
# Reviewing CUDA Based On GPU

Name: Yinhao Xiao

Course Name: Computer System Architecture

Instructor Name: Morris Lancaster

# Table of Contents

# Abstract

Graphic Processing Unit(GPU), has been rapidly developing since it had been invented and introduced to the public. However, early use of the GPU mainly focuses on rendering 3D models in computers. People gradually started to realize that the well-established and expensive GPU architecture can be used in not only rendering images, but also computing which what CPU mainly does. And the efficiency could be way better than what a traditional CPU offers. That is when and why CUDA, a platform and programming model created by NVIDIA and implemented by GPU, was introduced to the world.

In this paper, we will first discuss the basic architecture of GPU, including the history of its development. After that, we will introduce CUDA in details, how it works and what compatibility does it have. Then finally we are going to cover how CUDA operates based on GPU Architecture.

Key Words: GPU, CUDA, CPU, Architecture, Computing, Parallel Programming, Relationship.

# 1. Introduction

## 1.1 History of CUDA

Beginning from the 90s, graphic handling has gradually been becoming common and common. However, due to the limited programmability towards the shading device, there have not been many kinds of graphical handling application at that time.

On March 2001, NVIDIA proposed its first programmable towards shading device GPU – GeForce3 (NV20). After that, ATI has also proposed Radeon 9700 which allows us to use pixels and corners in the shading device to perform basic loop operation and float calculation. On January 2001. NVIDIA has proposed the first generation of GPU architecture based on CUDA (Compute Unified Device Architecture) – G80. CUDA is designed based on the common calculation and operation, and under the condition of guaranteeing graphical handling ability, CUDA has been introduced with new parallel programming model and set of instructions which make GPU programming more flexible. In China, there is a list indicating the top 100 fastest super computer, among which the top 5 has the architecture of CPU+GPU architecture. (Doserv, 2012)

The first GPUs were designed as graphics accelerators, supporting only specific fixed-function pipelines. Starting in the late 1990s, the hardware became increasingly programmable, culminating in NVIDIA's first GPU in 1999. Less than a year after NVIDIA coined the term GPU, artists and game developers weren't the only ones doing ground-breaking work with the

technology: Researchers were tapping its excellent floating point performance. The General Purpose GPU (GPGPU) movement had dawned.

But GPGPU was far from easy back then, even for those who knew graphics programming languages such as OpenGL. Developers had to map scientific calculations onto problems that could be represented by triangles and polygons. GPGPU was practically off-limits to those who hadn't memorized the latest graphics APIs until a group of Stanford University researchers set out to reimagine the GPU as a "streaming processor." (GeForce 256.)

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. Using concepts such as streams, kernels and reduction operators, the Brook compiler and runtime system exposed the GPU as a general-purpose processor in a high-level language. Most importantly, Brook programs were not only easier to write than hand-tuned GPU code, they were seven times faster than similar existing code.

NVIDIA knew that blazingly fast hardware had to be coupled with intuitive software and hardware tools, and invited Ian Buck to join the company and start evolving a solution to seamlessly run C on the GPU. Putting the software and hardware together, NVIDIA unveiled CUDA in 2006, the world's first solution for general-computing on GPUs. (Geforce 3 Product Overview 06.01v1, 2012)

## 1.2 History of GPU

GPU, Graphic Processing Unit, is a device that works on personal computer, workstation, game device and mobile devices. It is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. (Lengyel, 1990)

The term GPU was popularized by Nvidia in 1999, who marketed the GeForce 256 as "the world's first 'GPU', or Graphics Processing Unit, a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that are capable of processing a minimum of 10 million polygons per second". Rival ATI Technologies coined the term visual processing unit or VPU with the release of the Radeon 9700 in 2002.

With the introduction of the GeForce 8 series, which was produced by Nvidia, and then new generic stream processing unit GPUs became a more generalized computing device.
Today, parallel GPUs have begun making computational inroads against the CPU, and a subfield of research, dubbed GPU Computing or GPGPU for *General Purpose Computing on GPU*, has found its way into fields as diverse as machine learning, oil exploration, scientific image processing, linear algebra, statistics,3D reconstruction and even stock options pricing determination. Nvidia's CUDA platform was the earliest widely adopted programming model for GPU computing. More recently OpenCL has become broadly supported. OpenCL is an open standard defined by the Khronos Group which allows for the development of code for both GPUs and CPUs with an emphasis on portability. (NVIDIA, 2012) OpenCL solutions are

supported by Intel, AMD, Nvidia, and ARM, and according to a recent report by Evan's Data, OpenCL is the GPGPU development platform most widely used by developers in both the US and Asia Pacific.

