Take CUDA for a Test Drive

John Burkardt: burkardt@vt.edu
Advanced Research Computing
Virginia Tech
day? April 2017

Slides available at:
https://secure.hosting.vt.edu/www.arc.vt.edu/wp-content/uploads/2017/04/cuda_test_drive.pdf

Take a Test Drive on Our CUDA Cluster!

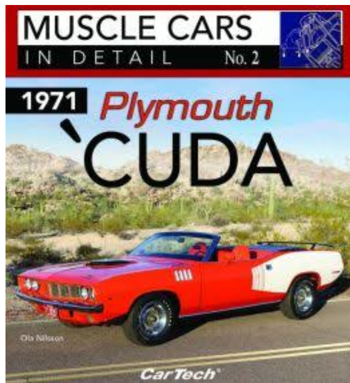


Figure: CUDA models 1971 and 2014

Coverage

Take a test drive on ARC's shiny new computer cluster NewRiver.

I'll show you how to understand this computer cluster as a connected stack of PC's, each with a powerful processor and an interesting object called a graphics processing unit (GPU).

We'll see how GPU's evolved from managing PC displays to high end game consoles by figuring out how to do parallel programming.

I'll show how GPU's have become so amazingly fast at some kinds of work that they can be added on to a standard CPU as a kind of supplementary computational toaster.

In order to allow programmers to "drive" this new hybrid monster of CPU + GPU, the C programming language was augmented with CUDA.

We'll get an overview of how CUDA can orchestrate computations on a CPU/GPU system, moving data to the GPU for fast processing, and pulling results back for analysis.

The History of GPU's

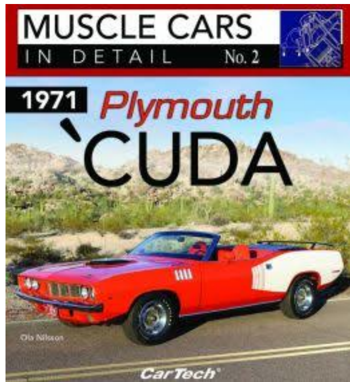
The display on a computer represents a significant chunk of memory that has to be updated over and over again.

Rather than waste the CPU's time and memory, the task of controlling the display was moved to a display driver, which was given separate memory (the size of the current screen + the "next" screen) and a stupid processor that dealt with the simple tasks associated with a black and white screen, then thousands, and millions of colors.

Computer games, originally running on PC's, moved to game machines that could satisfy the specialized needs of games. The computer display unit was soon upgraded to handle intense graphics processing (3D modeling, textures, shading).

An effective procedure was to essentially compute the next value of every pixel at the same time, in parallel. This required the development of hardware and software to make this programming possible.

HELLO Test Drive: Does It Start?



Hello, World!

Before we do any serious work, let's just make sure we know the most basic facts about what a CUDA program looks like and how to get it started.

By tradition, the training vehicle we start out with is a "Hello, world!" program.

Since CUDA is closely related to C, we'll try two models of the program, starting with a C code and then moving to a CUDA code.

We're not ready to talk about performance yet, but just about the similarities and differences between C and CUDA.

Compile and Run a C program

Looking ahead to CUDA, we encapsulate the repetitive work into a separate function, which we call the kernel. To make things happen:

- We write a C program with extension `.c`,
- We create an executable program **hello** with a compiler like `gcc`,
- We invoke the program by name.

```
gcc -o hello hello.c
./hello
```

```
Hello from C loop index 0
Hello from C loop index 1
Hello from C loop index 2
Hello from C loop index 3
Hello from C loop index 4
Hello from C loop index 5
Hello from C loop index 6
Hello from C loop index 7
Hello from C loop index 8
Hello from C loop index 9
```

The hello.c Program

```
1  # include <stdio.h>
2
3  void say_hello ( );
4
5  int main ( )
6  {
7    int n = 10;
8    say_hello ( n );
9    return 0;
10 }
11
12 void say_hello (  n )
13 {
14   for ( i = 0; i < n; i++ )
15   {
16     printf ( "Hello from C loop index %d\n", i );
17   }
18   return;
19 }
```

Compile and Run a CUDA program

A CUDA program is similar to a C program, with a few differences:

- The file extension is typically `.cu`,
- We use a compiler from NVidia called `nvcc`,
- We have to be running on a CPU that has an attached GPU.

```
nvcc -o hello hello.cu
./hello
```

```
Hello from CUDA thread 0
Hello from CUDA thread 1
Hello from CUDA thread 2
Hello from CUDA thread 3
Hello from CUDA thread 4
Hello from CUDA thread 5
Hello from CUDA thread 6
Hello from CUDA thread 7
Hello from CUDA thread 8
Hello from CUDA thread 9
```

