler Family

**World’s Fastest and Most Efficient HPC Accelerators**

<table>
<thead>
<tr>
<th>GPUs</th>
<th>Single Precision Peak (SGEMM)</th>
<th>Double Precision Peak (DGEMM)</th>
<th>Memory Size</th>
<th>Memory Bandwidth (ECC off)</th>
<th>System Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>K20X</td>
<td>3.95 TF (2.90 TF)</td>
<td>1.32 TF (1.22 TF)</td>
<td>6 GB</td>
<td>250 GB/s</td>
<td>Server only</td>
</tr>
<tr>
<td>K20</td>
<td>3.52 TF (2.61 TF)</td>
<td>1.17 TF (1.10 TF)</td>
<td>5 GB</td>
<td>208 GB/s</td>
<td>Server + Workstation</td>
</tr>
<tr>
<td>K10</td>
<td>4.58 TF</td>
<td>0.19 TF</td>
<td>8 GB</td>
<td>320 GB/s</td>
<td>Server only</td>
</tr>
</tbody>
</table>

**Applications**
- **Weather & Climate, Physics, BioChemistry, CAE, Material Science**
- **Image, Signal, Video, Seismic**
<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Cores</td>
<td>2688</td>
</tr>
<tr>
<td>Single Precision</td>
<td>~4.5 Tflops</td>
</tr>
<tr>
<td>Double Precision</td>
<td>~1.27 Tflops</td>
</tr>
<tr>
<td>Memory Size</td>
<td>6GB</td>
</tr>
<tr>
<td>Memory B/W</td>
<td>288GB/s</td>
</tr>
</tbody>
</table>

GTX Titan: For High Performance Gaming Enthusiasts
What is CUDA
(six ways to saxpy)
Programming GPUs

Applications

Libraries
Directives
CUDA Programming

Easiest Approach for 2x to 10x Acceleration

Maximum Performance
Single precision **Alpha X Plus Y (SAXPY)**

Part of Basic Linear Algebra Subroutines (BLAS) Library

\[ z = \alpha x + y \]

\( x, y, z \) : vector
\( \alpha \) : scalar

GPU SAXPY in multiple languages and libraries

A menagerie* of possibilities, not a tutorial

*technically, a program chrestomathy: http://en.wikipedia.org/wiki/Chrestomathy
OpenACC Compiler Directives

**Parallel C Code**

```c
void saxpy(int n,
        float a,
        float *x,
        float *y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

... // Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...```

**Parallel Fortran Code**

```fortran
subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
!$acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
!$acc end kernels
end subroutine saxpy

... ! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...```

You can also call cuBLAS from Fortran, C++, Python, and other languages.
void saxpy(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);

__global__
void saxpy(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];
}

int N = 1<<20;

cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);
cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
Serial C++ Code
with STL and Boost

int N = 1<<20;
std::vector<float> x(N), y(N);

...  

// Perform SAXPY on 1M elements
std::transform(x.begin(), x.end(),
y.begin(), y.end(),
2.0f * _1 + _2);

www.boost.org/libs/lambda

Parallel C++ Code

int N = 1<<20;
thrust::host_vector<float> x(N), y(N);

...  

thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;

// Perform SAXPY on 1M elements
thrust::transform(d_x.begin(), d_x.end(),
d_y.begin(), d_y.begin(),
2.0f * _1 + _2);

http://thrust.github.com
CUDA Fortran

Standard Fortran

module mymodule contains
  subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n
    do i=1,n
      y(i) = a*x(i)+y(i)
    enddo
  end subroutine saxpy
end module mymodule

program main
  use mymodule
  real :: x(2**20), y(2**20)
  x = 1.0, y = 2.0
  call saxpy(2**20, 2.0, x, y)
end program main

Parallel Fortran

module mymodule contains
  subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n
    do i=1,n
      y(i) = a*x(i)+y(i)
    enddo
  end subroutine saxpy
end module mymodule

program main
  use cudafor; use mymodule
  real, device :: x_d(2**20), y_d(2**20)
  x_d = 1.0, y_d = 2.0
  call saxpy<<<4096,256>>>(2**20, 2.0, x_d, y_d)
end program main