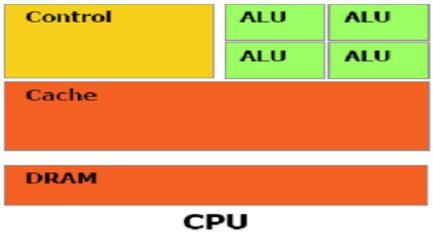
## 1.3 GPU Architecture

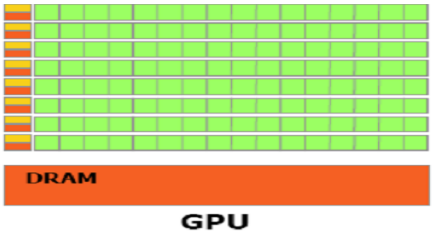
In the previous section, we have introduce what a GPU, and we are going to introduce the basic GPU architecture.

Why we need GPU instead of CPU sometimes? Because GPU can provide a separate dedicated graphics resource including a graphics processor and memory as well as provide a separate dedicated graphics resource including a graphics processor and memory.


That's a GPU manufactured by NVIDIA

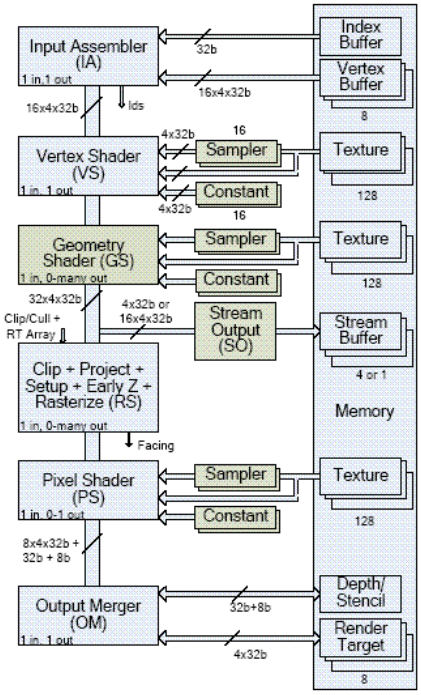
Due to the business interest driven, GPU aims to be faster than CPU. By the well-known Moore's Law which claims that over the history of computing hardware, the number of transistors in a dense integrated circuit doubles approximately every two years, GPU is faster not only it was simplified designed, but also because it was single-chip designed.




The Comparison Between GPU and CPU

The modern GPU architecture is a pipeline structure which consists six components, Input Assembler, Vertex Shader, Geometry Shader, Stream Output, Pixel Shader and Output Merger:




The functionality of the GPU Pipeline.

To Summarize:

GPUs use stream processing to achieve high throughput

GPUs designed to solve problems that tolerate high latencies

High latency tolerance -> Lower cache requirements

Less transistor area for cache -> More area for computing units

More computing units -> 10,000s of SIMD threads and high throughput

Additionally:

Threads managed by hardware ->You are not required to write code for each thread and manage them yourself

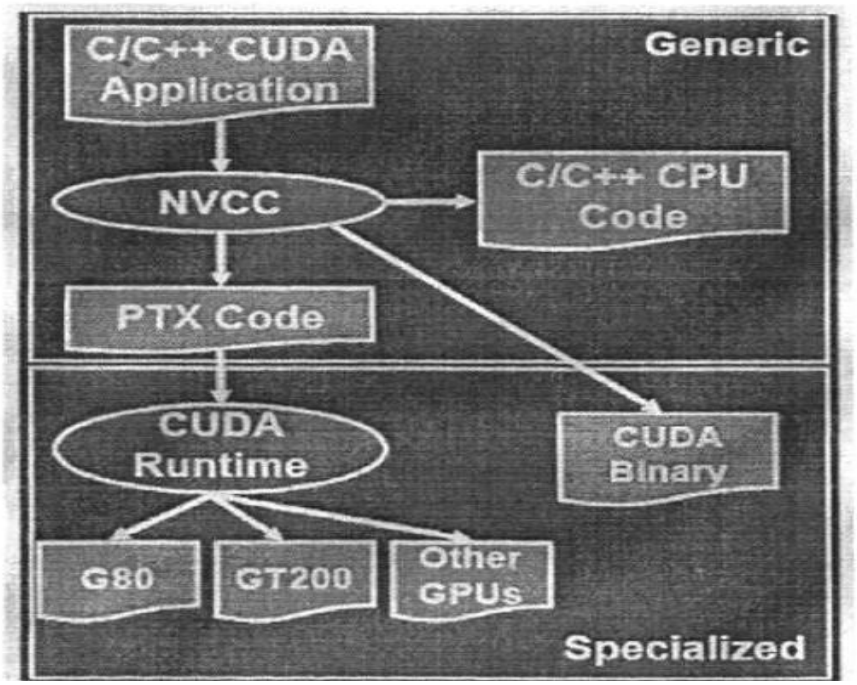
Easier to increase parallelism by adding more processors (PGI, CUDA Fortran Programming Guide and Reference, 2012)

