CUDA programing language

Đỗ Trí Nhân

Content

Contents

I. Introduction

By definition, CUDA is architecture and programming model developed by NVIDIA to run parallel computing on graphics processing units (GPUs)
CUDA is the acronym for Compute Unified Device Architecture
CUDA components include:

- Programming language based on C++
    - CUDA language is based on C++ with a few additional keywords and concepts, which makes it fairly easy for non-GPU programmers to pick up. It lacks the use of object oriented or C++ features in device code.
    - Third party wrappers are also available for Python, Perl, Fortran, Java, Ruby, Lua, Common Lisp, Haskell, R, MATLAB, IDL, Julia, and native support in Mathematica."
- Software development kit
    - includes libraries, debugging and optimization tools, a compiler, documentation, and a runtime library to deploy your applications.)
- Massively parallel hardware designed

This report will focus on Programming language part and some hardware design

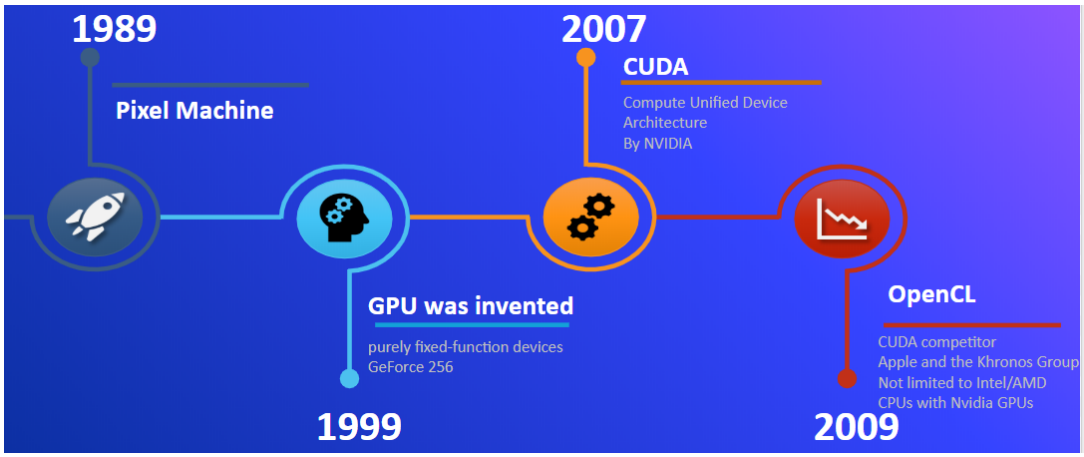
History

In 1989, the system architecture and the programming environment of the Pixel Machine - a parallel image computer with a distributed frame buffer was first introduced

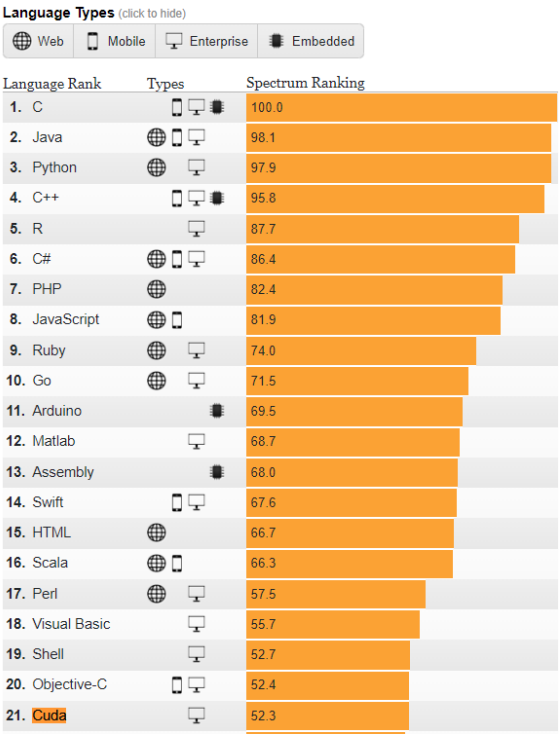
GPU was first invented by NVidia in 1999. Nvidia started trying to compete in the 3D accelerator market with weak products in 1996, but learned as it went, and in 1999 introduced the successful GeForce 256, the first graphics card to be called a GPU. At the

time, the principal reason for having a GPU was for gaming. It wasn't until later that people used GPUs for math, science, and engineering.