Python

**Standard Python**

```python
import numpy as np

def saxpy(a, x, y):
    return [a * xi + yi for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)

cpu_result = saxpy(2.0, x, y)
```

**Copperhead: Parallel Python**

```python
from copperhead import *
import numpy as np

@cu
def saxpy(a, x, y):
    return [a * xi + yi for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)

with places.gpu0:
gpu_result = saxpy(2.0, x, y)

with places.openmp:
cpu_result = saxpy(2.0, x, y)
```

[http://numpy.scipy.org](http://numpy.scipy.org)
[http://copperhead.github.com](http://copperhead.github.com)
Enabling Endless Ways to SAXPY

Developers want to build front-ends for Java, Python, R, DSLs

Target other processors like ARM, FPGA, GPUs, x86

CUDA Compiler Contributed to Open Source LLVM
CUDA Basics
Heterogeneous Computing

- Terminology:
  - *Host*  The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)
```cpp
#include <iostream>
#include <algorithm>
using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__
void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex-RADIUS] = in[gindex-RADIUS];
        temp[lindex+BLOCK_SIZE] = in[gindex+BLOCK_SIZE];
    }

    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex+offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;
    // host copies of a, b, c
    int *d_in, *d_out;
    // device copies of a, b, c

    int size = (N + 2*RADIUS) * sizeof(int);
    // Alloc space for host copies and setup values
    in  = (int*)malloc(size);
    fill_ints(in,  N + 2*RADIUS);
    out = (int*)malloc(size);
    fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void**)&d_in,  size);
    cudaMalloc((void**)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out,  size, cudaMemcpyHostToDevice);

    // Launch kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in+RADIUS, d_out+RADIUS);

    // Copy result back to host
    cudaMemcpy(in, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in);
    cudaFree(d_out);
    return 0;
}
```

**Parallel Code**: The `stencil_1d` function is executed in parallel on the GPU. It reads input elements into shared memory, applies a stencil operation, and then stores the result back to the device.

**Serial Code**: The `fill_ints` function is executed serially on the CPU. It fills the input arrays with a constant value.

**Serial Code**: The `main` function allocates space for both host and device copies, copies data from the host to the device, calls the GPU kernel, copies the result back to the host, and then frees the allocated memory.
1. Copy input data from CPU memory to GPU memory
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
CUDA Kernels: Parallel Threads

- **kernel** is a function executed on the GPU as an array of threads in parallel.

- All threads execute the same code, can take different paths.

- Each thread has an ID:
  - Select input/output data
  - Control decisions

```c
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```
CUDA Kernels: Subdivide into Blocks

Threads
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into **blocks**
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into **blocks**
- Blocks are grouped into a **grid**
- A **kernel** is executed as a **grid of blocks of threads**
Kernel Execution

- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks’ memory requirements and the SM’s memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time
Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store memory that they all use
  - Share results with each other
  - Cooperate to produce a single result
  - Synchronize with each other
Thread blocks allow scalability

Blocks can execute in any order, concurrently or sequentially
This independence between blocks gives scalability:
  A kernel scales across any number of SMs
**Warps (extra credit)**

- Blocks are divided into 32 thread wide units called warps
  - Size of warps is implementation specific and can change in the future

- The SM creates, manages, schedules and executes threads at warp granularity
  - Each warp consists of 32 threads of contiguous threadIds

- All threads in a warp execute the same instruction
  - If threads of a warp diverge the warp serially executes each branch path taken

- When a warp executes an instruction that accesses global memory it coalesces the memory accesses of the threads within the warp into as few transactions as possible
Memory hierarchy

- Thread:
  - Registers
Memory hierarchy

Thread:
- Registers
- Local memory
Memory hierarchy

- **Thread:**
  - Registers
  - Local memory

- **Block of threads:**
  - Shared memory
Memory hierarchy

- **Thread:**
  - Registers
  - Local memory

- **Block of threads:**
  - Shared memory

- **All blocks:**
  - Global memory
CUDA C Basics

(finally)
Hello World!

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

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no device code

Output:

```
$ nvcc hello_world.cu
$ a.out
Hello World!
$ 
```
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

- Two new syntactic elements...
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}
```

CUDA C/C++ keyword `__global__` indicates a function that:
- Runs on the device
- Is called from host code

`nvcc` separates source code into host and device components
- Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
- Host functions (e.g. `main()`) processed by standard host compiler
  - `gcc`, `cl.exe`
mykernel<<<1,1>>>();

- **Triple angle brackets** mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - First parameter is the number of blocks
  - Second parameter is the number of threads in each block
  - Parameters can be scalars (int) or multidimensional (dim3)

- That’s all that is required to execute a function on the GPU!
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}

mykernel() does nothing, somewhat anticlimactic!

Output:

$ nvcc hello.cu
$ a.out
Hello World!
$
Parallel Programming in CUDA C/C++

- But wait... GPU computing is about massive parallelism!
- We need a more interesting example...
- We’ll start by adding two integers and build up to vector addition
Addition on the Device

A simple kernel to add two integers

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

As before `__global__` is a CUDA C/C++ keyword meaning

- `add()` will execute on the device
- `add()` will be called from the host
Addition on the Device

Note that we use pointers for the variables

```c
__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

We need to allocate memory on the GPU
Host and device memory are separate entities

*Device* pointers point to GPU memory
  - May be passed to/from host code
  - May *not* be dereferenced in host code

*Host* pointers point to CPU memory
  - May be passed to/from device code
  - May *not* be dereferenced in device code

Simple CUDA API for handling device memory

- `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- Similar to the C equivalents `malloc()`, `free()`, `memcpy()`
Addition on the Device: \texttt{add()}\n
Returning to our \texttt{add()} kernel

\begin{verbatim}
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
\end{verbatim}

Let's take a look at \texttt{main()}...
int main(void) {
    int a, b, c;    // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Setup input values
    a = 2;
    b = 7;
Addition on the Device: main()

// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
GPU computing is about massive parallelism

So how do we run code in parallel on the device?

Instead of executing `add()` once, execute N times in parallel.
Vector Addition on the Device

With `add()` running in parallel we can do vector addition.

Terminology: each parallel invocation of `add()` is referred to as a **block**.
- The set of blocks is referred to as a **grid**.
- Each invocation can refer to its block index using `blockIdx.x`.

```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 into the array, each block handles a different index.
Vector Addition on the Device

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

On the device, each block can execute in parallel:

- Block 0: $c[0] = a[0] + b[0]$;
Vector Addition on the Device: \texttt{add()}

Returning to our parallelized \texttt{add()} kernel

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

Let's take a look at \texttt{main()}...
Vector Addition on the Device: `main()`

```c
#define N 512
int main(void) {
    int *a, *b, *c;  // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```
Vector Addition on the Device: \texttt{main()}

```c
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
```
Difference between host and device

- **Host** CPU
- **Device** GPU

Using `__global__` to declare a function as device code

- Executes on the device
- Called from the host

Passing parameters from host code to a device function
Basic device memory management
- `cudaMalloc()`
- `cudaMemcpy()`
- `cudaFree()`

Launching parallel kernels
- Launch $N$ copies of `add()` with `add<<<N,1>>>(...)`;
- Use `blockIdx.x` to access block index
CUDA Threads

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

- Let’s change `add()` to use parallel **threads** instead of parallel **blocks**

```c
__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`

- Need to make one change in `main()`...
Vector Addition Using Threads: main()

#define N 512
int main(void) {
    int *a, *b, *c;                         // host copies of a, b, c
    int *d_A, *d_B, *d_C;                   // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMemcpy((void **)&d_A, size);
    cudaMemcpy((void **)&d_B, size);
    cudaMemcpy((void **)&d_C, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
Vector Addition Using Threads: `main()`

```c
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
```
Combining Blocks *and* Threads

- We’ve seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads

- Let’s adapt vector addition to use both *blocks* and *threads*

- Why? We’ll come to that…

- First let’s discuss data indexing…
Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
- Consider indexing an array with one element per thread (8 threads/block)

```c
int index = threadIdx.x + blockIdx.x * M;
```