## 1.4 Compute Unified Device Architecture (CUDA)

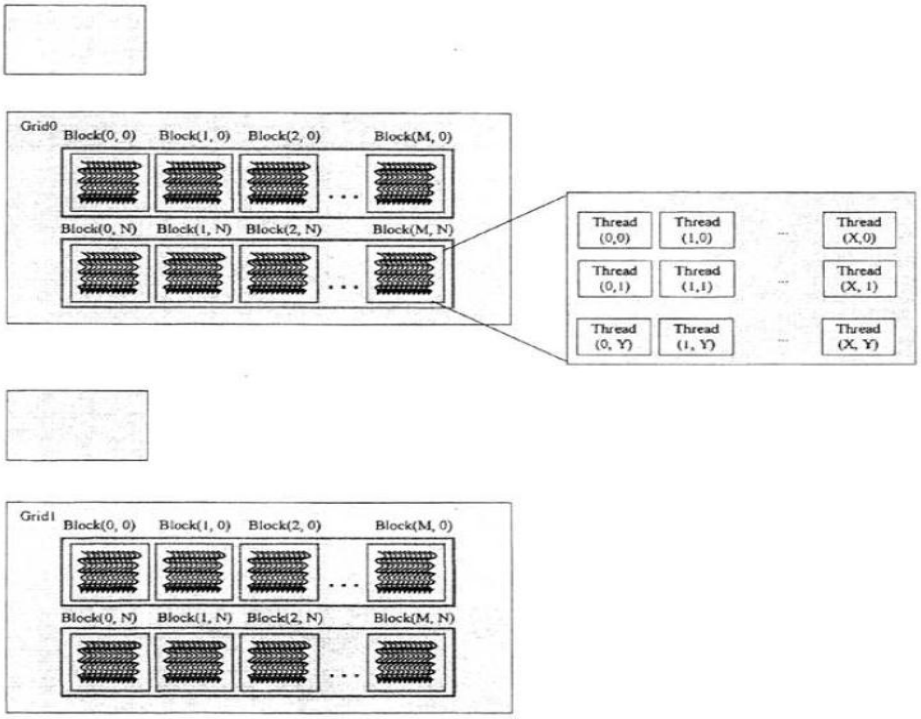
In the CUDA programming model, CPU serves as the host and GPU serves as the accessory processor or device, which is a kind of heterogeneous computing architecture. A system can have one host and multiple devices, they work together to separate the task. CPU is good at logic controlling, thus it mainly focuses on tasks that require strong logicality like concatenation

operation. GPU, on the other side, has a relative weaker ability to deal with the logicality, but has rich calculation resources as we explained in the previous section. Thus, GPU works on more highly-threading and parallel tasks. They have relative independent storing space: memory and video memory. CUDA uses different ways to control memory and video memory: For memory, it uses memory management function in C Language; for video memory, it uses CUDA API including assignment, initiation and frees of the video memory space as well communication and passing data between memory and video memory.

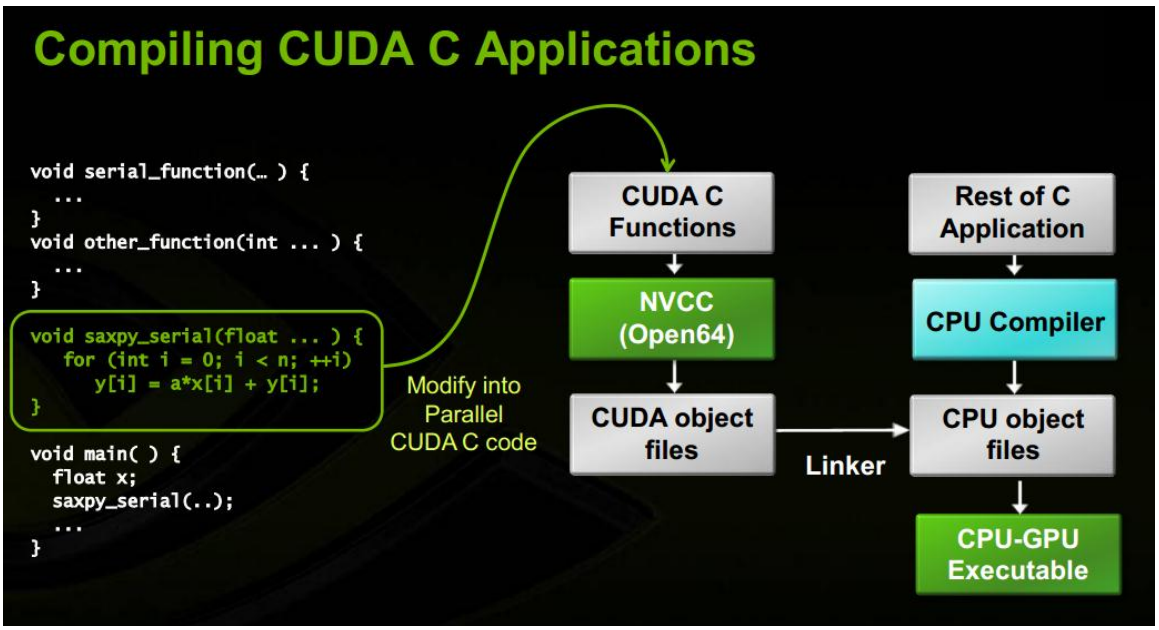
CUDA software system is composed with CUDA library functions, API and drivers. The current CUDA uses C Language as the programming language, and in the future, will add Fortran and C++. The compiling process of CUDA is shown as below. NVCC Compiler segments the source codes into host side which written by C and GPU PTX Code. Finally through PTX compiler (NVIDIA Corporation, 2007), the codes are compiled onto GPU (NVIDIA Corporation, 2008b).