In 2003, a team of researchers led by Ian Buck unveiled Brook, the first widely adopted programming model to extend C with data-parallel constructs. Buck later joined Nvidia and led the launch of CUDA in 2006, the first commercial solution for general-purpose computing on GPUs.



In 2009 OpenCL was launched by Apple and the Khronos Group, in an attempt to provide a standard for heterogeneous computing that was not limited to Intel/AMD CPUs with Nvidia GPUs. CUDA is still prefered for parallel programming despite only being possible to run the code in a NVidia's graphics card. On the other hand, many programmers prefer to use OpenCL because it may be considered as a heterogeneous system and be used with GPUs or CPUs multicore



Popularity

By the rankings of the top programming languages of IEEE in 2016, CUDA is in the 21st

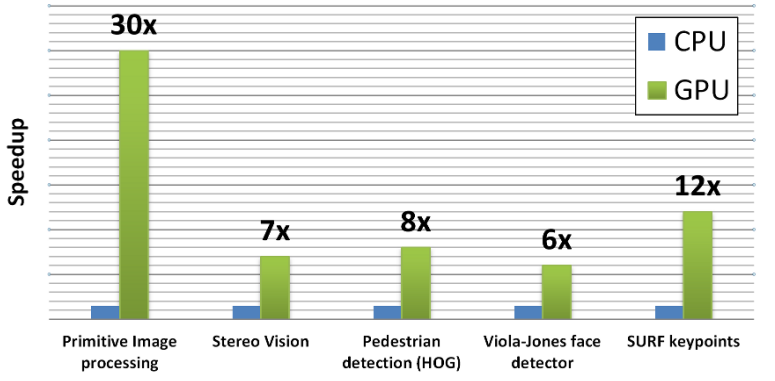
place. These rankings are based upon the trends that were noticed and take into account several metrics. The data used for ranking is from Google search, Github, Stack Overflow, Reddit and etc. CUDA is used in Enterprise, mainly for calculation and research, rarely used in web and application programming because it is difficult to access and depends on hardware. At start, they were used for graphics purposes only. But now GPUs are becoming more and more popular for a variety of general-purpose, non-graphical applications too. For example they are used in the fields of computational chemistry, sparse matrix solvers, physics models, sorting, and searching

Benefits and Limitations

We use CUDA when we have lots of data, and lots of computations

The benefits of CUDA is:

- Harnesses the power of the GPU by using parallel processing;
- Running thousands of simultaneous reads instead of single, dual, or quad reads on the CPU.
- C/C++ is widely used, easy to learn how to program for CUDA.
- Most graphics cards of NVIDIA support CUDA.
- Huge increase in processing power over conventional CPU processing. Early reports suggest speed increases of 5x to 200x over CPU processing speed.



As we can find from the figure above, performance when using GPU for computer vision methods is faster than using CPU.

The limitations of CUDA over traditional general purpose computation on GPUs is:

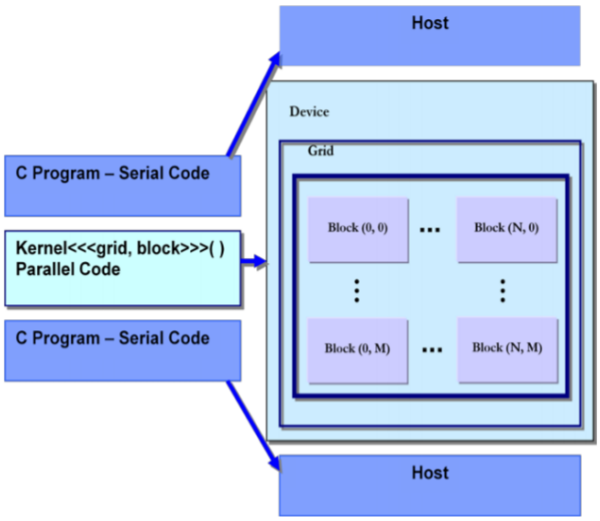
- Latency between the CPU and GPU
  Copying between host and device memory may incur a performance hit due to system bus bandwidth
- A single process must run spread across multiple memory spaces
  Threads should be running in groups of at least 32 for best performance
- CUDA-enabled GPUs are only available from NVIDIA (Unlike OpenCL)

II. System Structure

NVIDIA's graphics processing units (GPUs) are very powerful and highly parallel. GPUs have hundreds of processor cores and thousands of threads running concurrently on these cores, thus because of intensive computing power they are much faster than the CPU

Heterogeneous

Heterogeneous architecture is used to make an interaction between CPU and GPU programming models. Data may be copied from host memory to device memory and the results are copied back to host from the device memory.



Parallel execution is expressed by the

kernel function that is executed on a set of threads in parallel on GPU; GPU is also called device. This kernel code is a C code for only one thread. The numbers of thread blocks, and the number of threads within those blocks execute this kernel in parallel.

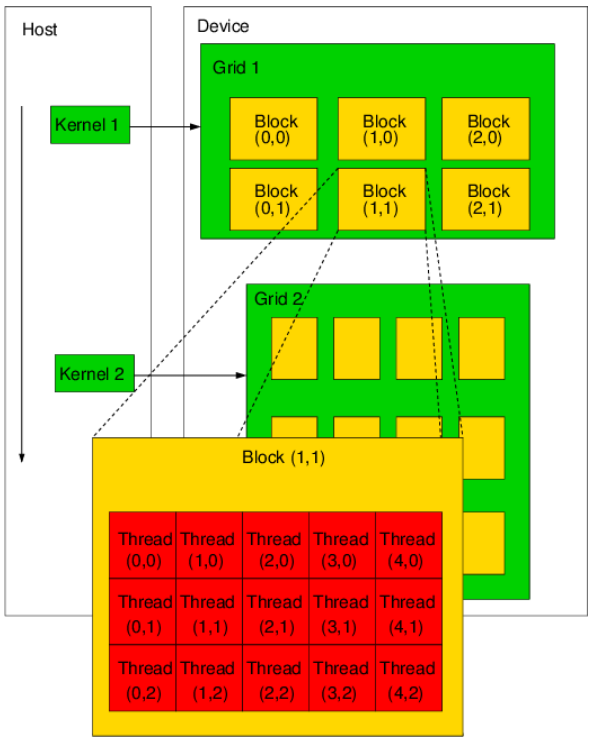
Paradigm

Since CUDA is extended from C++, it supports multiple-paradigms: imperative, object-oriented, functional and especially **parallel paradigm**.

Parallel programming is its strongest point, since it not only runs multiple tasks at the same time but also can get the combination of serial and parallel executions.

Programming Model

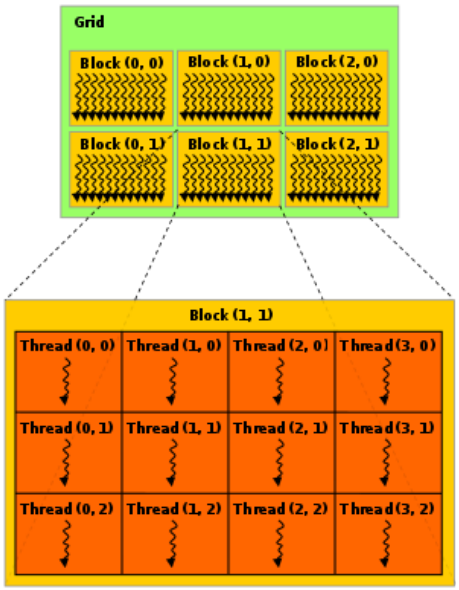
CUDA kernel is specific functions in CUDA, a kernel can be a function or a full program invoked by the CPU. It is executed N number of times in parallel on GPU by using N number of threads



