Introduce to GPU Computing

Shuo Peng
GWID: 36877532
Email: friday@gwu.edu
CS 6461 11
March 15, 2015
The George Washington University

Content

Abstract

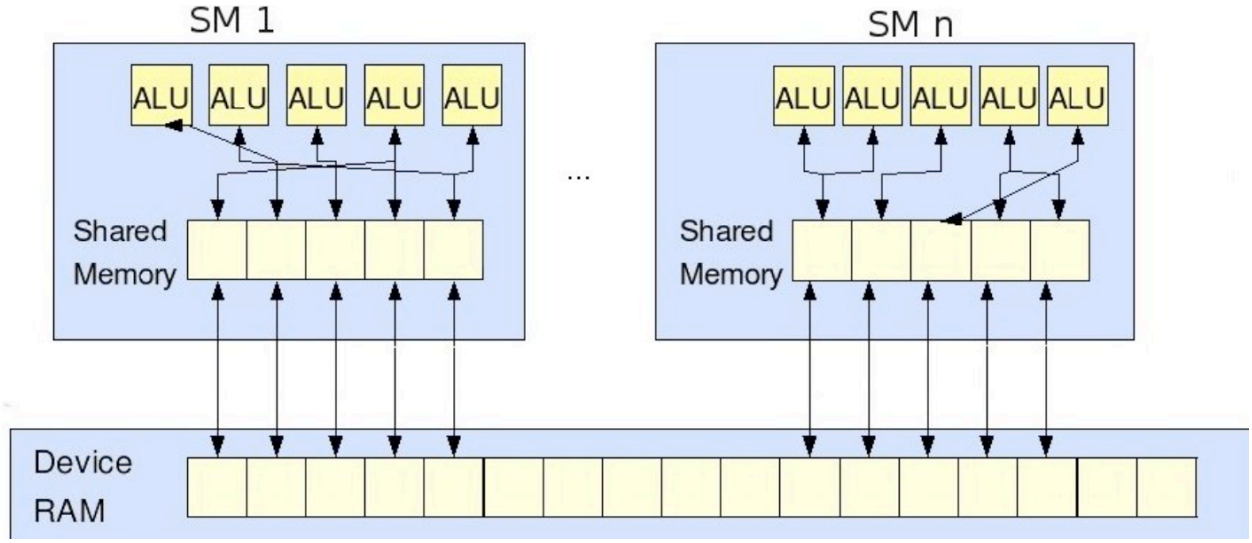
GPU (Graphic Processing Unit) is the processor featuring powerful processing of graphics. Over the years, GPU has been an exceedingly increased in its performance and capabilities. It is not only a graphic engine, but also the tool we use to do parallel computing and float computing, which is known as GPU computing. Parallel computing means to solve the computing problem by simultaneously using various resources; hence it can improve the performance dramatically. Some experts contend that the GPU is a compelling alternative to the traditional microprocessors in high-performance computer systems of the future. Since 2006, NVIDIA launched its programmable GPU; witch meant the era of GPU programming was coming. The most important GPU programming languages are CUDAs: CUDA C/C++, CUDA Fortran, CUDA x-86 and OpenACC. These languages make GPGPU more easily and for every developer.

GPU

General Architecture of GPU

Generally, GPU has a powerful and flexible architecture, which is able to do rendering works. It includes a large number of arithmetic logic units (ALU) in it, which replaced the separate vertex and pixel processor. ALU supports the whole float-computing, integer and byte operations. It strengthens the general computing ability of GPU a lot. There are two main factors about the GPU: one is about multi-level paralleled computing; another is about efficient data

access. There is a single instruction multiple data (SIMD), which can be used for paralleled computing.



