

Introduction to Cuda

By Alan T. Andrea
AT. Andrea Technologies

This paper can be found at:
[http://208.111.39.113/cuda/Introduction to Cuda.pptx](http://208.111.39.113/cuda/Introduction%20to%20Cuda.pptx)

What is CUDA and what is its purpose

- CUDA stands for: **Compute Unified Device Architecture**

It was developed by NVIDIA corporation and is a framework for allowing developers to build code in C/C++ to run on NVIDIA's CUDA enabled GPU devices. This allows you to essentially run code on a massively parallel gpu environment providing that you do some clever work to figure out how to partition your problem into a structure that can be distributed on a parallel environment.

- The advantage of doing GPGPU (General purpose computing on a GPU) is that you have access to a device that can run many concurrent threads (processes) in parallel. Many more than would be allowed in a multi-threaded CPU environment.

Many NVIDIA cards (e.g.: GeForce, Quadro, Tesla) support cuda and SDK tool-kits are available for Mac, Windows, and Linux. Therefore, the solution is widely available.

There are also wrappers available to facilitate access to the CUDA framework for non-C/C++ programmers. Wrappers are, for example, available for:

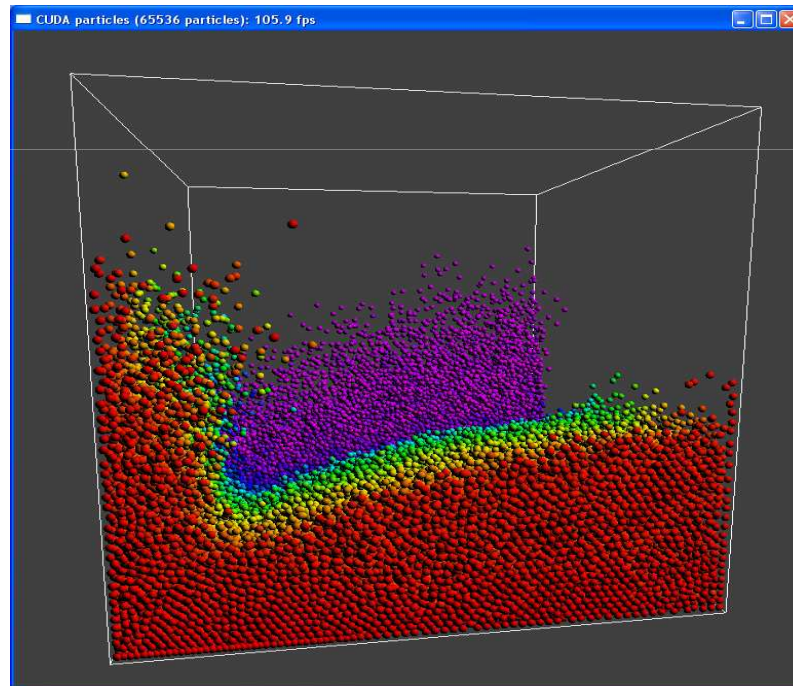
- Java,
- .NET,
- Perl,
- Python,
- Ruby,
- MatLab,
- Mathematica etc.

Therefore, you have options if you don't want to utilize C.

What types of Applications are there?

- Physical simulations (eg particle simulations)

example: <http://developer.download.nvidia.com/compute/cuda/sdk/website/C/src/particles/doc/particles.pdf>



- Medical imaging for CT, Ultrasound
- Gpus are good at implementing FFT in parallel allowing for very complicated

The screenshot shows a Mozilla Firefox browser window displaying the NVIDIA CUDA Zone website. The page is titled "CUDA for Medical" and features a navigation menu with options like "DOWNLOAD DRIVERS", "COOL STUFF", "SHOP", "PRODUCTS", "TECHNOLOGIES", "COMMUNITIES", and "SUPPORT". The main content area is titled "CUDA for Medical" and includes a sub-header "HIGH-SPEED, DETAILED ULTRASOUNDS" with the text "Ultrasound Imaging Company Develops Innovative Way to Quickly Deliver Highly-Detailed Scans". Below this, there is a section titled "Volumetric rendering from Techniscan Whole Breast Ultrasound system" and a paragraph explaining that the CUDA-based system is able to process Techniscan's algorithm twice as fast. The page also includes a search bar at the bottom and a footer with copyright information for 2011 NVIDIA Corporation.