The hello.cu Program

```
 1  # include <stdio.h>
 2
 3  __global__ void say_hello ( );
 4
 5  int main ( )
 6  {
 7    int n = 10;
 8    say_hello <<< 1, n >>> ( );
 9    return 0;
10  }
11
12  __global__ void say_hello ( )
13  {
14    int i = threadIdx.x;
15
16    printf ( "Hello from CUDA thread %d\n", i );
17
18    return;
19  }
```

A kernel function runs on the GPU

If we look at `hello.cu`, we see that function `say_hello` :

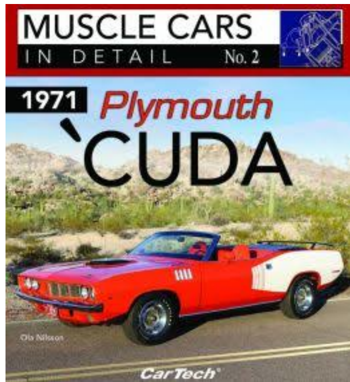
- seems to do the printing work;
- does not use a loop to print 10 times;
- is declared using a new qualifier: `__global__`;
- gets a thread number from structure `threadIdx.x`;
- is invoked with a *triple chevron*: `<<< 1, n >>>`;

The function `say_hello` is an example of a kernel function, which is a function that will run on the GPU, but is callable by the CPU.

A kernel function runs on the GPU

- seems to do the printing work;
  **** yes, and it runs on the GPU!*
- does not use a loop to print 10 times;
  **** yes, instead, the function is called 10 times;*
- is declared using a new qualifier: `__global__`;
  ***** `__global__` *is a GPU function that the CPU can call;*
- gets a thread number from structure `threadIdx.x`;
  **** each call to say_hello has a different thread number;*
- is invoked with a *triple chevron*: `<<< 1, n >>>`;
  **** this call determines how many times say_hello is called;*

ADD VECTORS Test Drive: Does It Move?



Moving Data between CPU and GPU

We saw in our `hello.cu` example how we could transfer *program control* from the CPU to the GPU and back again.

But the interesting task is to see that we can move **data** back and forth, so that the CPU can set up a problem, the GPU can solve it, and the CPU can retrieve the results.

As a model problem, we will look at the task of computing the pairwise sum of two vectors, which could be done by the C loop

```
1    for ( i = 0; i < n; i++ )
2    {
3      c[i] = a[i] + b[i];
4    }
```

Managing Two Copies of Data

The CPU and GPU have separate data spaces, so it looks like we will have to figure out a way to declare two copies of the vectors, and figure out how to assign them on the CPU, move them to the GPU and operate on them there, and then bring them back.

To keep from going crazy, we'll use pairs of names, such as a_cpu and a_gpu to distinguish the two sets of data.

The CPU program will have to manage memory on the GPU with the `cudaMalloc()` and `cudaFree()` commands.

Data can be transferred from the CPU to the GPU by:

```
cudaMemcpy ( a_gpu, a_cpu, memsize, cudaHostToDevice );
```

and results pulled back from the GPU to the CPU by:

```
cudaMemcpy ( c_cpu, c_gpu, memsize, cudaDeviceToHost );
```

The vecadd.cu Main Program

```
int main ( )
{
  float *a_cpu, *a_gpu, *b_cpu, *b_gpu, *c_cpu, *c_gpu;
  ns = n * sizeof ( float );

  a_cpu = ( float * ) malloc ( ns );
  b_cpu = ( float * ) malloc ( ns );
  loadArrays ( a_cpu, b_cpu, n );

  cudaMalloc ( ( void** ) &a_gpu, ns );
  cudaMalloc ( ( void** ) &b_gpu, ns );
  cudaMalloc ( ( void** ) &c_gpu, ns );
  cudaMemcpy ( a_gpu, a_cpu, ns, cudaMemcpyHostToDevice );
  cudaMemcpy ( b_gpu, b_cpu, ns, cudaMemcpyHostToDevice );

  add_vectors <<< 1, n >>> ( a_gpu, b_gpu, c_gpu );

  c_cpu = ( float * ) malloc ( ns );
  cudaMemcpy ( c_cpu, c_gpu, ns, cudaMemcpyDeviceToHost );
}
```

The Add_Vectors Function

```
1  __global__ void add_vectors ( float *a_gpu, float *b_gpu,
2    float *c_gpu )
3  {
4    int i = threadIdx.x;
5
6    c_gpu[i] = a_gpu[i] + b_gpu[i];
7
8    return;
9  }
```

Walk through `vecadd.cu`

The story begins with the creation and assignment of the vectors `a` and `b` on the CPU, just like any C program would do.