GPU architecture has been developing since years. Take the NVIDIA Tesla GT200 architecture for example. Tesla GT200 has two parts. One is Scalable Streaming Processor Array (SPA). Another is memory system. Scalable Streaming Processor consists lots of Thread Processing Cluster (TPC). Each TPC contains 2 or 3 Streaming Multiprocessor (SM). Streaming Multiprocessors have independent cache and instruction points. From the architecture, you can see each Streaming Multiprocessors as a Single Instruction Multiple Data (SIMD) processor. The difference is GPU makes the direction possible on it. That is what NVIDIA calls it Single Instruction Multiple Thread (SIMT).

Tesla GT200 was the first remarkable GPU ever. It makes the GPU programmable, and finds a balance point between flexibility and complicity. Threads are allocated to processor via Thread Block. Then each Thread Block is transferred as single thread to processor. It makes developers can program on a GPU. That is a important step not only on the history of GPU, but on the history of computer.

The Development of GPU

In 80s', Geometry Engine was launched. That was the prototype of Graphics Processing Unit. Although computer scientists would not call it GPU, it has a huge impact on the graphics development and industry. The kernel function of GE is geometric changing, cutting, projecting and so on. One of its designers Clark James established SGI Company and released graphic programming language Graphics Language. That was becoming the standard of graphic industry: OpenGL. Another graphic system was Pixel Planes designed by N.C University. Pixel Planes was the most powerful graphic system in that time. It can handle 1 million graphics per second, and had SMID architecture.

With the development of graphics, most of the tasks of graphics transferred from CPU to GPU. In that time, personal computers were more and more popular. Powerful graphic system was also embedded in PC from stations. In 1998, the first GPU was launched by NVIDIA. All of GPUs (NVIDIA TNT2, ATI Rage and 3Dfx Voodoo3) were independent from CPU. By 2000, the performance of GPU was so powerful that can handle Transformation and Lighting algorithms.

In 2001, GPU was programmable. This is hallmark of the history in GPU. Meanwhile, OpenGL and DirectX supported vertex rendering programming interface. In the end of 2003, NVIDIA GeForce FX and ATI Radeon 9700 supported pixel-level programming. All of the efforts given by OpenGL and DirectX make the general computing possible. That is what we called GPGPU (General Purpose GPU).

GPGPU have these features.

1) It provides vertex and pixel levels flexible
2) It supports IEEE 32bit float computing

