An Easy Introduction to CUDA C and C++

Posted on **October 31, 2012 (https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/)** by **Mark Harris (https://devblogs.nvidia.com/parallelforall/author/mharris/)** | **47 Comments (https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/#disqus_thread)**
Tagged **CUDA C/C++ (https://devblogs.nvidia.com/parallelforall/tag/cuda-cc/)**

Update (January 2017): Check out a new, even easier introduction to CUDA (https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/)!

This post is the first in a series on CUDA C and C++, which is the C/C++ interface to the CUDA parallel computing platform. This series of posts assumes familiarity with programming in C. We will be running a parallel series of posts about CUDA Fortran targeted at Fortran programmers . These two series will cover the basic concepts of parallel computing on the CUDA platform. From here on unless I state otherwise, I will use the term “CUDA C” as shorthand for “CUDA C and C++”. CUDA C is essentially C/C++ with a few extensions that allow one to execute functions on the GPU using many threads in parallel.

CUDA Programming Model Basics

Before we jump into CUDA C code, those new to CUDA will benefit from a basic description of the CUDA programming model and some of the terminology used.

The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the *host* refers to the CPU and its memory, while the *device* refers to the GPU and its memory. Code run on the host can manage memory on both the host and device, and also launches *kernels* which are functions executed on the device. These kernels are executed by many GPU threads in parallel.

Given the heterogeneous nature of the CUDA programming model, a typical sequence of operations for a CUDA C program is:

1. Declare and allocate host and device memory.
2. Initialize host data.
3. Transfer data from the host to the device.
4. Execute one or more kernels.
5. Transfer results from the device to the host.

Keeping this sequence of operations in mind, let’s look at a CUDA C example.

A First CUDA C Program

In a recent post, I illustrated Six Ways to SAXPY (http://devblogs.nvidia.com/parallelforall/six-ways-saxpy/), which includes a CUDA C version. SAXPY stands for “Single-precision A*X Plus Y”, and is a good “hello world” example for parallel computation. In this post I will dissect a more complete version of the CUDA C SAXPY, explaining in detail what is done and why. The complete SAXPY code is:

```
#include <stdio.h>

__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 main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
```

The function `saxpy` is the kernel that runs in parallel on the GPU, and the `main` function is the host code. Let's begin our discussion of this program with the host code.

Host Code

The main function declares two pairs of arrays.

```
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));
```

The pointers `x` and `y` point to the *host* arrays, allocated with `malloc` in the typical fashion, and the `d_x` and `d_y` arrays point to *device* arrays allocated with the `cudaMalloc` function from the CUDA runtime API. The host and device in CUDA have separate memory spaces, both of which can be managed from host code (CUDA C kernels can also allocate device memory on devices that support it).

The host code then initializes the host arrays. Here we set `x` to an array of ones, and `y` to an array of twos.

```
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
```

To initialize the device arrays, we simply copy the data from `x` and `y` to the corresponding device arrays `d_x` and `d_y` using `cudaMemcpy`, which works just like the standard C `memcpy` function, except that it takes a fourth argument which specifies the direction of the copy. In this case we use `cudaMemcpyHostToDevice` to specify that the first (destination) argument is a device pointer and the second (source) argument is a host pointer.

```
  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
```

After running the kernel, to get the results back to the host, we copy from the device array pointed to by `d_y` to the host array pointed to by `y` by using `cudaMemcpy` with `cudaMemcpyDeviceToHost`.

```
  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

LAUNCHING A KERNEL

The `saxpy` kernel is launched by the statement:

```
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
```

The information between the triple chevrons is the *execution configuration*, which dictates how many device threads execute the kernel in parallel. In CUDA there is a hierarchy of threads in software which mimics how thread processors are grouped on the GPU. In the CUDA programming model we speak of launching a kernel with a *grid* of *thread blocks*. The first argument in the execution configuration specifies the number of thread blocks in the grid, and the second specifies the number of threads in a thread block.

Thread blocks and grids can be made one-, two- or three-dimensional by passing dim3 (a simple struct defined by CUDA with `x`, `y`, and `z` members) values for these arguments, but for this simple example we only need one dimension so we pass integers instead. In this case we launch the kernel with thread blocks containing 256 threads, and use integer arithmetic to determine the number of thread blocks required to process all `N` elements of the arrays (`(N+255)/256`).

For cases where the number of elements in the arrays is not evenly divisible by the thread block size, the kernel code must check for out-of-bounds memory accesses.

CLEANING UP

After we are finished, we should free any allocated memory. For device memory allocated with `cudaMalloc()`, simply call `cudaFree()`. For host memory, use `free()` as usual.

```
cudaFree(d_x);
  cudaFree(d_y);
  free(x);
  free(y);
```

Device Code

We now move on to the kernel code.

```
__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];
}
```

In CUDA, we define kernels such as `saxpy` using the `__global__` declaration specifier. Variables defined within device code do not need to be specified as device variables because they are assumed to reside on the device. In this case the `n`, `a` and `i` variables will be stored by each thread in a register, and the pointers `x` and `y` must be pointers to the device memory address space. This is indeed true because we passed `d_x` and `d_y` to the kernel when we launched it from the host code. The first two arguments, `n` and `a`, however, were not explicitly transferred to the device in host code. Because function arguments are passed by value by default in C/C++, the CUDA runtime can automatically handle the transfer of these values to the device. This feature of the CUDA Runtime API makes launching kernels on the GPU very natural and easy—it is almost the same as calling a C function.

There are only two lines in our `saxpy` kernel. As mentioned earlier, the kernel is executed by multiple threads in parallel. If we want each thread to process an element of the resultant array, then we need a means of distinguishing and identifying each thread. CUDA defines the variables `blockDim`, `blockIdx`, and `threadIdx`. These predefined variables are of type `dim3`, analogous to the execution configuration parameters in host code. The predefined variable `blockDim` contains the dimensions of each thread block as specified in the second execution configuration parameter for the kernel launch. The predefined variables `threadIdx` and `blockIdx` contain the index of the thread within its thread block and the thread block within the grid, respectively. The expression:

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

generates a global index that is used to access elements of the arrays. We didn't use it in this example, but there is also `gridDim` which contains the dimensions of the grid as specified in the first execution configuration parameter to the launch.

Before this index is used to access array elements, its value is checked against the number of elements, `n`, to ensure there are no out-of-bounds memory accesses. This check is required for cases where the number of elements in an array is not evenly divisible by the thread block size, and as a result the number of threads launched by the kernel is larger than the array size. The second line of the kernel performs the element-wise work of the SAXPY, and other than the bounds check, it is identical to the inner loop of a host implementation of SAXPY.

```
if (i < n) y[i] = a*x[i] + y[i];
```

Compiling and Running the Code

The CUDA C compiler, nvcc, is part of the NVIDIA CUDA Toolkit (http://www.nvidia.com/content/cuda/cuda-toolkit.html). To compile our SAXPY example, we save the code in a file with a .cu extension, say `saxpy.cu`. We can then compile it with nvcc.

```
nvcc -o saxpy saxpy.cu
```

We can then run the code:

```
% ./saxpy
Max error: 0.000000
```

Summary and Conclusions

With this walkthrough of a simple CUDA C implementation of SAXPY, you now know the basics of programming CUDA C. There are only a few extensions to C required to "port" a C code to CUDA C: the `__global__` declaration specifier for device kernel functions; the execution configuration used when launching a kernel; and the built-in device variables `blockDim`, `blockIdx`, and `threadIdx` used to identify and differentiate GPU threads that execute the kernel in parallel.

One advantage of the heterogeneous CUDA programming model is that porting an existing code from C to CUDA C can be done incrementally, one kernel at a time.

In the next post of this series, we will look at some performance measurements and metrics.

Note: this post is based on the post "An Easy Introduction to CUDA Fortran (http://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-fortran/)" by Gregory Reutsch (http://devblogs.nvidia.com/parallelforall/author/gruetsch/).

RELATED POSTS

A CUDA Dynamic Parallelism Case Study: PANDA (https://devblogs.nvidia.com/parallelforall/a-cuda-dynamic-parallelism-case-study-panda/)

CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics (https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/)

Accelerating Bioinformatics with NVBIO (https://devblogs.nvidia.com/parallelforall/accelerating-bioinformatics-nvbio/)

GPU Pro Tip: Fast Dynamic Indexing of Private Arrays in CUDA (https://devblogs.nvidia.com/parallelforall/fast-dynamic-indexing-private-arrays-cuda/)

About Mark Harris

Mark is Chief Technologist for GPU Computing Software at NVIDIA. Mark has fifteen years of experience developing software for GPUs, ranging from graphics and games, to physically-based simulation, to parallel algorithms and high-performance computing. Mark has been using GPUs for general-purpose computing since before they even supported floating point arithmetic. While a Ph.D. student at UNC he recognized this nascent trend and coined a name for it: GPGPU (General-Purpose computing on Graphics Processing Units), and started GPGPU.org to provide a forum for those working in the field to share and discuss their work.

Follow @harrism on Twitter (https://twitter.com/intent/user?screen_name=harrism)
View all posts by Mark Harris → (https://devblogs.nvidia.com/parallelforall/author/mharris/)

Muhammad Idil Haq Amir • 4 months ago

Hello! Nice tutorial. I want to build a web page that can utilize Cuda in optimizing the calculation and maybe simulation. How to deal with it? Javascript+Html+C+Cuda? I have no idea even whether it is possible or not. Please give me some references.

Reply • Share ›

Ab Er (Nudelimperium) ↗ Muhammad Idil Haq Amir • 3 months ago

As long as I understand your question, this won't be possible as you describe it. This is easily deducible from how websites work.

Javascript/ Html gets rendered on the client side (on the computer that opens the website). Since you can't know the client's architecture and wether she even uses a Nvidia GPU, it is impossible to supply the correct binaries for each and every possible system.

Server-side however is easy (speaking of php/ aspx or web services/ api calls). Just develop your Cuda Code in C/ C++, provide a web-api to make callbacks to via javascript (for example https://github.com/Microsof...) and run it on your server.

Reply • Share ›

Barsik_The_CaT ↗ Muhammad Idil Haq Amir • 4 months ago

You need to keep in mind that CUDA is meant to work with GPU a server is unlikely to have.
But, if you server has a GPU, you could make a basic webpage with input form, get all that data, and then run your CUDA program on a server using PHP program exec().

Reply • Share ›

Kshitij Shah • 4 months ago