Some properties of model:

- In each block of a grid, there are up to 512, 1024 or 2048 threads
- All blocks will define a grid and execute the same program (kernel) and they are independent
- Only one kernel can run at a time

The Grid consists of one-dimensional, two-dimensional or three-dimensional thread blocks. Each thread block is further divided into one-dimensional or two-dimensional threads. A thread block is a set of threads running on one processor



Thread

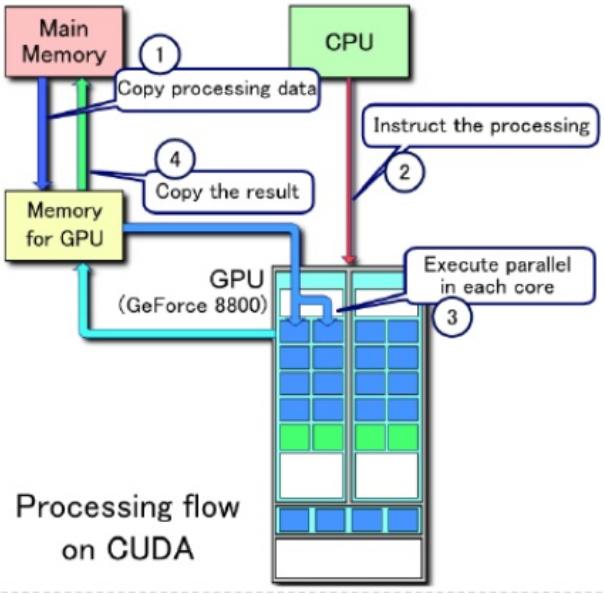
NVIDIA's graphics card is a new technology that is extremely multithreaded computing architecture. It consists of a set of parallel multiprocessors, that are further divided into many cores and each core executes instructions from one thread at a time

All threads of a single thread block can communicate with each other through shared memory; therefore they are executed on the same multiprocessor. In this way it becomes possible to synchronize these threads.

For synchronization purposes among threads CUDA API provides a hardware thread-barrier function syncthreads() that acts as synchronization point. As threads are scheduled in hardware, this function is implemented in hardware. The threads will wait at the synchronization point until all of the threads have reached this point. The communication among threads (if required) is possible through per-block shared memory. Hence thread synchronization is possible only at thread block level. Since threads of a thread block may communicate with each other, these threads must execute on the same processor.

Processing Flow

As the kernel function runs on the device, memory must be allocated on device in advance before kernel function invocation and if the kernel function has to execute on some data then the data must be copied from the host memory to the device memory.



First of all, when a CUDA program is executed, with the code for running in GPU, data is copied from main memory to GPU memory.

Then CPU will instructs the process to GPU for execution

GPU execute parallel tasks in each core

After running all tasks in GPU, the system copy result from GPU memory to Main memory

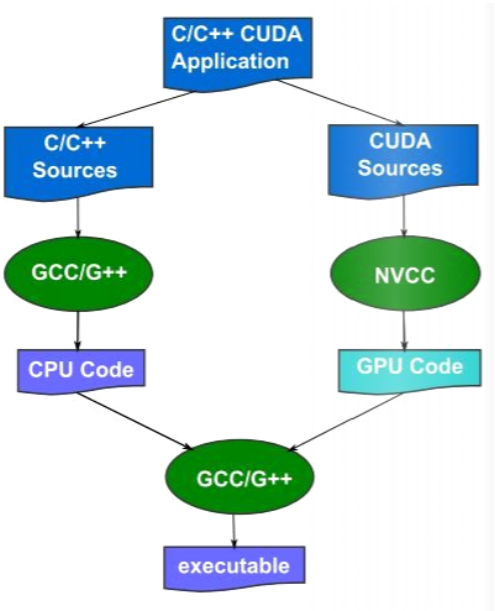
```
cudaMemcpy( d_a, a, SIZE*sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( d_b, b, SIZE*sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( d_c, c, SIZE*sizeof(int), cudaMemcpyHostToDevice );

VectorAdd<<< 1, SIZE >>>(d_a, d_b, d_c, SIZE);

cudaMemcpy( c, d_c, SIZE*sizeof(int), cudaMemcpyDeviceToHost );
```