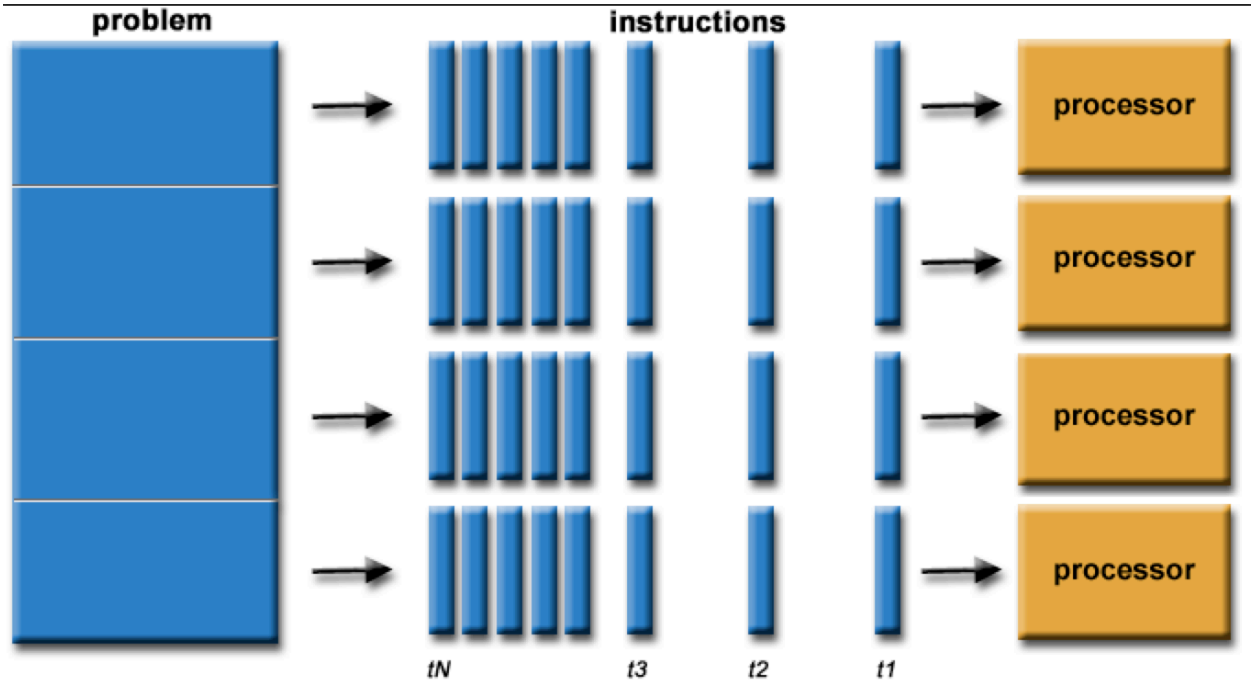
3) It supports loop control syntax

4) It supports 4-direction vector.

5) It has powerful bandwidth and fast transportation (more than 27.1GB/s)

6) It supports Render to Texture (RTT)

2006 is the most important year for GPU. In the end of 2006, NVIDIA launched its GeForce 8800GTX. It was not designed the traditional vertex and pixel rendering architecture. Instead of that, it used uniform rendering. The more important is it was the first GPU hardware that has special architecture that designed for general computing. It was not only for graphics, but also supported float computing, logic computing, thread communication, etc. For then, GPU is no longer a graphics processor, but a paralleled processor. That indicates the era of GPU computing.



Parallel Computing in GPU

Parallel Computing is one kind of computing that two or more threads in a computing task can be done at the simultaneously. It means data and instructions are able to be run at multi-processor at a unit time.

Parallel Computing comes from the needs of reality. Precise weather forecast in a large scale, oil and gas development, gene analyzing, environment simulation and other important science fields, all of these cannot be done with general computing. In order to enhance our computer to be capable with such sophisticated computing, we have to use parallel computing and the computers can do that.

Graphics Processing Unit (GPU) is designed for handling graphics, such as geometrics transformation, light, cutting or rendering. GPU makes computer to process graphics in a fast speed, and good quality. Unlike the Central Processing Unit (CPU), GPU was designed with parallel architecture. The GPU general computing means to do general calculation on GPU, not only graphics.