Next the CPU must allocate space on the GPU for `a`, `b`, and `c`, and then transfer the values of `a` and `b` to the GPU.

Then the CPU transfers control to the GPU by calling the kernel function, `add_vectors`.

The GPU invokes `n` threads, each one executing a single instance of `add_vectors` to set a single entry of `c`.

Once the kernel function is finished, the CPU regains control and copies back the computed vector `c`.

Separate CPU and GPU Memories

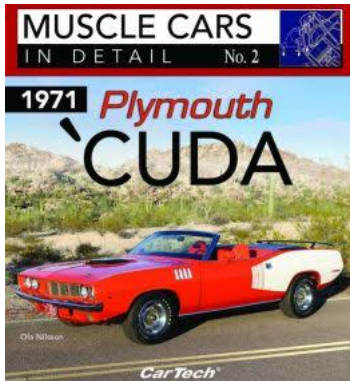
You should think about the fact the the vectors `a`, `b`, and `c` were stored on both the CPU and GPU.

It is the programmer's intent that these pairs of arrays are equal, but that only happens if a `cudaMemcpy()` command is used.

Otherwise, changes to an array made on the CPU do not affect the GPU copy, and vice versa.

In some cases, such as an iterative algorithm, the GPU might update array values many times. Since data transfers take some time, a CUDA program will typically wait until the final result is computed on the GPU, and only copy those completed values back to the CPU.

COLLATZ Test Drive: How Fast Does it Go?



The Collatz Sequence

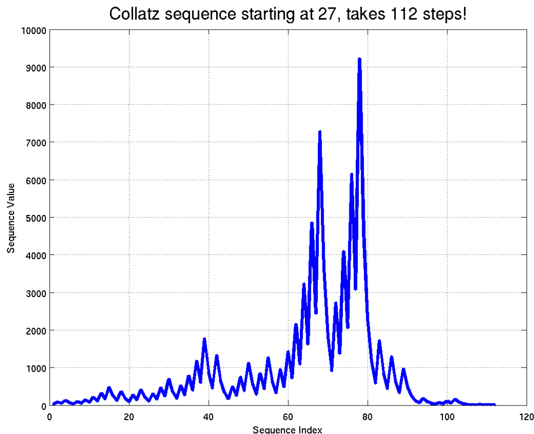
Start with any positive number, divide it by 2 if even, or triple and add 1 if odd, and stop when you reach the value 1. It's not clear why, but the process always does seem to reach 1.

Suppose we start with the number 17?

```
 0  17 is odd,  so 17 --> 3*17+1 = 52;
 1  52 is even, so 52 --> 52/2   = 26;
 2  26 is even, so 26 --> 26/2   = 13;
 3  13 is odd,  so 13 --> 3*13+1 = 40;
 4  40 is even, so 40 --> 40/2   = 20;
 5  20 is even, so 20 --> 20/2   = 10;
 6  10 is even, so 10 --> 10/2   = 5;
 7   5 is odd,  so  5 --> 3*5+1  = 16;
 8  16 is even, so 16 --> 16/2   = 8;
 9   8 is even, so  8 --> 8/2    = 4;
 0   4 is even, so  4 --> 4/2    = 2;
11   2 is even, so  2 --> 2/2    = 1;
12   1 is 1, so we stop after 12 steps.
```

The Collatz Sequence

An interesting question is, for any starting value **n**, how many iterations does it take before the sequence reaches 1 (or the hailstone hits the ground?) For 17, this value is 12; for 1, this value is 0. What about 27?



The program collatz.cu

We can set up a CUDA program to compute the length of the Collatz sequence for every integer from 1 to N. In this case, there is no need for the CPU to initialize any data array to be sent to the GPU; the GPU simply starts up threads 0 through N-1, using each thread index (plus 1) as a starting value.