With M threads/block a unique index for each thread is given by:
Which thread will operate on the red element?

```
int index = threadIdx.x + blockIdx.x * M;
=  5  +  2  * 8;
= 21;
```
Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block
  ```c
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  ```

- Combined version of `add()` to use parallel threads and parallel blocks
  ```c
  __global__ void add(int *a, int *b, int *c) {
      int index = threadIdx.x + blockIdx.x * blockDim.x;
      c[index] = a[index] + b[index];
  }
  ```

- What changes need to be made in `main()`?
Addition with Blocks and Threads: `main()`

```c
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;  // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
b = (int *)malloc(size); random_ints(b, N);
c = (int *)malloc(size);
}
```
Addition with Blocks and Threads: main()

// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

  ```c
  __global__ void add(int *a, int *b, int *c, int n) {
      int index = threadIdx.x + blockIdx.x * blockDim.x;
      if (index < n)
          c[index] = a[index] + b[index];
  }
  ```

- Update the kernel launch:

  ```c
  add<<<(N + M-1) / M, M>>>(d_a, d_b, d_c, N);
  ```
Why Bother with Threads?

 Threads seem unnecessary
   - They add a level of complexity
   - What do we gain?

 Unlike parallel blocks, threads have mechanisms to:
   - Communicate
   - Synchronize

 To look closer, we need a new example…
Need More?

• Get CUDA: www.nvidia.com/getcuda
• Nsight: www.nvidia.com/nsight
• Programming Guide/Best Practices… docs.nvidia.com
• Questions:
  • NVIDIA Developer forums devtalk.nvidia.com
  • Search or ask on www.stackoverflow.com/tags/cuda
• General: www.nvidia.com/cudazone
## Quantum Chemistry Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Features Supported</th>
<th>GPU Perf</th>
<th>Release Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abinit</strong></td>
<td>Local Hamiltonian, non-local Hamiltonian, LOBPCG algorithm, diagonalization / orthogonalization</td>
<td>1.3-2.7X</td>
<td>Released; Version 7.0.5 Multi-GPU support</td>
<td><a href="http://www.abinit.org">www.abinit.org</a></td>
</tr>
<tr>
<td><strong>ADF</strong></td>
<td>Fock Matrix, Hessians</td>
<td>TBD</td>
<td>Pilot project completed, Under development Multi-GPU support</td>
<td><a href="http://www.scrm.com">www.scrm.com</a></td>
</tr>
<tr>
<td><strong>Casino</strong></td>
<td></td>
<td>TBD</td>
<td>Under development, Spring 2013 release Multi-GPU support</td>
<td><a href="http://www.tcm.phy.cam.ac.uk/~mdt26/casino.html">http://www.tcm.phy.cam.ac.uk/~mdt26/casino.html</a></td>
</tr>
</tbody>
</table>

GPU Perf benchmarked on GPU supported features and may be a kernel to kernel perf comparison.
## Quantum Chemistry Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Features Supported</th>
<th>GPU Perf</th>
<th>Release Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPAW</td>
<td>Electrostatic poisson equation, orthonormalizing of vectors, residual minimization method (rmm-diis)</td>
<td>8x</td>
<td>Released Multi-GPU support</td>
<td><a href="https://wiki.fysik.dtu.dk/gpaw/devel/projects/gpu.html">https://wiki.fysik.dtu.dk/gpaw/devel/projects/gpu.html</a>, Samuli Hakala (CSC Finland) &amp; Chris O’Grady (SLAC)</td>
</tr>
<tr>
<td>MOLCAS</td>