CUA for Medical - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.nvidia.com/object/cuda_medical.html
SEARCH
Enter up to 3 stocks
GURU ANALYSIS
HamFair2010Report
Frequency Allocations
FEMA: Determine your Risk
Man and Machines | SPIKE
W: CUDA - Wikipedia, the free ency...
CUDA Zone
NVIDIA Parallel Nsight
CUDA for Medical

DOWNLOAD DRIVERS COOL STUFF SHOP PRODUCTS TECHNOLOGIES COMMUNITIES SUPPORT

CUDA ZONE WHAT'S NEW WHAT IS CUDA? CUDA GPU DEVELOPERS

NVIDIA Home > Technologies > CUDA Zone > CUDA in Action > CUDA for Medical

CUDA IN ACTION
CUDA for Research
CUDA for Medical
CUDA for Video and Photos
CUDA for Energy
CUDA for Finance

CUDA for Medical

HIGH-SPEED, DETAILED ULTRASOUNDS
Ultrasound Imaging Company Develops Innovative Way to Quickly Deliver Highly-Detailed Scans

The ability to quickly produce highly-detailed images in a short timeframe is particularly relevant in the field of breast cancer scanning. Techniscan, a developer of automated ultrasound imaging systems, ported its proprietary algorithm from a traditional CPU-based system to CUDA and NVIDIA Tesla GPUs.

Volumetric rendering from Techniscan Whole Breast Ultrasound system

The CUDA-based system is able to process Techniscan's algorithm twice as fast. Once this investigational device receives FDA clearance, patients will be able to have their results within a single visit.

For more information, please visit www.techniscanmedicalsyste.ms.com.

Click [here](#) to see the CUDA Community Showcase.
Click [here](#) to see more examples of CUDA in action.
Click [here](#) to see world map of educational institutions where CUDA and GPU Computing are taught.
Click [here](#) to see a list of GPU computing clusters built with CUDA-based Tesla systems.
Click [here](#) to see a list of organizations offering consulting and training services.

What's New | What is CUDA? | CUDA GPU | Developers
Copyright © 2011 NVIDIA Corporation | [Legal Info](#) | [Privacy Policy](#) | [RSS Feeds](#) | [NVIDIA Widgets](#)

Find: particle
Next Previous Highlight all Match case
Done

Other Uses of Cuda:


- Linear Algebra (CUBLAS)
- Matrix operations
- Monte Carlo Pricing
- Accelerated Encryption / Decryption
- Fourier Analysis FFTs
- Sorting
- Graphics Operations

Limitation of Cuda

- Bottleneck of copying data between cpu and gpu in terms of bandwidth.
(operations run very fast on gpu and sometimes a latency of things are processed more quickly than you can provide the gpu with data with which to work on.)
- Cuda enabled gpus only available from Nvidia.
- You can only de-reference GPU pointers in your cuda kernel and you CANNOT de-reference CPU pointers or get at any cpu memory location !!
- ** for GPUs supporting CUDA compute capability 1.3 and above, there are deviations from the IEEE 754 standard for rounding that one must consider in the results that are computed to assure they are getting accurate results.
- All threads execute the same program – its not like you can have some threads performing one task and others doing something totally different.
- Threads in a block can share memory and communicate with each other but threads in one block CANNOT communicate with threads in another block.
- GPU code is c code but no static variables, no recursion, no variable arguments.