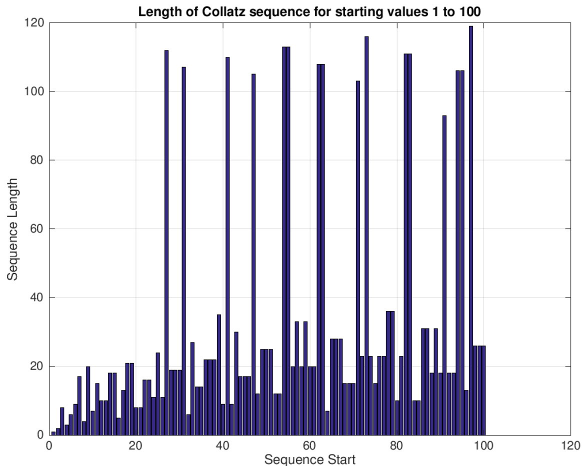
```
1  steps_num = 100;
2  steps_size = steps_num * sizeof ( int );
3  cudaMalloc ( ( void** ) &steps_gpu, steps_size );
4
5  collatz_steps <<< 1, steps_num >>> ( steps_gpu );
6
7  steps_cpu = ( int * ) malloc ( steps_size );
8  cudaMemcpy ( steps_cpu, steps_gpu, steps_size,
       cudaMemcpyDeviceToHost );
```

The kernel collatz_steps

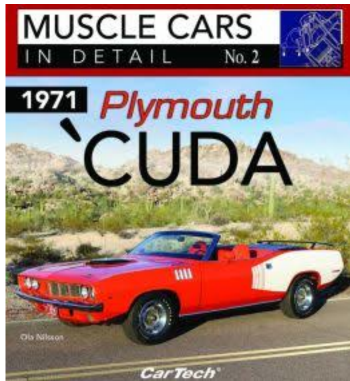
```
1  __global__ void collatz_steps ( int *steps_gpu )
2  {
3     int i, n, s;
4
5     i = threadIdx.x;
6     n = i + 1;
7     s = 0;
8
9     while ( 1 < n )
10    {
11       if ( ( n % 2 ) == 0 )
12       {
13          n = n / 2;
14       }
15       else
16       {
17          n = 3 * n + 1;
18       }
19       s = s + 1;
20    }
21
22    steps_gpu[i] = s;
23
24    return;
25 }
```

The Collatz Sequence

Here is the variation in sequence length for starting points 1 to 100:



JACOBI Test Drive: Does It Go Faster?



Jacobi Iteration

Jacobi iteration is a standard iterative technique for estimating the solution of certain linear systems A*x=b.

Jacobi iteration is a common example of a parallel algorithm. In outline,

```
for steps 1 to stepmax:
  for indices 1 to n:
    compute xnew(i) from x values.
  overwrite all x values by xnew values.
```

If we suppose the matrix is the -1,2,-1 matrix, then the form of the kernel would seem to be easy. Except for the first and last values of x, we compute

```
xnew[i] = ( - x[i-1] - x[i+1] - b[i] ) / 2.0
```

When all the xnew's have been computed we want to do an overwrite, but the kernel doesn't know when it's safe to overwrite. What do we do?

The CPU Helps Synchronize

Although we may want to compute hundreds of Jacobi iterations, we need to enforce a barrier of some kind, at which time we know that every entry of the new solution estimate x2 has been computed. Then we need to ensure that the new data is used on the next iteration.

One way to manage this is to write the main CPU program so that the GPU kernel is called in pairs of steps:

```
for steps 1 to stepmax
  jacobi <<< 1, n >>> ( x used to compute x2 )
  jacobi <<< 1, n >>> ( x2 used to compute x )
```

Although we are now forced to take an even number of steps, we are guaranteed to have the appropriate synchronization, because the kernel will return to the CPU each time it has completed an update.

The main program jacobi.cu

```
1   n = 10;
2   memsize = n * sizeof ( float );
3   b_cpu = ( float * ) malloc ( memsize );
4   x_cpu = ( float * ) malloc ( memsize );
5
6   initialize ( n, b, x );
7
8   cudaMalloc ( ( void** ) &b_gpu, memsize );
9   cudaMalloc ( ( void** ) &x1_gpu, memsize );
10  cudaMalloc ( ( void** ) &x2_gpu, memsize );
11
12  cudaMemcpy( b_gpu, b_cpu, memsize, cudaMemcpyHostToDevice);
13  cudaMemcpy( x1_gpu, x_cpu, memsize, cudaMemcpyHostToDevice);
14
15  for ( it = 0; it < 100; it++ )
16  {
17    jacobi <<< 1, n >>> ( b_gpu, x1_gpu, x2_gpu, n );
18    jacobi <<< 1, n >>> ( b_gpu, x2_gpu, x1_gpu, n );
19  }
20
21  cudaMemcpy( x_cpu, x1_gpu, memsize, cudaMemcpyDeviceToHost);
```

The kernel jacobi()

```
 1 __global__ void jacobi ( float *b_gpu, float *x1_gpu,
 2    float *x2_gpu, int n )
 3 {
 4    int i = threadIdx.x;
 5    float xim1, xip1;
 6
 7    if ( 0 < i )
 8    {
 9       xim1 = x1_gpu[i-1];
10    }
11    else
12    {
13       xim1 = 0.0;
14    }
15    if ( i < n - 1 )
16    {
17       xip1 = x1_gpu[i+1];
18    }
19    else
20    {
21       xip1 = 0.0;
22    }
23    x2_gpu[i] = 0.5 * ( b_gpu[i] + xim1 + xip1 );
24    return;
25 }
```