<td>CU_BLAS support</td>
<td>1.1x</td>
<td>Released, Version 7.8 Single GPU. Additional GPU support coming in Version 8</td>
<td><a href="http://www.molcas.org">www.molcas.org</a></td>
</tr>
<tr>
<td>MOLPRO</td>
<td>Density-fitted MP2 (DF-MP2), density fitted local correlation methods (DF-RHF, DF-KS), DFT</td>
<td>Under development Multiple GPU</td>
<td><a href="http://www.molpro.net">www.molpro.net</a>, Hans-Joachim Werner</td>
<td></td>
</tr>
</tbody>
</table>

GPU Perf compared against Multi-core x86 CPU socket. GPU Perf benchmarked on GPU supported features and may be a kernel to kernel perf comparison.
# Quantum Chemistry Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Features Supported</th>
<th>GPU Perf</th>
<th>Release Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOPAC2009</td>
<td>pseudodiagonalization, full diagonalization, and density matrix assembling</td>
<td>3.8-14X</td>
<td>Under Development Single GPU</td>
<td>Academic port. <a href="http://openmopac.net">http://openmopac.net</a></td>
</tr>
<tr>
<td>Octopus</td>
<td>DFT and TDDFT</td>
<td>TBD</td>
<td>Released</td>
<td><a href="http://www.tddft.org/programs/octopus/">http://www.tddft.org/programs/octopus/</a></td>
</tr>
<tr>
<td>PEtot</td>
<td>Density functional theory (DFT) plane wave pseudopotential calculations</td>
<td>6-10X</td>
<td>Released Multi-GPU</td>
<td>First principles materials code that computes the behavior of the electron structures of materials</td>
</tr>
</tbody>
</table>

GPU Perf compared against Multi-core x86 CPU socket. 
GPU Perf benchmarked on GPU supported features and may be a kernel to kernel perf comparison.
# Quantum Chemistry Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Features Supported</th>
<th>GPU Perf</th>
<th>Release Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>QMCPACK</td>
<td>Main features</td>
<td>3-4x</td>
<td>Released Multiple GPUs</td>
<td>NCSA University of Illinois at Urbana-Champaign</td>
</tr>
<tr>
<td>Quantum Espresso/PWscf</td>
<td>PWscf package: linear algebra (matrix multiply), explicit computational kernels, 3D FFTs</td>
<td>2.5-3.5x</td>
<td>Released Version 5.0</td>
<td>Created by Irish Centre for High-End Computing</td>
</tr>
<tr>
<td>VASP</td>
<td>Hybrid Hartree-Fock DFT functionals including exact exchange</td>
<td>2x</td>
<td>Available on request</td>
<td>Multi-GPU support</td>
</tr>
<tr>
<td>WL-LSMS</td>
<td>Generalized Wang-Landau method</td>
<td>3x</td>
<td>Under development</td>
<td>Multi-GPU support</td>
</tr>
</tbody>
</table>

GPU Perf compared against Multi-core x86 CPU socket. GPU Perf benchmarked on GPU supported features and may be a kernel to kernel perf comparison.
<table>
<thead>
<tr>
<th>Related Applications</th>
<th>Features Supported</th>
<th>GPU Perf</th>
<th>Release Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amira 5®</td>
<td>3D visualization of volumetric data and surfaces</td>
<td>70x</td>
<td>Released, Version 5.3.3 Single GPU</td>
<td>Visualization from Visage Imaging. Next release, 5.4, will use GPU for general purpose processing in some functions <a href="http://www.visageimaging.com/overview.html">http://www.visageimaging.com/overview.html</a></td>
</tr>
<tr>
<td>BINDSURF</td>
<td>Allows fast processing of large ligand databases</td>
<td>100X</td>