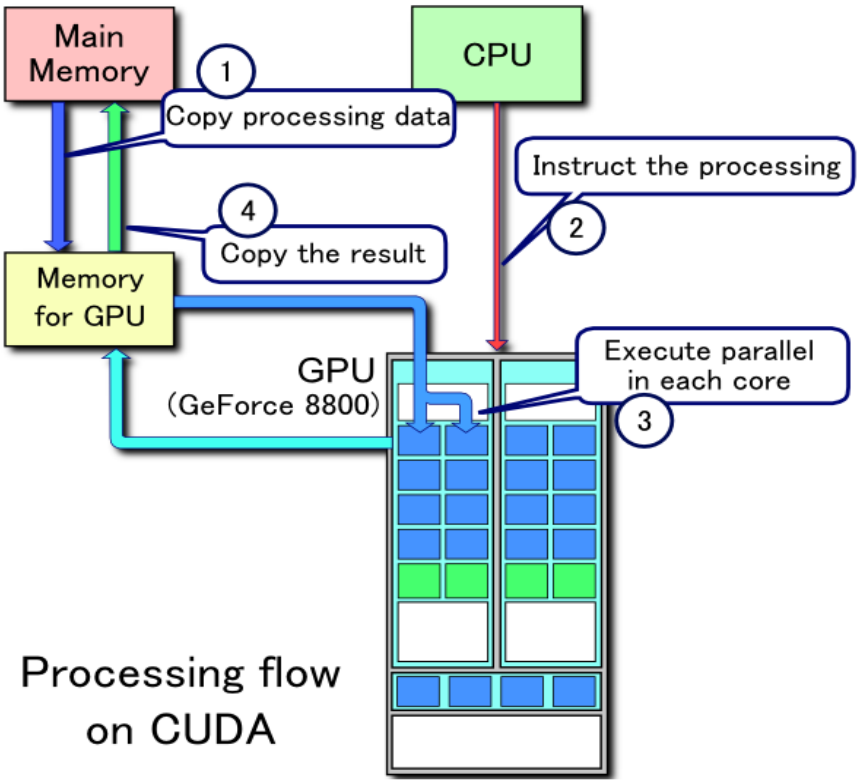

A complete CUDA program should consists two part: Host Program and Device Function Kernel. As the graph show below, the host program is executed by CPU, including data preparation, initialization before starting up kernel, data handling, and post-operation before the program expires. Under the ideal situation, due to the computability of GPU is much higher than the one of CPU, therefore, we hope to put more calculations on GPU, while CPU only needs to deal with some starting-up works for kernel in order to reduce the data transmission between memory and video memory. Unfortunately, the functionality of GPU is limited by now; stream controlling is still weak so that it can barely complete the task with complicated calculation, making the workload of CPU remaining large.




CUDA Program Execution Model



An Overview of CUDA Compiling Process




# 2. GPU Programming Under CUDA

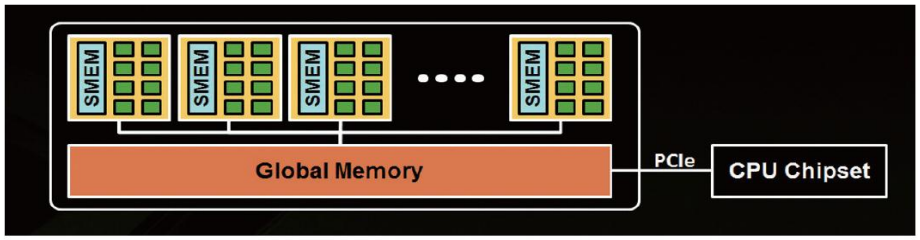
## 2.1 GPU Structure and GPU Programming Model

GPU is a processor that aims at optimizing throughput of parallel process, which causes the number of kernels of GPU is much more than the one of CPU. This structure is called multi-kernels structure. The throughput of Multi-kernels structure of GPU Architecture is higher than the one of CPU both on instructions and memory and is more capable of dealing with large

amount of parallel calculation target. (PGI, PGI CUDA-x86: CUDA Programming for Multi-core CPUs, 2010)

In the computer system structure, GPU helps CPU to calculate, deals with some tasks involved with multi-threads and large amount of throughput which CPU can barely finish on its own. GPU connects with main wire through PCI-E. The instructions that are executed by CPU need to pass the interface in order to pass into GPU.