10 / 333 | 120% | Find

Double Precision Floating Point

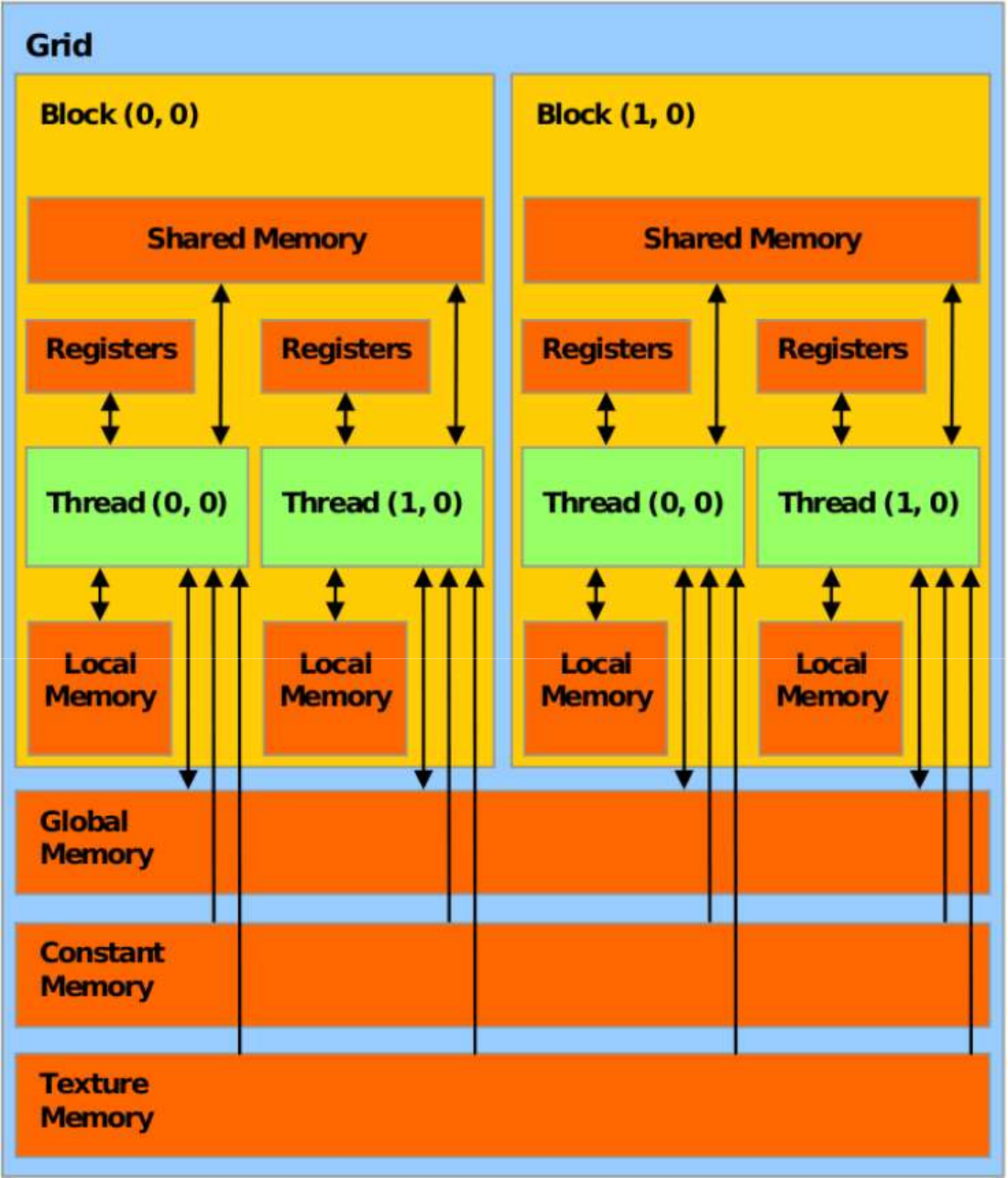


	NVIDIA GPU	SSE2	NVIDIA Cell SPE
Precision	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	All 4 IEEE, round to nearest, zero, inf, -inf	All 4 IEEE, round to nearest, zero, inf, -inf	Round to zero/truncate only
Denormal handling	Full speed	Supported, costs 1000's of cycles	Flush to zero
NaN support	Yes	Yes	No
Overflow and Infinity support	Yes	Yes	No infinity, clamps to max norm
Flags	No	Yes	Some
FMA	Yes	No	Yes
Square root	Software with low-latency FMA-based convergence	Hardware	Software only
Division	Software with low-latency FMA-based convergence	Hardware	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit
log ₂ (x) and 2 ^x estimates accuracy	23 bit	No	No