<td>Available upon request to authors; single GPU</td>
<td>High-Throughput parallel blind Virtual Screening, <a href="http://www.biomedcentral.com/1471-2105/13/S14/S13">http://www.biomedcentral.com/1471-2105/13/S14/S13</a></td>
</tr>
<tr>
<td>BUDE</td>
<td>Empirical Free Energy Forcefield</td>
<td>6.5-13.4X</td>
<td>Released Single GPU</td>
<td>University of Bristol <a href="http://www.bris.ac.uk/biochemistry/cpfg/bude/bude.htm">http://www.bris.ac.uk/biochemistry/cpfg/bude/bude.htm</a></td>
</tr>
<tr>
<td>PyMol</td>
<td>Lines: 460% increase Cartoons: 1246% increase Surface: 1746% increase Spheres: 753% increase Ribbon: 436% increase</td>
<td>1700x</td>
<td>Released, Version 1.5 Single GPUs</td>
<td><a href="http://pymol.org/">http://pymol.org/</a></td>
</tr>
<tr>
<td>VMD</td>
<td>High quality rendering, large structures (100 million atoms), analysis and visualization tasks, multiple GPU support for display of molecular orbitals</td>
<td>100-125X or greater on kernels</td>
<td>Released, Version 1.9</td>
<td>Visualization from University of Illinois at Urbana-Champaign <a href="http://www.ks.uiuc.edu/Research/vmd/">http://www.ks.uiuc.edu/Research/vmd/</a></td>
</tr>
</tbody>
</table>

GPU Perf compared against Multi-core x86 CPU socket. GPU Perf benchmarked on GPU supported features and may be a kernel to kernel perf comparison.
<table>
<thead>
<tr>
<th>Application</th>
<th>Features Supported</th>
<th>GPU Speedup</th>
<th>Release Status</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarraCUDA</td>
<td>Alignment of short sequencing reads</td>
<td>6-10x</td>
<td>Version 0.6.2 - 3/2012 Multi-GPU, multi-node</td>
<td><a href="http://seqbarracuda.sourceforge.net/">http://seqbarracuda.sourceforge.net/</a></td>
</tr>
<tr>
<td>CUDASW++</td>
<td>Parallel search of Smith-Waterman database</td>
<td>10-50x</td>
<td>Version 2.0.8 - Q1/2012 Multi-GPU, multi-node</td>
<td><a href="http://sourceforge.net/projects/cudasw/">http://sourceforge.net/projects/cudasw/</a></td>
</tr>
<tr>
<td>CUSHAW</td>
<td>Parallel, accurate long read aligner for large genomes</td>
<td>10x</td>
<td>Version 1.0.40 - 6/2012 Multiple-GPU</td>
<td><a href="http://cushaw.sourceforge.net/">http://cushaw.sourceforge.net/</a></td>
</tr>
<tr>
<td>GPU-BLAST</td>
<td>Protein alignment according to BLASTP</td>
<td>3-4x</td>
<td>Version 2.2.26 - 3/2012 Single GPU</td>
<td><a href="http://eudoxus.cheme.cmu.edu/gpublast/gpublast.html">http://eudoxus.cheme.cmu.edu/gpublast/gpublast.html</a></td>
</tr>
<tr>
<td>GPU-HMMER</td>
<td>Parallel local and global search of Hidden Markov Models</td>
<td>60-100x</td>
<td>Version 2.3.2 - Q1/2012 Multi-GPU, multi-node</td>
<td><a href="http://www.mpihmmer.org/installguideGPUHMMER.htm">http://www.mpihmmer.org/installguideGPUHMMER.htm</a></td>
</tr>
<tr>
<td>mCUDA-MEME</td>
<td>Scalable motif discovery algorithm based on MEME</td>
<td>4-10x</td>
<td>Version 3.0.12 Multi-GPU, multi-node</td>
<td><a href="https://sites.google.com/site/yongchaosoftw/re/mcuda-meme">https://sites.google.com/site/yongchaosoftw/re/mcuda-meme</a></td>
</tr>
<tr>
<td>SeqNFind</td>
<td>Hardware and software for reference assembly, blast, SW, HMM, de novo assembly</td>
<td>400x</td>
<td>Released. Multi-GPU, multi-node</td>
<td><a href="http://www.seqnfind.com/">http://www.seqnfind.com/</a></td>