The inner structure of GPU is very simple, it consists two parts: calculation part and the storing part. Calculation part is composed with stream processors (SM), which mainly focuses on GPU instructions execution, while the storing part is called the Global Memory, which takes care of storing of data.



GPU Structure Model

## 2.2 GPU Programming Methods

CUDA structured GPU has various programming categories. Some most common including CUDA C/C++, CUDA Fortran, OpenACC, HMPP, CUDA-x86, OpenCL, JCuda, PyCUDA, Direct Compute, Microsoft C++ AMP etc.

Among all list above, CUDA C/C++ is a set of programming technologies developed by NVIDIA, as well as NVIDIA's partnership. Regarding to the programming languages, C/C++ is supported by CUDA C/C++, CUDA Fortran, OpenACC, HMPP, CUDA-x86, OpenCL, JCuda, PyCUDA, Direct Compute, and Microsoft C++ AMP. Fortran is Supported by CUDA Fortran, OpenACC, HMPP. As for JAVA, JCuda is the API. PyCUDA is for Python.

Multi-threading programming languages can be classified into 2 types, one is compiling-oriented, which means adding compliable instructions before the concatenation program begins. The other one is threading-display model, which allows users directly code the parallel program and visually call GPU threads. Relatively, compiling-oriented one is easier. Both because it is easy for beginners to learn, and is convenient for concatenated program turns into a parallel as well as short developing lifecycle. However, because its compiler will automatically finish paralleling the contented program, thus, GPU operation is relatively less flexible than threading-display model.

In our introduction to programming techniques, OpenACC and HMPP are compiling-oriented languages, they are very similar to parallel programming language OpenMP based on CPU. CUDA Fortran supports both compiling-oriented and threading-display models, others are threading-displays models. However, CUDA-x86 is special, it is used to let CUDA C/C++ program directly compiled into CPU multi-threading program, without the need to revising codes. The following form shows every programming details.

**Table 1 GPU Programming Language**

| 编程技术 | 支持语言 | 开发者 | 分类 |
|---|---|---|---|
| CUDA C/C++ | C/C++ | NVIDIA | 显式线程模型 |
| CUDA Fortran | Fortran | PGI | 显式线程模型/编译指导 |
| OpenACC | C/C++ Fortran | PGI/CAPS/GREY/NVIDIA | 编译指导 |
| HMPP | C/C++ Fortran | CAPS | 编译指导 |
| CUDA-x86 | CUDA C/C++ | PGI | 显式线程模型 |
| OpenCL | C/C++ | Khronos | 显式线程模型 |
| Direct Compute | C/C++集成于Direct3D之中 | Microsoft | 显式线程模型 |
| Microsoft C++ AMP | C/C++ | Microsoft | 显式线程模型 |
| JCuda | JAVA | Personal | 显式线程模型 |
| PyCUDA | Python | Personal | 显式线程模型 |

### 2.2.1 CUDA C/C++

#### (1) Characteristic Introduction

CUDA C/C++ is developed by NVIDIA by the year of 2006 in order to program on GPU, now it has already announced the fifth version CUDA 5.0. It is a simple extension of C/C++. Those who are familiar with C/C++ languages should be able to get used to it real fast.

CUDA C/C++ contains libraries from every realm in order to convenience the developer directly invoke. In CUDA 5.0, it contains calculating fast Fourier Transform Library cuFFT, calculating linear algebra library cuBLAS, calculating Sparse Matrix library cdSPARSE, generating random number library cuRAND, handling images signal and video library NPP, handling sorting, scanning, transferring library Thrust, and standard math supporting library. (PGI, PGI CUDA-x86: CUDA Programming for Multi-core CPUs, 2010)

CUDA C/C ++ support for Windows, Linux, Mac Three Platform, and supports Visual Studio, Eclipse, and other integration Development environment. Developers can use the Windows

platform Nsight Tools for debugging, and the developer of the Linux platform can using CUDA-GDB for debugging, using similar methods and gdb. Also in the major platform also provides memory checking tool cudamemcheck check the memory errors.

CUDA C / C ++ support for Windows, Linux, Mac Three Platform, and supports Visual Studio, Eclipse, and other integration development environment. Developers can use the Windows platform Nsight Tools for debugging, and the developer of the Linux platform can Using CUDA-GDB for debugging, using similar methods and gdb. Also in the major platform also provides memory checking tool cudamemcheck Check the memory errors.