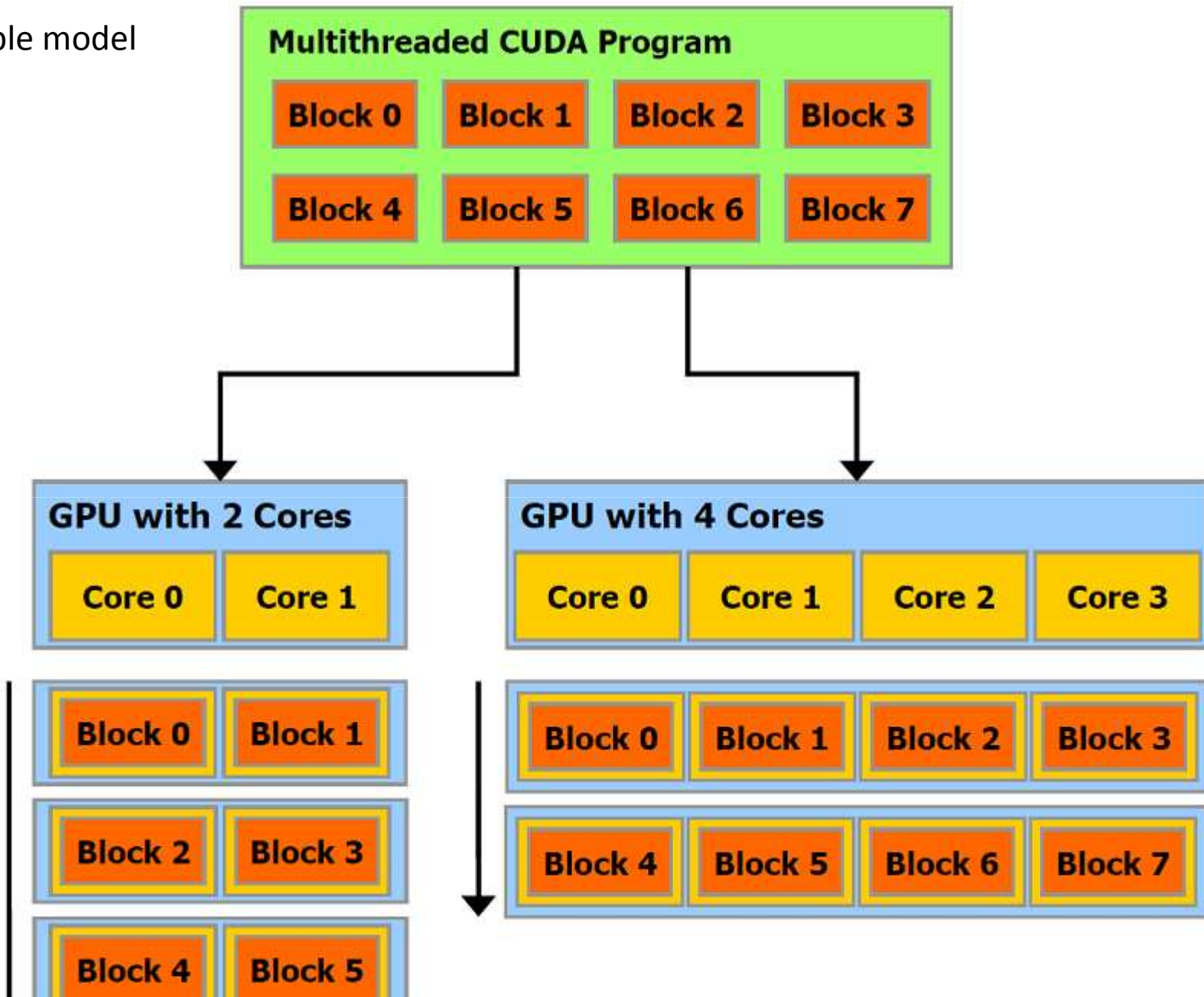
© NVIDIA Corporation 2008

Find: nvidia | Next | Previous | Highlight all | Match case

IEEE 754 are standards for floating point operations and rounding etc. It is important to know how the GPU compares and handles IEEE 754 for calculation intensive apps.