I just stated CUDA programming today. After basics, I was playing with around with this program (and a similar one I wrote). I ran the program on larger numbers. It worked perfectly well up to 2^23. But with 2^24 it got all the answers wrong (I changed it count number of wrong answers). I noticed that with (2^24+255)/256 was going out of range for unsigned 16 bit integer. I changed number of threads to 512 and it gave correct answers. I tried 2^25 and 1024, still correct. But 2^26 and 2048 gave all wrong answers. I tried 4096 still all wrong. So are there some kind of limits to the number of blocks an threads ? Or it has to do something with memory ? I am running on 8 gb GTX 1070.

Reply • Share ›

Mark Harris Mod ↗ Kshitij Shah • 4 months ago

Highly recommend you jump to this new introductory post I just published last week: https://devblogs.nvidia.com...

Yes, there are limit: as you guessed the maximum grid size is 65535 in each dimension. You can query these limits using the device properties API. If you write your kernels using grid-stride loops as in the example above you can overcome this issue with only a change to your launch to clamp the grid size to maximum 65535: e.g. min(65535, (N+255)/256).

Maximum block size is 1024 threads, I believe

Maximum block size is 1024 threads, I believe.

Reply • Share ›

Kshitij Shah ➚ Mark Harris • 4 months ago

Thanks, Mark. I will surely check the new post. Block size is indeed 1024 thread for modern GPUs, I checked.

Reply • Share ›

vasu gupta • 5 months ago

I am a complete beginner in cuda and have somewhat knowledge of c++, how can i get started on cuda

Reply • Share ›

Michael • 5 months ago

I am a CUDA beginner... Thanks for the great tutorial, helped me a lot in getting started!

My question concerns the execution configuration: Out of curiosity, I am also outputting the blockIdx, blockDim and threadIdx for every thread of the saxpy kernel (added one line to void saxpy: printf("Block idx.x, dim.x, threadIdx.x: %i %i %i\n", blockIdx.x, blockDim.x,threadIdx.x);)

Now I created this output 3 times with different execution configurations:
1) above original: I get a list of 4096 rows, the second column for all rows is 256. The sum is 1,048,576 (== N, as expected).
2) configuration: saxpy<<<(N+511)/512, 512>>>(N, 2.0f, d_x, d_y); I still get 4096 rows, but this time the second column is always 512. The total number is 2,097,152.
3) configuration: saxpy<<<(N+127)/128, 128>>>(N, 2.0f,
d_x, d_y); I still get 4096 rows, but this time the second column is
always 128. The total number is 524,288.

I don't understand... Why is the total number of threats always 4096, but the total product of dimension*threads is not preserving N in all three cases?

Also, in the follow-up post (on measuring), the integer division of the execution configuration is changed from /256 to /512, but the comment line still reads "SAXPY on 1M elements". What am I missing?

Reply • Share ›

Mark Harris Mod ➚ Michael • 5 months ago

I think if you look at your code carefully you'll discover that each example is actually trying to print N (= 1M) lines. But you are running up against the printf FIFO default size of 1 MB and getting many fewer than that printed. If you call cudaDeviceSetLimit(cudaLimitPrintfFifoSize, X); for some large value of X you'll get more, but you may instead want to limit it to only print the first thread of every block instead ("if (threadIdx.x == 0)").

Reply • Share ›

Michael ➚ Mark Harris • 5 months ago

That explains everything... Thank you very much!

Reply • Share ›

Jacek Łysiak • 5 months ago

What about freeing allocated memory?
I'm beginner in CUDA C, but I think you should free requested memory, formally.

Reply • Share ›

Mark Harris Mod ➚ Jacek Łysiak • 5 months ago

Hi Jacek, great point. I corrected this omission in the post.

Reply • Share ›

PatMcC • 6 months ago

Hi, I've run both the .cu code and .cuf code for this example. The .cu code runs as is and gives the proper result, however the .cuf code returns 2.00000. I'm new to CUDA and am wondering if you have any idea why the results are different? The machine I'm using does have 2 gpus installed.

Reply • Share ›

PatMcC ➚ PatMcC • 6 months ago