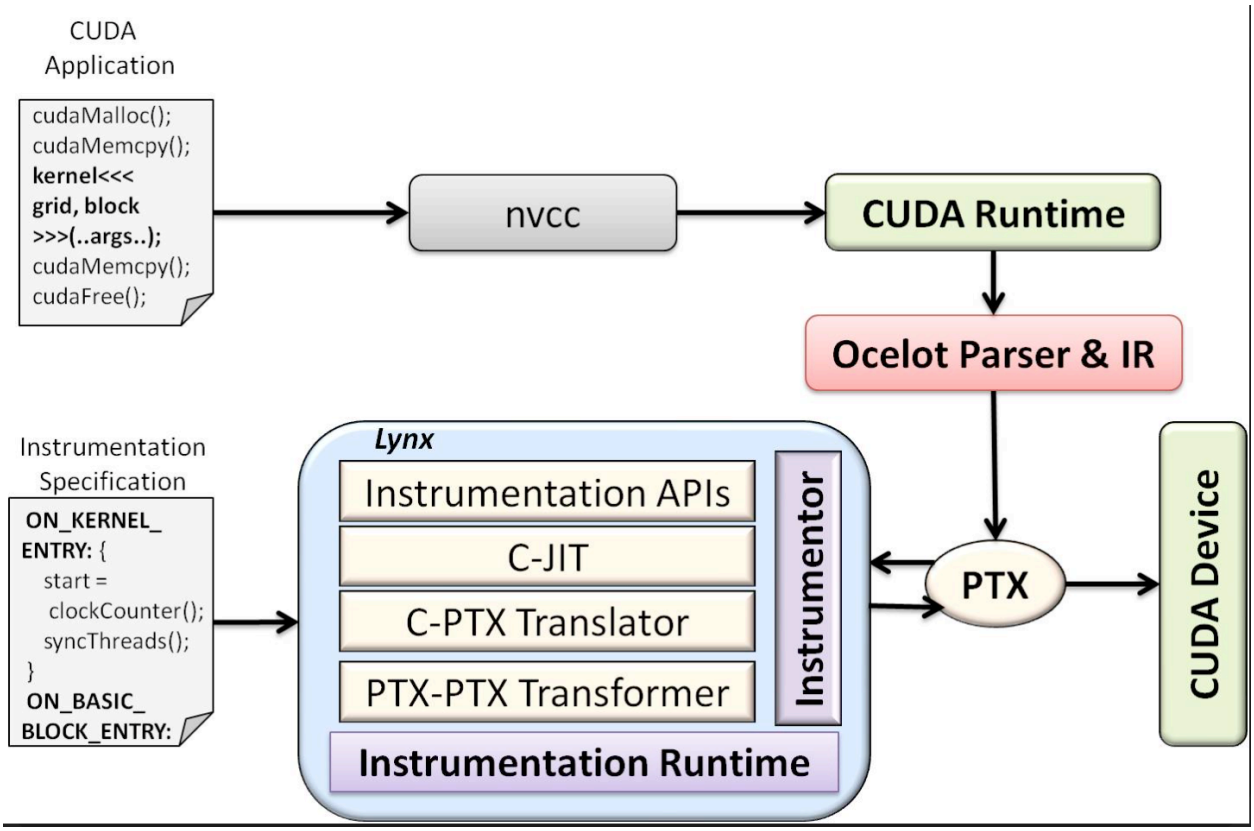
GPGPU

Concepts of GPGPU

When people found that GPU could do general computing besides graphics, people focus on it a lot. People from every field found that here are rooms for development of GPU. Every developer can program on it. GPU is like a platform for them, it makes them could do kinds of

computing on it. So, the first problem is how to do that, and how's the result. In 2003, experts and scientists who work on GPU focus on general purpose over graphic purpose. They made efforts on it. After that, IEEE, SUPERCOMUTING and other institution also made research on GPGPU.

Due to GPU's original purpose, which is focus on graphics, scientists designed an idea that the general computing should come with graphics calculating. That sounds very reasonable. GPU is to do what CPU cannot do, and make it visional. Some famous algorithm came up in that time. Such as computing with cutting, using vertex or rendering. All of this have wonderful result, some algorithm is 20 times faster than it on CPU.