</tr>
<tr>
<td>UGENE</td>
<td>Fast short read alignment</td>
<td>6-8x</td>
<td>Version 1.11 - 5/2012 Multi-GPU, multi-node</td>
<td><a href="http://ugene.unipro.ru/">http://ugene.unipro.ru/</a></td>
</tr>
<tr>
<td>WideLM</td>
<td>Parallel linear regression on multiple similarly-shaped models</td>
<td>150x</td>
<td>Version 0.1-1 - 3/2012 Multi-GPU, multi-node</td>
<td><a href="http://insilicos.com/products/widelm">http://insilicos.com/products/widelm</a></td>
</tr>
</tbody>
</table>

GPU Perf compared against same or similar code running on single CPU machine
Performance measured internally or independently
Overview

- GPU performance and scalability across multiple scientific domains & multiple algorithmic motifs
- Different approaches taken (libraries, OpenACC, programming languages)
- Lessons learned
LSMS – Materials Science

LSMS
Fe32 (32 atoms per node)
Relative Performance (FLOPS) vs. Dual-Socket E5-2687w 3.10 GHz Sandy Bridge

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Relative Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xCPU</td>
<td>0.6</td>
</tr>
<tr>
<td>2xCPU</td>
<td>1.0</td>
</tr>
<tr>
<td>1xCPU+1xGPU</td>
<td>3.2</td>
</tr>
<tr>
<td>1xCPU+2xGPU</td>
<td>6.1</td>
</tr>
<tr>
<td>2xCPU+1xGPU</td>
<td>1.3</td>
</tr>
<tr>
<td>2xCPU+2xGPU</td>
<td>2.6</td>
</tr>
<tr>
<td>K20X Single-Socket</td>
<td>3.2</td>
</tr>
<tr>
<td>M2090 Dual-Socket</td>
<td>6.2</td>
</tr>
</tbody>
</table>

Preliminary, NVIDIA Confidential – not for distribution
LSMS Multi-Node Scalability

Fe32 (32 atoms per node)
GFLOPS vs. Dual-Socket E5-2687w 3.10 GHz Sandy Bridge

5x vs. Dual-Socket Sandy Bridge @8 nodes

1 M2090/node
2 M2090/node
1 K20X/node
2 K20X/node

GFLOPs (log scale)

# Nodes (log scale)
gWL-LSMS – Materials Science

**gWL-LSMS**
Fe1024 (1024 atoms per 32 nodes)
Relative Scaling (FLOPS) on Cray XK7

"CPU" node = XK7 (1x Interlagos)
"M2090" node = XK6 (1x M2090 + 1x Interlagos)
"K20X" node = XK7 (1x K20X + 1x Interlagos)

![Graph showing relative scaling (FLOPS) on Cray XK7](image)

- **7.01x vs. CPU @64 nodes**

Preliminary, NVIDIA Confidential – not for distribution
Application Power Efficiency of the Cray XK7 WL-LSMS for CPU-only and Accelerated Computing

Power consumption traces for identical WL-LSMS runs with 1024 Fe atoms, 16 atoms/node, on 18,561 Titan nodes (99% of Titan)

- Runtime is **8.6X** faster for the accelerated code
- Energy consumed is **7.3X** less
  - GPU accelerated code consumed **3,500 kW-hr**
  - CPU only code consumed **25,700 kW-hr**
### SPECFEM3D

**meshfed3D-256x128x15**

Relative Performance (solver time) vs. E5-2687w 3.10 GHz Sandy Bridge

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Relative to 2x CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xCPU</td>
<td>0.66</td>
</tr>
<tr>
<td>2xCPU</td>
<td>1.00</td>
</tr>
<tr>
<td>1xCPU+1xGPU</td>
<td>8.85</td>
</tr>
<tr>
<td>1xCPU+2xGPU</td>
<td>13.80</td>
</tr>
<tr>
<td>2xCPU+1xGPU</td>
<td>5.41</td>
</tr>
<tr>
<td>2xCPU+2xGPU</td>
<td>9.27</td>
</tr>
<tr>
<td>2xCPU+1xGPU</td>
<td>8.84</td>
</tr>
<tr>
<td>2xCPU+2xGPU</td>
<td>12.83</td>
</tr>
</tbody>
</table>
SPECFEM3D on Sandy Bridge

512x512x20, 5 M Elastic cells, 1000 timesteps
Relative Scaling (solver time)

“K20X” node = XK7 (1x K20X + 1x Interlagos)