Developers can download from the official website of NVIDIA CUDA 5.0 Development Kit begin development work. Kit contains CUDA C/C ++ compiler NVCC, analyzer, programming manuals, GPU-driven, and for each typical example of application design program.

### (2) Programming Technique

Different from other programming techniques, the typical CUDA C/C++ also has two parts. One is GPU-CPU memory transmission, the other part is GPU inner parallel algorithm. The following is a simple CUDA C/C++ program:

```
__global__ void vec_add(float *x,float *y,int n)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  y[i]=x[i]+y[i];
}

int main(){
  float *d_x,*d_y;
  cudaMalloc(&d_x,n*sizeof(float));
  cudaMalloc(&d_y,n*sizeof(float));
  cudaMemcpy(d_x,x,n*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_y,y,n*sizeof(float),cudaMemcpyHostToDevice);
  vec_add<<<n/128,128>>>(d_x,d_y,n);
  cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
  cudaFree(d_x);
  cudaFree(d_y);
}
```

CUDA C/C++ program is similar to the CPU C/C++ which we are quite familiar with. From the example above, there are two functions, one is the entrance function main, the other is GPU function vec_add, using __global__ to label. In the main function, the program will first use cudaMalloc function to give necessary memory to GPU, and pass the memory address to pointer by using cudaMemcpy functions to pass data from CPU to GPU. After that, the program invokes vec_add function to distribute threads for the GPU calculation. Finally, it runs the function cudaMemcpy to pass the result from GPU to CPU.

GPU function vec_add consists two instructions. The first one is used through thread ID and process Id calculation, which makes sure that every ID of every thread is globally unique. The other one is used to sum an objected generated by calculations of all threads.

### 2.2.2 CUDA Fortran

#### (1) Characteristic Introduction

NVIDIA CUDA Fortran is a partner of the PGI Fortran users for the launch of a programming language. The latest version of Release 2012. It contains a Fortran2003 compiler. CUDA Fortran And CUDA C / C ++ programming model is very similar to the familiar with Fortran or CUDA C developers can quickly become familiar with the operation.

With CUDA C / C ++, like CUDA Fortran, etc. You can also access cuBLAS CUDA library. And full support for LINUX, MacOS X, Windows three platforms to support Visual Studio (Reference PVF) and Eclipse (need to install the plug-in). CUDA Fortran does not directly support Debug, you can use the compiler options open -Mcuda = emu emulator mode, CPU simulation using GPU, and CPU utilization for debugging Debugger.

You need to download the PGI CUDA Fortran development kit on the official website of PGI CUDA Fortran programming prior to the current version of the "PGI Release 2012". Kit contains: CUDA Fortran compiler pgfortran, OpenACC compiler pgcc, CUDA-x86 compiler pgcpp, analyzer pgprof, debugger pgdbg, instructions and other contents of the document.

#### (2) Programming Technique

CUDA Fortran has really good compatibility of putting GPU programming into Fortran. Those who are familiar with CUDA C/C++ can easily pick up CUDA Fortran after learning Fortran

basic programming. For FORTRAN users, they also only need to supplement themselves with basic knowledge of heterogeneous calculation.

The following is example of CUDA FORTRAN program:

```
module cuda_kernels
contains
attributes(global) subroutine vec_add(y,x,n)
implicit none
integer, value :: n
integer :: y(n)
integer :: x(n)
integer :: idx
idx = (blockIdx%x-1)*blockDim%x + threadIdx%x
if (idx .le. n) y(idx) = y(idx) + x(idx)
end subroutine kernel
end module cuda_kernels
program main
use cuda_kernels
use cudafor
implicit none
integer, parameter :: dimx = 16
integer, allocatable :: h_y(:)
integer, allocatable, device :: d_y(:)
integer, allocatable, device :: d_x(:)
type(Dim3) grid,block
allocate(h_y(dimx))
allocate(d_y(dimx))
allocate(d_x(dimx))
block=dim3(4,1,1)
grid=dim3(dimx/block%x,1,1)
call vec_add<<<grid,block>>>(d_y,d_x,dimx)
h_y = d_y
deallocate(h_y)
deallocate(d_y)
deallocate(d_x)
end program main
```

This example realizes the same functionality of the CUDA C/C++ as we showed in the previous section. Still, there are several points we need to pay attention to: First, in the CUDA Fortran, we add attribute (global) before subroutine to label that is subprogram is calculated in GPU. After that, we add the word “device” before the declaration of variables meaning that those variables are distributed in the GPU video memory. In GPU, the memory assignment function is totally the

same as the one in CPU, it is either allocate or deallocate. In the end, if data needs to transmitted between CPU and GPU, one only needs to use "=" to operate, without any special function.