Processing Flow in Programming

Since the bandwidth between the device memory and the host memory is much less as compared to the bandwidth between the device and the device memory which is very high, we should try to minimize data transfer between the host and the device. Some of the efforts could be like moving some code from the host to the device and creating and destroying data structures in the device memory (instead of copying them to the device)

Compiler

Because CUDA programs include code running in CPU and GPU, therefore, for each partition of code we compile each of them. GCC/G++ Compiler for Host code, NVCC Compiler for device code and GCC/G++ for linking both of them to create the execution file.



III. Syntax and Semantic

1. Identifier

RULES FOR NAMING IDENTIFIER

- Case sensitive
- Cannot start with a digit
- Underscore can be used as first character
- Other special characters are not allowed
- Cannot use keywords as identifiers.

For example:

- Legal identifier: a, bcd, _this, w1c, nm_2, _, …
- Illegal identifier: 1cvb, @qwe, ab@, ...

2. Specification

a. Function qualifier

```
#include <stdio.h>

__global__
void add(int *a, int *b, int *c, int n) {

      int i = threadIdx.x;

      __shared__ bool sharedArr[100];

      for (int i = 0; i < n; i++)
              c[i] = a[i] + b[i];
}
```

Some function qualifiers:

- __global__ :

  The functions with global qualifier are executed on the device but they are callable from the host only.

- __device__ :

  The functions with device qualifier are executed on the device. These functions are callable from the device only.

- __host__ :

  The functions with host qualifier are executed on the host and are callable from the host only. When no qualifier is used, it means that the function will run on the host; it is equivalent to the function declared with the _host_ qualifier.

b. Function call

```
      for (int i = 0; i < SIZE; i++) {
              a[i] = i;
              b[i] = i;
      }

      add<<<1,1>>>(a,b,c,SIZE);

      cudaDeviceSynchronize();

      cudaFree(a);
      cudaFree(b);
      cudaFree(c);
```

In the kernel function call grid and block variables are written in three angular brackets <<< grid, block >>>

Syntax for calling function:

kernel_function<<<M,N>>>(list of arguments);

M is number of block

N is number of threads each block (size of block)

M*N is number of times that function is executed.

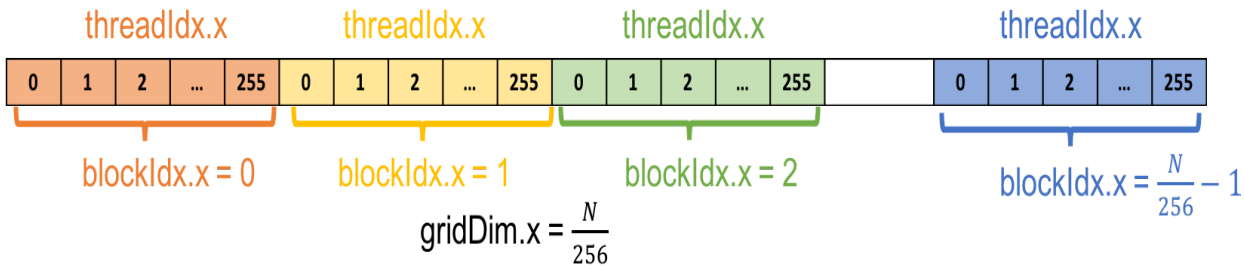
c. Device variable

```
__global__
void add(int *a, int *b, int *c, int n) {

      int i = threadIdx.x;

      __shared__ bool sharedArr[100];

      for (int i = 0; i < n; i++)
              c[i] = a[i] + b[i];
}
```

The CUDA paradigm provides some built-in variables to use this structure efficiently.