Scalable model



Compute Capabilities

- See this website for compute capabilities of your device:

http://www.nvidia.com/object/cuda_gpus.html

e.g. : [GeForce 9400GT](#) compute capability is: 1.0 number of multi-processors is 2 and number of cuda cores is: 16

Technical specifications	Compute capability (version)				
	1.0	1.1	1.2	1.3	2.x
Maximum x- or y- dimensions of a grid of thread blocks	65535				
Maximum number of threads per block	512			1024	
Maximum x- or y- dimension of a block	512			1024	
Maximum z- dimension of a block	64				
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24	32		48	
Maximum number of resident threads per multiprocessor	768	1024		1536	
Number of 32-bit registers per multiprocessor	8 K	16 K		32 K	
Maximum amount of shared memory per multiprocessor	16 KB			48 KB	
Number of shared memory banks	16			32	
Amount of local memory per thread	16 KB			512 KB	
Constant memory size	64 KB				
Cache working set per multiprocessor for constant memory	8 KB				
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				
Maximum width for 1D texture reference bound to a CUDA array	8192			32768	
Maximum width for 1D texture reference bound to linear memory	2^{27}				
Maximum width and height for 2D texture reference bound to linear memory or a CUDA array	65536 x 32768			65536 x 65535	
Maximum width, height and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 x 2048 x 2048				
Maximum number of textures that can be bound to a kernel	128				
Maximum width for a 1D surface reference bound to a CUDA array	Not supported			8192	
Maximum width and height for a 2D surface reference bound to a CUDA array				8192 x 8192	
Maximum number of surfaces that can be bound to a kernel				8	
Maximum number of instructions per kernel	2 million				

GPU Architecture

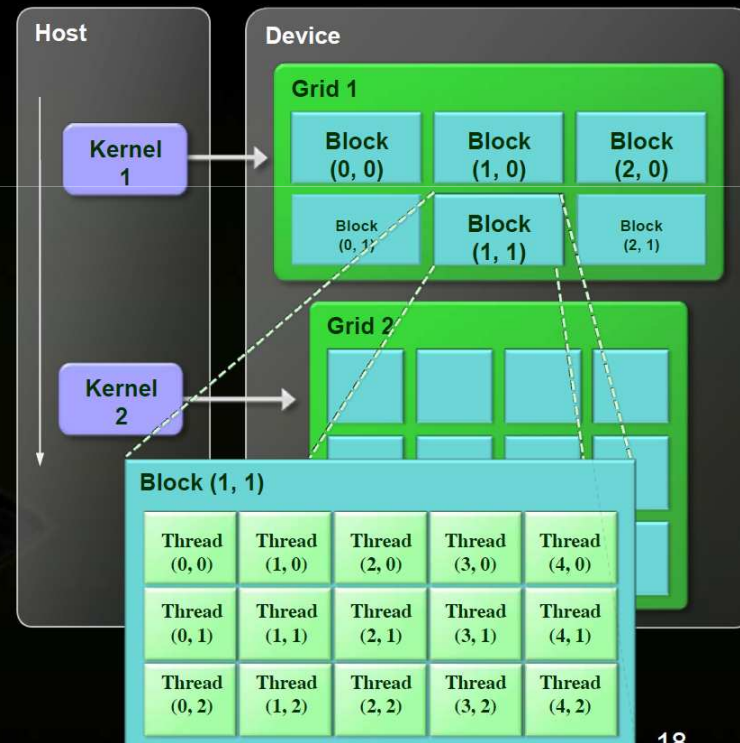
CUDA Programming Model



A kernel is executed by a **grid**, which contain **blocks**.