### 2.2.3 OpenACC

#### (1) Characteristic Introduction

OpenACC is a set of open standards several partners CRAY, PGI, CAPS and co-developed NVIDIA. And the first two different programming techniques, OpenACC is a compiled language guide, now supports C / C ++, Fortran. It can specify the period of the cycle or the code block is transferred to the GPU for parallel execution. Grammar rules and OpenMP are very similar.

Relatively, the CUDA C/C ++ with CUDA Fortran, OpenACC are significantly more suitable for parallel programs. It has accelerated the development, short learning curve, allowing developers to focus on the development of CPU code, no need to consider the advantages of GPU implementation. The disadvantage is not easy to more sophisticated programming. To this end the company's CAPS OpenACC products increased OpenACC to CUDA C / C ++ code conversion function. Users can generate code to CUDA C / C ++ program implemented by OpenACC simply ported to GPU, and then for CUDA C / C ++ code that takes a long fragment finer optimization.

Conduct OpenACC program development, Linux users can download the official website CAPS OpenACC 1.0 or HMPP3.2 (HMPP now fully supports OpenACC), then programming. Or PGI

PGI official website development kit, the current version is "PGI Release 2012", which already contains OpenACC compiler pgcc.

(2) Programming Technique

```
__global__ void vec_add(float *x,float *y,int n)
{

#pragma acc kernels
  for(int i=0; i<n; ++i)
    y[i]=x[i]+y[i];
}

int main(){
  float *x=(float*) malloc(n*sizeof(float));
  float *y=(float*) malloc(n*sizeof(float));
  vec_add(x,y,n);
  free(x);
  free(y);
}
```

An OpenACC Example

Here, we still use an example sum of vector to illustrate the OpenACC Programming.

In this example, we can see that this program only has one statement different from the CPU vector and thread program which we are quite familiar with. We call it compiler-oriented. It can automatically distribute n thread to GPU and copy to data from CPU side to GPU for calculation, and after the calculation, over copy the data back to CPU from GPU. These operation are done automatically by complier.

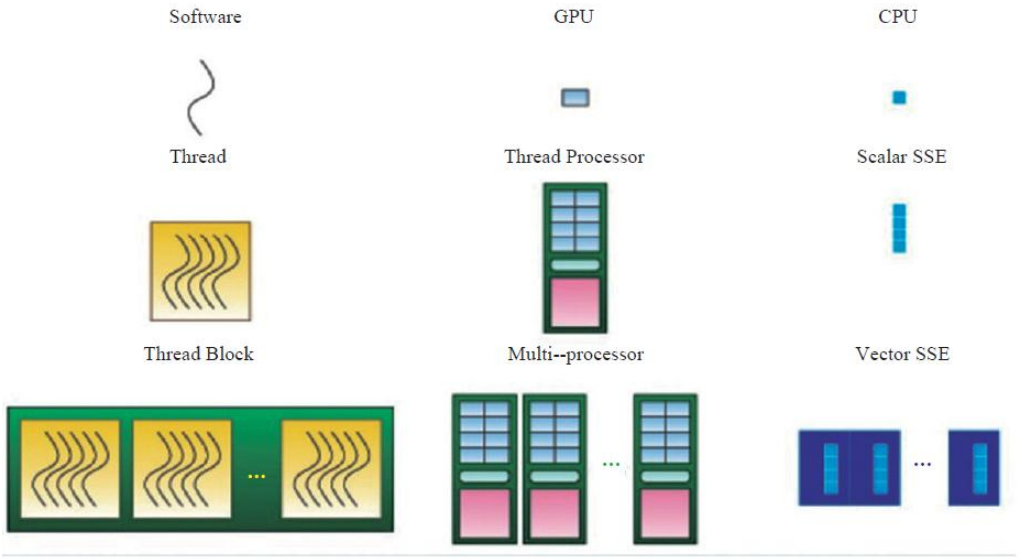
### 2.2.4 CUDA-x86

(1) Characteristic Introduction

CUDA-x86 is proposed and developed by PGI Cooperation for the purpose of running the same block of code which is run both on GPU and multi-thread CPU at the same time. OpenACC is

realized through compiling CPU code into GPU code and work on GPU. In contrast, CUDA-x86 is realized through compiling GPU code and executes them on multi-thread CPU.

GPU and CPU architecture are actually very similar. In the CPU core is between multiple parallel MIMD architecture, usually 4 to 12 cores. In each of the internal CPU core are parallel SIMD vector processing unit, for processing or SSE AVX instruction. Thus, we can be in the GPU SM corresponding to the CPU of each core, and should check each CUDA Scalar SSE to each processing unit.