Some device variables:

- dim3 gridDim : Dimensions of the grid in blocks
- dim3 blockDim: Dimensions of the block in threads
- dim3 blockIdx: Block index within the grid
- dim3 threadIdx: Thread index within the block



The thread index i is calculated by the following formula :

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

The allocation of the number of thread blocks to each multiprocessor is dependent on the necessity of the shared memory and registers by each thread block. More memory and registers requirement by each thread block means allocation of less thread blocks to each multiprocessor.

d. Variable qualifiers

```
__global__
void add(int *a, int *b, int *c, int n) {

      int i = threadIdx.x;

      __shared__ bool sharedArr[100];

      for (int i = 0; i < n; i++)
              c[i] = a[i] + b[i];
}
```

Here are some variable qualifiers:

- __constant__ :
  This qualifier is used to allocate constants on the device. It is optionally used together with __device__ qualifier. This constant resides in constant memory, and has the

lifetime of an application. It is accessible from all the threads (within grid) and host through the runtime library.

- __shared__ :

  This qualifier is used to allocate the shared variable. It is optionally used together with __device__ qualifier. Shared variable resides in shared memory of a thread block, and has the lifetime of a block. It is only accessible from all the threads within the block.

- __device__ : accessible by all threads.

  The variables declared with __device__ reside on the device. Other type qualifiers are optionally used together with __device__. If a variable is declared only with __device__ qualifier then this variable resides in the global memory and it has the lifetime of the application. Since it resides in the global memory, it is accessible from all the threads (within the grid) and host through the runtime library.

IV. Experiment

In our GPU using for experiment, maximum number of block is 65535 and maximum number of threads each block = 1024

(Compute Capability: 3.5)

Problem 1

Adding two vectors a and b, resulted in vector c

We will do this by adding each element at each index position together and separately.

1. **Without parallelism**

ss

```
#include <stdio.h>

__global__
void add(int *a, int *b, int *c, int n) {

      // Iteration to do addition
      for (int i = 0; i < n; i++)
            c[i] = a[i] + b[i];
}
```

```
int main() {
      int *a, *b, *c;
      int SIZE = 1000000;

      // Allocation memory in devices for a,b and c
      cudaMallocManaged(&a, SIZE*sizeof(int));
      cudaMallocManaged(&b, SIZE*sizeof(int));
      cudaMallocManaged(&c, SIZE*sizeof(int));

      // Simple initialization
      for (int i = 0; i < SIZE; i++) {
            a[i] = i;
            b[i] = i;
      }

      // Launch kernel on device
      add<<<1,1>>>(a,b,c,SIZE);

      // Free memory
      cudaFree(a);
      cudaFree(b);
      cudaFree(c);

      return 0;
}
```

Using only one block with one thread.

Using a loop to do addition on each index.

```
int main() {
      int *a, *b, *c;
      int SIZE = 1000000;

      // Allocation memory in devices for a,b and c
      cudaMallocManaged(&a, SIZE*sizeof(int));
      cudaMallocManaged(&b, SIZE*sizeof(int));
      cudaMallocManaged(&c, SIZE*sizeof(int));

      // Simple initilization
      for (int i = 0; i < SIZE; i++) {
            a[i] = i;
            b[i] = i;
      }

      // Launch kernel function with 1000 block
      // Each block has 1000 threads
      add<<<1000,1000>>>(a,b,c,SIZE);

      // Free memory
      cudaFree(a);
      cudaFree(b);
      cudaFree(c);

      return 0;
}
```

2. With parallelism

Using only 1000 blocks, each block has 1000 threads.Each thread we do addition on one index.

```
#include <stdio.h>

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

3. Result