These blocks contain our **threads**.

- A **thread block** is a batch of threads that can cooperate:
 - Sharing data through shared memory
 - Synchronizing their execution
- Threads from different blocks operate independently



Getting Cuda on your Env

- To get the Cuda SDK, go to:
- http://developer.nvidia.com/object/cuda_3_2_downloads.html
- Download and install.

On Unix/Mac, you should have a directory structure as follows:

- /usr/local/cuda
- drwxr-xr-x 26 root wheel 884 Mar 27 11:19 doc
- drwxr-xr-x 43 root wheel 1462 Mar 27 11:19 include
- drwxr-xr-x 9 root wheel 306 Mar 27 11:19 lib
- drwxr-xr-x 10 root wheel 340 Mar 27 11:19 src
- drwxr-xr-x 11 root wheel 374 Mar 27 11:19 bin
- drwxr-xr-x 6 root wheel 204 Mar 27 11:19 computeprof
- drwxr-xr-x 4 root wheel 136 Oct 19 22:23 open64

Installing on your Environment cont

- In the bin dir, you should see these files including the nvidia c compiler: nvcc:

```
-rwxr-xr-x 1 root wheel  22132 Oct 19 22:23 cuda-memcheck
-rwxr-xr-x 1 root wheel 8295184 Oct 19 22:23 fatbin
-rw-r--r-- 1 root wheel   271 Oct 19 22:23 nvcc.profile
-rwxr-xr-x 1 root wheel  88352 Oct 19 22:23 bin2c
-rwxr-xr-x 1 root wheel 2888820 Oct 19 22:23 cudafe
-rwxr-xr-x 1 root wheel 2617072 Oct 19 22:23 cudafe++
-rwxr-xr-x 1 root wheel  84100 Oct 19 22:23 filehash
-rwxr-xr-x 1 root wheel 8331904 Oct 19 22:23 nvcc
-rwxr-xr-x 1 root wheel 8576388 Oct 19 22:23 ptxas
```

```
$ ./nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
```

```
Copyright (c) 2005-2010 NVIDIA Corporation
```

```
Built on Tue_Oct_19_17:52:08_PDT_2010
```

```
Cuda compilation tools, release 3.2, V0.2.1221
```

Compiling a sample program

- Firstly, I create a script to set my local environment

```
export DYLD_LIBRARY_PATH=/usr/local/cuda/lib:$DYLD_LIBRARY_PATH  
export PATH=$PATH:/usr/local/cuda/bin
```

This will ensure that the required libraries are in the library path
And that the cuda/bin dir is in your PATH so that you can get to the nvcc
compiler program.

That's IT, now lets compile and run some samples !!

Compiling your first program

- `nvcc vector_addition.cu -o vector_add`
- `vector_addition.cu` is the name of your cuda program
- `-o` is the output file name that is the executable file that is produced.

Analysis of a Cuda Program

- There are two essential parts of a cuda c program:
 1. The CPU part – contains normal c program functions
 2. The GPU part – contains your device Kernel
IE the part that runs on the GPU device.

Device Function

```
__global__ void vector_add(const float *a,  
                           const float *b,  
                           float *c,  
                           const size_t n)  
{  
    // compute the global element index this thread should process  
    unsigned int i = threadIdx.x + blockDim.x * blockIdx.x;  
  
    // avoid accessing out of bounds elements  
  
    if(i < n)  
    {  
        // sum elements  
        c[i] = a[i] + b[i];  
    }  
}
```

See some interesting things?

```
unsigned int i = threadIdx.x + blockDim.x * blockIdx.x;
```

```
// avoid accessing out of bounds elements  
if(i < n)  
{  
    ... <<your code here >>  
}
```

Remember that we are partitioning our code to run on a gpu device

That has many blocks and threads within each block. The above assignment of integer i assures that we have a unique index amongst all the blocks/threads that have been allocated to our program.