**GPU Architecture Mapping to CPU in CUDA-x86**

Such CUDA-x86 CUDA compiler actually mapped to the core of the blocks on the CPU, and the thread-level parallelism CUDA to SSE or AVX SIMD processing unit and CPU use optimization

methods to optimize the code to achieve piece of code can be compiled into a purpose in accordance with user requirements GPU and CPU executable.

CUDA-x86 also supports Windows, Linux, Mac OSX three platforms. Users can simply by PGI Release pgcpp compilation tool in the existing CUDA code is compiled to run on top of a multi-core CPU. The method of operation and general CPU code also no difference. (Tesla, 2012)

# 3. Summary

Beginning from the year of 2001 to now, people are becoming more familiar with GPU common calculation. More and more developers are doing research and study in the related fields and gained benefits from it. On the other hand, as the number of developers grows, the specific programming methods are being proposed. However, it does not necessary we should be stick to GPU Programming all the time without any condition. In this section, we are going to show advantages and disadvantages of CUDA programming.

## 3.1 Advantages of CUDA

CUDA has several advantages over traditional general-purpose computation on GPUs (GPGPU) using graphics APIs:

1. It is scattered reads, which means the codes can be read from arbitrary address in memory. And you can also store your data freely in any memory. Since there are functions like malloc in CUDA C/C++. (PGI C. C., 2012)
2. When it comes to CUDA 4.0 and above, it supports unified virtual memory.
3. When it comes to CUDA 6.0 and above, it supports unified memory.
4. CUDA exposes a fast shared memory region (up to 48 KB per multi-processor) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups. This is so called the shared memory.

5. CUDA Program has faster downloads speed and readbacks to and from the GPU.
6. It has full support for integer and bitwise operations, including integer texture lookups. This convenience many developers since many of them do not need to pick up a new way of declaring large amount of variables which are not quite different than the original languages they used to write.

## 3.2 Limitations

1. CUDA does not support the full C standard, as it runs host code through a C++ compiler, which makes some valid C (but invalid C++) code fail to compile.
2. Interoperability with rendering languages such as OpenGL is one-way, with OpenGL having access to registered CUDA memory but CUDA not having access to OpenGL memory.
3. Copying between host and device memory may incur a performance hit due to system bus bandwidth and latency (this can be partly alleviated with asynchronous memory transfers, handled by the GPU's DMA engine).
4. Threads should be running in groups of at least 32 for best performance, with total number of threads numbering in the thousands. Branches in the program code do not affect performance significantly, provided that each of 32 threads takes the same execution path;
   the SIMD execution model becomes a significant limitation for any inherently divergent task (e.g. traversing a space partitioning data structure during ray tracing).
5. Unlike OpenCL, CUDA-enabled GPUs are only available from Nvidia.
6. No emulator or fallback functionality is available for modern revisions/

7. Valid C/C++ may sometimes be flagged and prevent compilation due to optimization techniques the compiler is required to employ to use limited resources.
8. A single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments.
9. C++ Run-Time Type Information (RTTI) is not supported in CUDA code, due to lack of support in the underlying hardware.
10. Exception handling is not supported in CUDA code due to performance overhead that would be incurred with many thousands of parallel threads running.
11. CUDA (with compute capability 2.x) allows a subset of C++ class functionality, for example member functions may not be virtual (this restriction will be removed in some future release). *[See CUDA C Programming Guide 3.1 – Appendix D.6]*
12. In single precision on first generation CUDA compute capability 1.x devices, denormal numbers are not supported and are instead flushed to zero, and the precisions of the division and square root operations are slightly lower than IEEE 754-compliant single precision math. Devices that support compute capability 2.0 and above support denormal numbers, and the division and square root operations are IEEE 754 compliant by default. However, users can obtain the previous faster gaming-grade math of compute capability 1.x devices if desired by setting compiler flags to disable accurate divisions, disable accurate square roots, and enable flushing denormal numbers to zero.

# References


Geforce 3 Product Overview 06.01v1. (2012/10). Soruce: http://www.nvidia.com/page/geforce3.html

Doserv. (2012/OCT/29th). 从 Top100 看 HPC 发展趋势. Source: HPC China: http://www.doserv.com/article/2012/1029/1210358.shtml

GeForce 256. Source: http://en.wikipedia.org/wiki/GeForce_256 .

LengyelReichert, M., Donald, B.R., Greenberg, D.P.J.,. (1990). Real-Time Robot Motion Planning Using Rasterizing. SIGGRAPH.

NVIDIA. (2012). GPU-Accelerated applications.

PGI. (2010). PGI CUDA-x86: CUDA Programming for Multi-core CPUs.

PGI. (2012). CUDA Fortran Programming Guide and Reference.

PGICRAY NVIDIACAPS. (2012). The OpenACC API QUICK REFERENCE GUIDE Version.

Tesla. (2012). C2050/C2070 GPU Computing Processor Supercomputing at 1/10th the Cost, NVIDIA,.