```
==12305== NVPROF is profiling process 12305, command: ./add_normal
==12305== Profiling application: ./add_normal
==12305== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min
 GPU activities:  100.00%  266.64ms         1  266.64ms  266.64ms
      API calls:   50.29%  274.20ms         3  91.401ms  831.23us
                   49.00%  267.15ms         1  267.15ms  267.15ms
                    0.42%  2.3071ms         1  2.3071ms  2.3071ms
                    0.19%  1.0519ms         3  350.62us  321.01us
                    0.04%  236.00us        96  2.4580us     191ns
                    0.04%  213.08us         1  213.08us  213.08us
                    0.00%  22.577us         1  22.577us  22.577us
                    0.00%  3.6050us         1  3.6050us  3.6050us
                    0.00%  1.8500us         3     616ns     235ns
                    0.00%  1.2910us         2     645ns     262ns
                    0.00%     339ns         1     339ns     339ns
```

No parallelism

```
==11831== NVPROF is profiling process 11831, command: ./add_parallel
==11831== Profiling application: ./add_parallel
==11831== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min
 GPU activities:  100.00%  57.537us         1  57.537us  57.537us
      API calls:   98.42%  262.15ms         3  87.383ms  631.43us
                    0.95%  2.5211ms         1  2.5211ms  2.5211ms
                    0.39%  1.0363ms         3  345.43us  300.18us
                    0.09%  249.13us        96  2.5950us     192ns
                    0.08%  204.57us         1  204.57us  204.57us
                    0.05%  145.72us         1  145.72us  145.72us
                    0.01%  31.471us         1  31.471us  31.471us
                    0.00%  4.1350us         1  4.1350us  4.1350us
                    0.00%  2.2310us         3     743ns     227ns
                    0.00%  1.4950us         2     747ns     402ns
                    0.00%     324ns         1     324ns     324ns
```

Parallelism

	GPU one thread	GPU parallel	CPU
Runtime	266.64ms	51.537 us	0.0072s

From the table above, we can find that when running GPU with only one thread, it takes more time than running in CPU because of its powerful computing. But when running GPU CUDA with parallel core, multi-thread, GPU performs faster than CPU.

Problem 2

To see the parallel processing power of CUDA, let's try Solving 1000 quadratic equations
We was going to set the parameter A,B,C of quadratic equation in random, then execute the program writing in CUDA C++ , the result is 0.65ms

```
__device__
void equation(int a, int b, int c) {
        float delta = b*b - 4*a*c;
        if (delta > 0)
                printf("There are two roots: %f %f \n",
                                (-b-sqrt((float)delta))/(2*a),
                                (-b+sqrt((float)delta))/(2*a));
        else
                if (delta == 0)
                        printf("There is one root: %f\n",-b/(2*a));
}
```

We will solve each quadratic equation in **equation** function, which can be called from other GPU code by using function qualifier **__device__**.

```
__global__
void add(int *a, int *b, int *out, int n) {
        int thread = threadIdx.x;
        int block = blockIdx.x;
        int dim = blockDim.x;

        int index = thread + block * dim;
        if (index < n)
                equation(a[index],b[index],out[index]);
}
```

In add function, we will calculate the thread index of the current thread and then solve the quadratic equation by passing the corresponding parameters to **equation**.

```
int main() {
        int _$=9;
        int *a, *b, *out;
        int SIZE = 10000;

        cudaMallocManaged(&a, SIZE*sizeof(int));
        cudaMallocManaged(&b, SIZE*sizeof(int));
        cudaMallocManaged(&out, SIZE*sizeof(int));

        srand(time(NULL));
        for (int i = 0; i < SIZE; i++) {
                a[i] =-100+(std::rand()%200);
                b[i] =-100+(std::rand()%200);
                out[i] =-100+(std::rand()%200);
        }

        add<<<100,100>>>(a,b,out,SIZE);
        cudaDeviceSynchronize();