“XE6” node = XE6 (2x Interlagos)
“XC30” node = XC30 (2x Sandy Bridge)

K20X
2.75x vs. XC30
@256 nodes
SPECFEM3D – Earth Science

SPECFEM3D on Titan
2048x2048x15, 33 M cells, 2000 timesteps
Relative Scaling (solver time) on Cray XK7

"CPU" node = XK7 (1x Interlagos)
“K20X” node = XK7 (1x K20X + 1x Interlagos)

2.45x vs. CPU @2048 nodes
Chroma (Lattice QCD) – High Energy & Nuclear Physics

**Chroma**

24³x128 lattice
Relative Performance (Propagator) vs. E5-2687w 3.10 GHz Sandy Bridge

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>1xCPU+ 1xGPU</th>
<th>1xCPU+ 2xGPU</th>
<th>2xCPU+ 1xGPU</th>
<th>2xCPU+ 2xGPU</th>
<th>2xCPU+ 1xGPU</th>
<th>2xCPU+ 2xGPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1xCPU</td>
<td>3.7</td>
<td>6.8</td>
<td>2.8</td>
<td>5.5</td>
<td>3.5</td>
<td>6.7</td>
</tr>
<tr>
<td>2xCPU</td>
<td>0.5</td>
<td>1.0</td>
<td>K20X Single-Socket</td>
<td>M2090 Dual-Socket</td>
<td>K20X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Preliminary, NVIDIA Confidential – not for distribution
Chroma (Lattice QCD) – High Energy & Nuclear Physics

**Chroma**
48³x512 lattice
Relative Scaling (Application Time)

"XK7" node = XK7 (1x K20X + 1x Interlagos)
"XE6" node = XE6 (2x Interlagos)

- XE6 (2x Interlagos)
- XK7 (K20X) (BiCGStab)
- XK7 (K20X) (DD+GCR)

0 2 4 6 8 10 12 14 16 18
0 128 256 384 512 640 768 896 1024 1152 1280

Relative Scaling vs. # Nodes

3.58x vs. XE6 @ 1152 nodes

Preliminary, NVIDIA Confidential – not for distribution
QMCPACK – Materials Science

QMCPACK
Graphite 4x4x1, E5-2687w 3.10 GHz Sandy Bridge

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Single-Socket</th>
<th>Dual-Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xCPU</td>
<td>0.44</td>
<td>1.00</td>
</tr>
<tr>
<td>2xCPU</td>
<td>2.86</td>
<td>4.05</td>
</tr>
<tr>
<td>1xCPU+1xGPU M2090</td>
<td>2.84</td>
<td>5.60</td>
</tr>
<tr>
<td>1xCPU+1xGPU K20X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2xCPU+1xGPU M2090</td>
<td>4.01</td>
<td></td>
</tr>
<tr>
<td>2xCPU+1xGPU K20X</td>
<td></td>
<td>7.97</td>
</tr>
<tr>
<td>2xCPU+2xGPU M2090</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2xCPU+2xGPU K20X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Relative to 2x CPU

Preliminary, NVIDIA Confidential – not for distribution
QMCPACK – Materials Science

QMCPACK
3x3x1 Graphite
Relative Scaling (Efficiency) on Cray XK7

“CPU” node = XK7 (1x Interlagos)
“K20X” node = XK7 (1x K20X + 1x Interlagos)

3.97x vs. CPU @2048 nodes
**Single-node Performance Summary**

**Relative Performance vs. dual-socket Sandy Bridge**

E5-2687w 3.10 GHz Sandy Bridge

<table>
<thead>
<tr>
<th>Application</th>
<th>Configuration</th>
<th>1xCPU + 2xK20X</th>
<th>1xCPU + K20X</th>
<th>2xCPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECFEM3D</td>
<td></td>
<td>1.0</td>
<td>8.9</td>
<td>13.8</td>
</tr>
<tr>
<td>AMBER</td>
<td></td>
<td>1.0</td>
<td>7.2</td>
<td>8.5</td>
</tr>
<tr>
<td>Chroma</td>
<td></td>
<td>1.0</td>
<td>3.7</td>
<td>6.8</td>
</tr>
<tr>
<td>WL-LSMS</td>
<td></td>
<td>1.0</td>
<td>3.2</td>
<td>6.1</td>
</tr>
<tr>
<td>NAMD</td>
<td></td>
<td>2.5</td>
<td>3.5</td>
<td>8.5</td>
</tr>
</tbody>
</table>

---

Preliminary, NVIDIA Confidential – not for distribution
Summary

- Performance
- Scalability
- Variety of programming approaches
- Futures