If There's One Awkward Solution, There's Always More

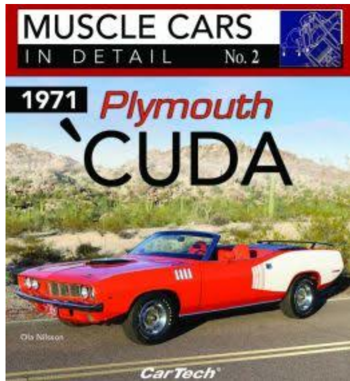
Of course, it seems a bit awkward to have to do pairs of kernel calls in this way.

Here are some alternate approaches:

- Forget about `x1` and `x2`. Just overwrite `x` by its jacobi-improved value, and iterate 100 times.
- add a second GPU kernel that swaps `x1` and `x2` after each jacobi step;
- cudaMemcpy `x2` back to CPU as `x`, then cudaMemcpy `x` back to GPU as `x1`;
- on the CPU, try to swap the pointers to `x1` and`x2`;

These may or may not work, but in every case, they seem at least as awkward as the method we have looked at.

Are You Sold on CUDA?



Your Own Copy of CUDA

If you have a laptop computer, you may be able to install CUDA on it.

To see if your Linux system will allow an installation of CUDA, look for "nvidia" in your system:

```
lspci | grep -i nvidia
```

Verify that you have a supported version of LINUX:

```
uname -m && cat /etc/*release
```

If the output includes the line **x86_64**, then your Linux is supported.

One way to get CUDA is from:

```
https://developer.nvidia.com/cuda-downloads
```

Before using CUDA, you may have to issue commands like:

```
export PATH=/usr/local/cuda-8.0/bin/$PATH
export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64:$LD_LIBRARY
```

GPU's on the ARC Clusters

Virginia Tech's Advanced Research Computing (ARC) maintains a number of computer clusters. The clusters vary in age, size, capability. And in particular, only some of the ARC cluster machines have nodes with GPU's.

Cluster	GPU Type	GPU Nodes	GPU's per Node	CUDA Cores per GPU
BlueRidge	Kepler K40	4	2	2,880
Cascades	—	—	—	—
DragonsTooth	Kepler K80	4	2	2,496
HokieOne	—	—	—	—
HokieSpeed	Fermi C2050	204	2	448
NewRiver	Kepler K80	8	2	2,496

Since HokieSpeed is being decommissioned in June, we recommend that users interested in GPU's try out the NewRiver cluster.

The Module Command for CUDA

In order to compile and run a CUDA program on the cluster, certain **module** commands must be issued to set up the environment.

A typical command would be

```
module load cuda
```

This makes the CUDA compiler **nvcc** available.

The CUDA compiler works like a typical C compiler; it expects your source to have the extension `.cu`. To compile and execute a program whose source is in the file *jacobi.cu*, the commands would be

```
module load cuda
nvcc -o jacobi jacobi.cu
./jacobi
```

Running a Job

On the ARC clusters, most jobs are not run directly or interactively.

Instead, the necesary commands are given inside of a batch file and preceded by a list of somewhat mysterious statements that

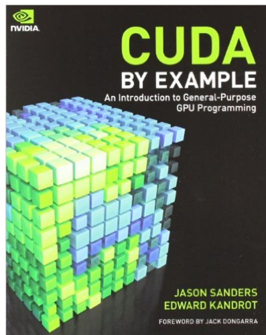
1. specify that the batch file is a BASH script;
2. tell PBS the resources you request;
3. move you to your working directory;
4. issue the **module** commands to set up your environment;
5. run your commands.

The Batch File for CUDA

```
1  #! /bin/bash
2  #PBS -l walltime=00:05:00
3  #PBS -l nodes=1:ppn=1:gpus=1
4  #PBS -W group_list=newriver
5  #PBS -q open_q
6  #PBS -j oe
7  #
8  cd $PBS_O_WORKDIR
9  #
10 module purge
11 module load cuda
12 #
13 nvcc -o collatz collatz.cu
14 #
15 ./collatz
16 #
17 rm collatz
```

The Owner's Manual:

NVidia has published a useful and readable CUDA guide:



CUDA by Example, by Jason Sanders and Edward Kandrot.

Information about the book and the examples is at:
`https://developer.nvidia.com/cuda-example`