`threadIdx` — index of current thread; the thread index is between 0 and `blockDim - 1`

`blockIdx` — the index of current block; the block index is between 0 and `gridDim - 1`

`blockDim` — the block size dimensions

```

int main(void)
{
// create arrays of 5K elements
const int num_elements = 5000;

// compute the size of the arrays in bytes
const int num_bytes = num_elements * sizeof(float);

// points to host & device arrays
float *device_array_a = 0;
float *device_array_b = 0;
float *device_array_c = 0;
float *host_array_a = 0;
float *host_array_b = 0;
float *host_array_c = 0;

// malloc the host arrays
host_array_a = (float*)malloc(num_bytes);
host_array_b = (float*)malloc(num_bytes);
host_array_c = (float*)malloc(num_bytes);

// cudaMalloc the device arrays
cudaMalloc((void*)&device_array_a, num_bytes);
cudaMalloc((void*)&device_array_b, num_bytes);
cudaMalloc((void*)&device_array_c, num_bytes);

// if any memory allocation failed, report an error message
if(host_array_a == 0 || host_array_b == 0 || host_array_c == 0 ||
   device_array_a == 0 || device_array_b == 0 || device_array_c == 0)
{
printf("couldn't allocate memory\n");
return 1;
}

// initialize host_array_a & host_array_b
for(int i = 0; i < num_elements; ++i)
{
// make array a a linear ramp
host_array_a[i] = (float)i;

// make array b random
host_array_b[i] = (float)rand() / RAND_MAX;
}

// copy arrays a & b to the device memory space
cudaMemcpy(device_array_a, host_array_a, num_bytes, cudaMemcpyHostToDevice);
cudaMemcpy(device_array_b, host_array_b, num_bytes, cudaMemcpyHostToDevice);

```

```
// compute c = a + b on the device
const size_t threads_per_block = 256;
size_t block_size = num_elements / block_size;

// launch the kernel
// IMPORTANT NOTE – THE CALL TO THE KERNEL IS ASYNCHRONOUS – I.E. AFTER THIS CALL IS DONE, CONTROL
// RETURNS IMMEDIATELY TO THE CPU
vector_add<<<block_size, threads_per_block>>>(device_array_a, device_array_b, device_array_c, num_elements);

// copy the result back to the host memory space
cudaMemcpy(host_array_c, device_array_c, num_bytes, cudaMemcpyDeviceToHost);

// IMPORTANT NOTE: cudaMemcpy is SYNCHRONOUS, copy start ONLY after all previous cuda kernel calls are
// complete.
// print out the first 10 results
for(int i = 0; i < 10; ++i)
{
    printf("result %d: %1.1f + %7.1f = %7.1f\n", i, host_array_a[i], host_array_b[i], host_array_c[i]);
}

// deallocate memory
free(host_array_a);
free(host_array_b);
free(host_array_c);

cudaFree(device_array_a);
cudaFree(device_array_b);
cudaFree(device_array_c);
}
```

Elements in Detail

- First, we need to allocate memory on the GPU Device via not a cpu Malloc but a GPU Malloc:
- `cudaMalloc((void**)&device_array_a, num_bytes);`
- This will allocate `num_bytes` number of bytes of type float
- Next, we need to copy data to the GPU and again there is a cuda function that facilitates this copy:
- `cudaMemcpy(device_array_a, host_array_a, num_bytes, cudaMemcpyHostToDevice);`

- Next, we need to invoke our GPU function:

```
// launch the kernel
```

```
vector_add<<<block_size,threads_per_block>>>(device_array_a, device_array_b, device_array_c,  
    num_elements);
```

- Vector_add is the name of the gpu device function. It takes four parameters which are 3 arrays and the number of elements
- Block size is the number of blocks to utilize
- Thread Size is the number of threads per block
- Therefore, in our example we set our threads to be 256
- And blocks to be 5,000 / 256
- $5000 / 256 * 256 = 5,000$ elements to process.

Analysis and Demo

- I will now show some live examples on my mac using the command line and nvcc and using eclipse for c.