From pgforums... Pascal GPUs need to explicitly generate binaries from cuda-8.0... See link for solution, if interested.

https://www.pgroup.com/user...

Reply • Share ›

MD • 7 months ago

Hi , I'm new in this side and I have final project and I need to use CUDA to handle Big data .
My question how I can read file in CUDA C++?

Reply • Share ›

Mark Harris Mod ➚ MD • 7 months ago

The host code (which does the file loading) is just regular C/C++. So load files just like you normally would.

Reply • Share ›

Shrikanth Yadav • 8 months ago

Thank you for the post. I am new to CUDA and would like to clarify some errors I came across. Running the 'nvcc -o saxpy saxpy.cu' on my command promt gives me 'Cannot find compliler 'cl.exe' in Path '. Also I have the following errors on the sample code I plan to run as seen in the screenshot. Does this indicate a mistake in the compiler installation?
I am using Visual Studio 2015 and have a Nvidia geforce 820m.

Thank you
Shrikanth Yadav

Reply • Share ›

Mark Harris Mod ➚ Shrikanth Yadav • 7 months ago

Do you have CUDA 8 installed? Previous versions did not support Visual Studio 2015. I can't see the errors in your screenshot.

Reply • Share ›

Shrikanth Yadav ➚ Mark Harris • 7 months ago

Hi
I was able to solve the prolem after reinstalling both VS15 and CUDA. The compiling issue is also solved. Thank you

Reply • Share ›

David Phillips • a year ago

Just a small edit, there's a missing '\' before the 'n' in the printf statement. This post has been a very useful, simple and concise starting point for getting into CUDA. Thanks!

Reply • Share ›

Missed Connection • a year ago

There's a missing > in a < code > HTML tag, just search for "coden" and it should be the only instance on this page (OK, other than mine!)

Reply • Share ›

Mark Harris Mod ➚ Missed Connection • a year ago

Fixed -- thanks!

Reply • Share ›

Reid • 2 years ago

I know this is old, but in your kernel call, why do you pass "2.0f"? Is that because there are two floating point operations taking place in your kernel?

Reply • Share ›

Bram ➚ Reid • 2 years ago

Is just some arbitrary number for 'A' in the 'A*X + Y' expression.

Reply • Share ›

Reid ➚ Bram • 2 years ago

That what I figured after looking at it a bit closer was that it was just a generic constant.

Thanks!

Reply • Share ›

Jae • 2 years ago

Thanks for your posting. I'm newbie for CUDA. When I compile your code, i got the same result "Max error: 0.00..0n". However, when I debug it, it doesn't get in __global__ void saxpy function. Is it normal?

Reply • Share ›

Jae ➚ Jae • 2 years ago

sorry, one more!
How can "int N = 1<<20" work?
i have no idea.

Reply • Share ›

Mark Harris Mod ➚ Jae • 2 years ago

1<<20 shifts the value 1 left 20 bits. This is equivalent to 2 raised to the power of 20 == 1048576.

Reply • Share ›

Jae ➚ Mark Harris • 2 years ago

Thanks Mark!!, then N should be 1048576/32 = 32768 right?

And i'm using Nsight but

__global__

void saxpy(int n, float a, float *x, float *y)

{ <---- here

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

if (i < n) y[i] = a*x[i] + y[i];

}

in this function, i cannot get under the first bracket, which i pointed. just empty red circle

Reply • Share ›

Mark Harris Mod ➚ Jae • 2 years ago

I suspect you are not setting a GPU breakpoint. If you are stepping on the host, it will not automatically step into GPU code because different threads are running it, so you need to set a breakpoint in the GPU function. Please check the debugger docs / tutorials.

Reply • Share ›

Jae ➚ Mark Harris • 2 years ago

Yeab, you are right. It's been just two days I studied this. Thanks for your reply!!

Reply • Share ›

Reply • Share ›

Jae Jae • 2 years ago

is there anything more header file than stdio.h?

Reply • Share ›

Mark Harris Mod Jae • 2 years ago

Not sure what you are asking. That's the only header needed by this example.

Reply • Share ›

Jae Jae • 2 years ago

I mean it gets in saxpy but it doesn't read

int i = blockIdx.x*blockDim.x + threadIdx.x;