        //for (int i = 0; i < SIZE; i++) cout << out[i] << ' ';

        cudaFree(a);
        cudaFree(b);
        cudaFree(out);

        return 0;
}
```

Finally, in the main function, we will allocate memory in the device, initialize random parameters for each equation and call the function **add** using 100 blocks, each block has 100 threads, with a total of 10000 parallel threads, which is equal to the **SIZE** - number of quadratic equations.

V. More about CUDA

Opponent

There have been other proposed APIs for GPUs, such as OpenCL, and there are competitive GPUs from other companies, such as AMD, the combination of CUDA and Nvidia GPUs dominates several application areas, including deep learning, and is a foundation for some of the fastest computers in the world.

CUDA in deep learning

Deep learning has an outsized need for computing speed. For example, to train the models for Google Translate in 2016, the Google Brain and Google Translate teams did hundreds of one-week TensorFlow runs using GPUs; they had bought 2,000 server-grade GPUs from Nvidia for the purpose. Without GPUs, those training runs would have taken months rather than a week to converge. For production deployment of those TensorFlow translation models, Google used a new custom processing chip, the TPU (tensor processing unit).

In addition to TensorFlow, many other DL frameworks rely on CUDA for their GPU support, including Caffe2, CNTK, Databricks, H2O.ai, Keras, MXNet, PyTorch, Theano, and Torch. In most cases they use the cuDNN library for the deep neural network computations. That library is so important to the training of the deep learning frameworks that all of the frameworks using a given version of cuDNN have essentially the same performance numbers for equivalent use cases.

References

1. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html
2. https://stackoverflow.com/questions/5211746/what-is-cuda-like-what-is-it-for-what-are-the-benefits-and-how-to-start

3. https://www.tutorialspoint.com/cuda/cuda_key_concepts.htm
4. https://www.tutorialspoint.com/cuda/cuda_keywords_and_thread_organization.htm
5. https://developer.nvidia.com/cuda-faq
6. https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html
7. https://www.sciencedirect.com/science/article/pii/S0167819119301759
8. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016#index/2016/1/1/1/1/1/50/1/50/1/50/1/30/1/30/1/30/1/20/1/20/1/5/1/5/1/20/1/100/
9. https://medium.com/@inaccel/cpu-gpu-fpga-or-tpu-which-one-to-choose-for-my-machine-learning-training-948902f058e0
10. http://www.diva-portal.org/smash/get/diva2:447977/FULLTEXT01.pdf
11. http://proceeding.vap.ac.vn/index.php/proceedingvap/article/view/000214/203
12. https://www.slideserve.com/ryu/cuda-programming
13. https://www.myzhar.com/blog/tutorials/tutorial-nvidia-gpu-cuda-compute-capability/
14. https://books.google.com.vn/books?id=CYxjDwAAQBAJ&pg=PA7&lpg=PA7&dq=pytorch+is+wrapper+of+cuda&source=bl&ots=43Smx_h6jw&sig=ACfU3U20lxlu8I4kSDSTS5RGQEFk6QMnuQ&hl=en&sa=X&ved=2ahUKEwiz_tLSqa7pAhXOzIsBHU55DCMQ6AEwD3oECBIQAQ#v=onepage&q=pytorch%20is%20wrapper%20of%20cuda&f=false
15. http://www.c4learn.com/cplusplus/cpp-variable-naming/
16. https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html
17. https://tatourian.blog/2013/09/03/nvidia-gpu-architecture-cuda-programming-environment/
18. https://www.yumpu.com/en/document/read/50976433/cuda-parallel-programming-tutorial
19. http://www.diva-portal.org/smash/get/diva2:447977/FULLTEXT01.pdf
20. https://www.slideserve.com/ryu/cuda-programming
21. https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf
22. https://dl.acm.org/doi/10.1145/74334.74340
23. https://spectrum.ieee.org/ns/IEEE_TPL_2016/methods.html