Mathematica examples

- Lets now look at some examples of calling cuda from Mathematica.
- First we need to import the Cuda package to give us the ability to do cuda:
`Needs["CUDALink`"]`
- If You need to install cuda, issue this command and it will retrieve the latest version via the internet:
- `CUDAResourcesInstall[]`

Cuda on Mathematica

- To ensure that your environment is ready to go issue this command:
- `CUDAQ[]` - this will return `True` if cuda is enabled and ready or `false` otherwise.
- To see the compute capability that your card has, issue this command:

```
TableView[Table[ CUDAInformation[ii, "Name"] -> CUDAInformation[ii, "Compute Capabilities"], {ii, $CUDADeviceCount}]]
```

Ok, lets build a device function to and invoke it from
Mathematica

– First lets define a couple of data sets
in Mathematica to hold some numbers:

```
ds1={1,3,5,7,9}
```

```
ds2={2,4,6,8,10}
```

Lets add these lists together using cuda and get the
result list back from the gpu

- So, lets define our kernel function:

```
code = " __global__ void vectAdd (int * list1, int *list2, int * out, int length)
{
    int index = threadIdx.x + blockIdx.x*blockDim.x;

    if (index < length) out[index] = list1[index] + list2[index];

}";
```

Now, lets define a wrapper to that function in Mathematica and load the function:

```
cudaFun = CUDAFunLoad[code, "vectAdd", {[_Integer, _, "Input"], [_Integer, _, "Input"], [_Integer, _, "Output"], _Integer}, 5 ]
```

Define a variable to hold the size of our list in mathematica:

```
listSize = 10;
```

- Now we can run our newly defined cuda function and get results back
- `res = cudaFun[ds1, ds2, ds3, listSize];`
- `Take[First@res,5]`

Other useful built in Mathematica Cuda things:

- Multiply two random integer matrices:
 - `CUDADot[RandomInteger[1, {5, 5}], RandomInteger[1, {5, 5}]]; MatrixForm[%]`
- Sort an List of Integers:
 - `CUDASort[Reverse@Range[100]]`

• Image Processing

- The *CUDALink* Image Processing module can be classified into three categories. The first is convolution, which is optimized for CUDA. The second is morphology, which contains abilities such as erosion, dilation, opening, and closing. Finally, there are the binary operators. These are the image multiplication, division, subtraction, and addition operators. All operations work on either images or lists.
- [CUDAImageConvolve](#) convolve the kernel with the specified kernel.
- [CUDABoxFilter](#) convolve the kernel with the [BoxMatrix](#) kernel
- [CUDAErosion](#) perform morphological erosion
- [CUDADilation](#) perform morphological dilation
- [CUDAOpening](#) perform morphological opening
- [CUDAClosing](#) perform morphological closing
- [CUDAClamp](#) clamp the values between a range
- [CUDAColorNegate](#) invert the values of input
- [CUDAImageAdd](#) add two inputs
- [CUDAImageSubtract](#) subtract two inputs
- [CUDAImageMultiply](#) multiply two inputs
- [CUDAImageDivide](#) divide two inputs

• **Linear Algebra and Fourier Transforms**

- [CUDADot](#) give product of vectors and matrices
- [CUDATranspose](#) tranpose input matrix
- [CUDAArgMaxList](#) give the index with maximum absolute element
- [CUDAArgMinList](#) give the index with minimum absolute element
- [CUDAFourier](#) find the Fourier transform
- [CUDAInverseFourier](#) find the inverse Fourier transform

Summary

- Cuda is ubiquitous – if you have an nvidia graphics card in your computer, you probably have the ability to do cuda. It is good at computations that can be expressed as data parralel computations. IE the same program executed on many elements in parralel.
- Cuda allows massively parralel apps for the masses.
- Even if you don't know C, there are wrappers for Cuda available in many other programming languages and in high level tools like Mathematica so it is a very open technology.
- **But...**
 - Always verify calculations to make sure you get the correct results and precision that you are looking for.
 - Always know the limitations and bottlenecks like the time it takes to copy memory from cpu to gpu. All depending upon what you are doing, that can be a show stopper right there.