if (i < n) y[i] = a*x[i] + y[i];

Reply • Share ›

Mark Harris Mod Jae • 2 years ago

What debugger are you using? You need to use a gpu-aware debugger such as cuda-gdb or NSight (Visual Studio Edition or Eclipse Edition).

Reply • Share ›

Huioon Kim • 2 years ago

There are missing '&'s in front of 'd_x' and 'd_y' when calling cudaMalloc in the first code snippets of the Host Code section.

Reply • Share ›

Mark Harris Mod Huioon Kim • 2 years ago

Fixed. Thanks!

Reply • Share ›

rogers • 3 years ago

I got the error: #include expects filename, what is the filename?

Reply • Share ›

Mark Harris Mod rogers • 3 years ago

Sorry about that, the code got mangled by the site. I've fixed it. (The filename is <stdio.h>).

Reply • Share ›

dylanholmes206 • 3 years ago

There are typos in the first paragraphs of both the Device Code and Host Code sections, x_d and y_d should be d_x and d_y.

Reply • Share ›

Mark Harris Mod dylanholmes206 • 3 years ago

I've fixed these. Thank you!

Reply • Share ›

Charles Turner • 3 years ago

Hi, thanks for this tutorial!

I have a GeForce 210 card, and when I run this program, I get

Max error: 2.000000

Whereas you see a max error of zero. It seems like the y[i] array is not getting operated on with my computer setup, but I get no compiler errors with nvcc. When I print out the result of max(maxError, abs(y[i]-4.0f)), I see the value 2.000000 every time, indicating that nothing happened to the y array on the device.

Do you have any advice on what might be going wrong here?

Thanks,
Charles.

Reply • Share ›

Mark Harris Mod ➚ Charles Turner • 3 years ago

I suspect a CUDA error. Unfortunately the code example doesn't check for errors, because that is the lesson of the follow up post. Can you add error checking as shown in the post http://devblogs.nvidia.com/... and see what errors you see?

Reply • Share ›

Charles Turner ➚ Mark Harris • 3 years ago

Ah, I did have some errors. I was using Ubuntu 13.10 and tried some "workarounds" involving third-party PPAs since this isn't an officially supported platform. I couldn't get the workarounds to work it seems, but instead of tracking down the problem, I installed 12.04 LTS and followed the official instructions, now everything seems to be working and I get the "Max error: 0.00000" output now. Yay!

Thank you for taking the time to help me, Mark.

Reply • Share ›

Mark Harris Mod ➚ Charles Turner • 3 years ago

My pleasure, glad you got it working.

Reply • Share ›

disqus_P9AzcUzcTJ • 3 years ago

I think there's an error in the line "In this case we use cudaMemcpyHostToDevice to specify that the first argument is a host pointer and the second argument is a device pointer.". Shouldn't it be "In this case we use cudaMemcpyHostToDevice to specify that the first argument is a device pointer and the second argument is a host pointer."? I think you exchanged host pointer with device pointer.

Reply • Share ›

Mark Harris Mod ➚ disqus_P9AzcUzcTJ • 3 years ago

Thanks Nisarg, good catch. I've fixed this error.

Reply • Share ›

ACCELERATED COMPUTING (HTTPS://DEVELOPER.NVIDIA.COM/ACCELERATED-COMPUTING)

GAMEWORKS (HTTPS://DEVELOPER.NVIDIA.COM/GAMEWORKS)

EMBEDDED COMPUTING (HTTPS://DEVELOPER.NVIDIA.COM/EMBEDDED-COMPUTING)

DESIGNWORKS (HTTPS://DEVELOPER.NVIDIA.COM/DESIGNWORKS)

GET STARTED

About CUDA (https://developer.nvidia.com/about-cuda)

Parallel Computing (https://developer.nvidia.com/accelerated-computing-training)

CUDA Toolkit (https://developer.nvidia.com/cuda-toolkit)

CUDACast (http://www.youtube.com/playlist?list=PL5B692fm6--vScfBaxgY89IRWFzDt0Khm)

LEARN MORE

Training and Courseware (https://developer.nvidia.com/cuda-education-training)

Tools and Ecosystem (https://developer.nvidia.com/tools-ecosystem)

Academic Collaboration (https://developer.nvidia.com/academia)

Documentation (http://docs.nvidia.com/cuda/index.html)

GET INVOLVED

Forums (https://devtalk.nvidia.com/)

Parallel Forall Blog (https://devblogs.nvidia.com/parallelforall/)