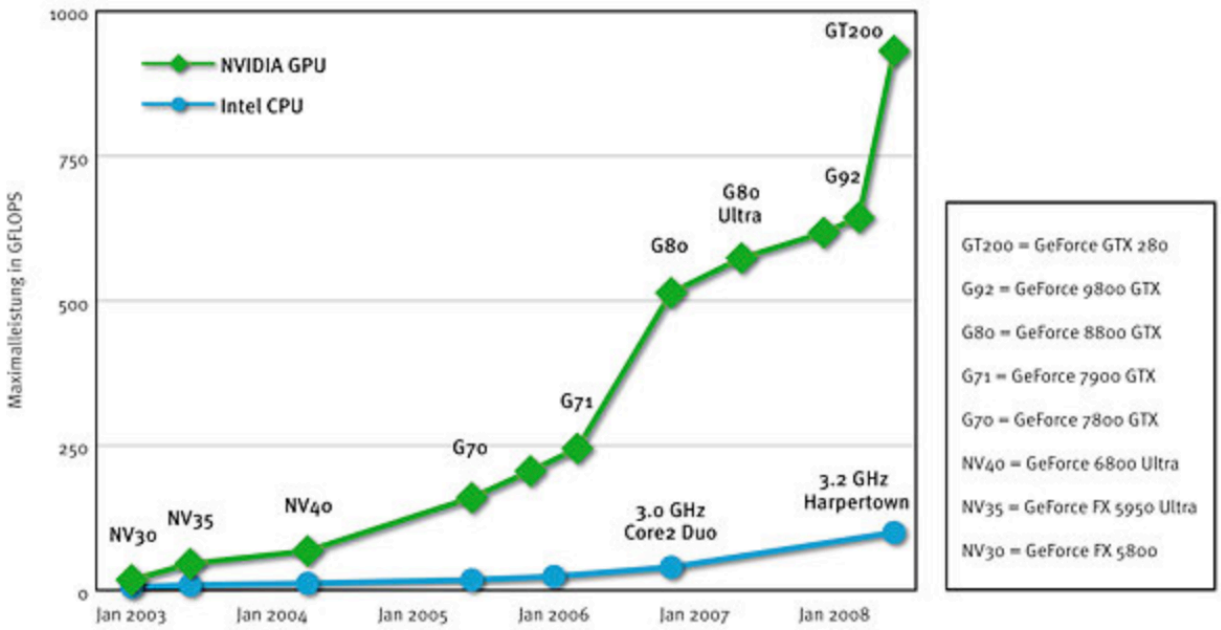
Languages of GPGPU

Using high-level languages to code GPU is a goal for GPGPU developers. That desire sounds reasonable, because low-level languages cannot be popular. Just like programming languages in CPU, low-lever languages like assembly language need programmer know a lot about hardware. High-level languages are more easily to write. With the efforts of academy and graphic industries, some high-level languages come up: OpenGL and shading language, RTSL (real-time shading language), Microsoft HLSL and NVIDIA Cg. General computing wraps data to texture and makes algorithms to the process of texture. They use graphic languages to write render program, then use API to execute it. This method made general computing in GPU to a new level. We do not need to learn the concepts of computer graphics and API. However, we still need OpenGL and DirectX to use it for us. The programmers who want to run program on GPU have to learn the interface of GPU, and have knowledge of computer hardware. The languages based on graphics are a kind of tentative effort in GPGPU. On one hand, there is no supporting for precision, data form, functions. On the other hand, there is no software for that platform, because they cannot get the result they want.

In 2003, Ian Buck and his Stanford colleague designed a complier named Brook based on Cg GPU by developing ANSI C. Brook can transfer Brook C (C-like language) to Cg code by compiler. It hidden the bunch of details of graphics API, and made the programming much easier. However, the early version of Brook had a low efficiency of compiling, and can only use pixel render to do computing. Bound by the limit of GPU architecture, Brook had no efficient data accessing. Stream, which launched by AMD, had a better version of Brook, Brook +, in it. Brook + is much better than Brook, it has a different logic from Brook. However, it still lacked of data communicating.

CPU and GPU

The difference of architectures



CPU has a high development since the first personal computer. It has firm architecture and designed logic. What people could do to promote it is make CPU more concentrated by putting more transistors on it.

CPU just stays in an instruction parallel level. The main method to prompt its performance is to increase its CPU GHz. However, by the limits of manufacture, CPU has been increasing in GHz slowly. Instead of working on CPU GHz, INTEL and AMD focus on the CPU architecture. They make the development of CPU on a multi-core way. Since 2005, the first dual-core was launched, triple-core processor and quad-core processors have also released to the world.

It is easy to say that the multi-core CPU makes CPU possible to handle multi-thread program. Usually, CPU just can handle one thread of program. Now multi-thread can be run on multi-core CPUs. It seems like more cores more efficiency. That is why people tend to buy multiple core computers.

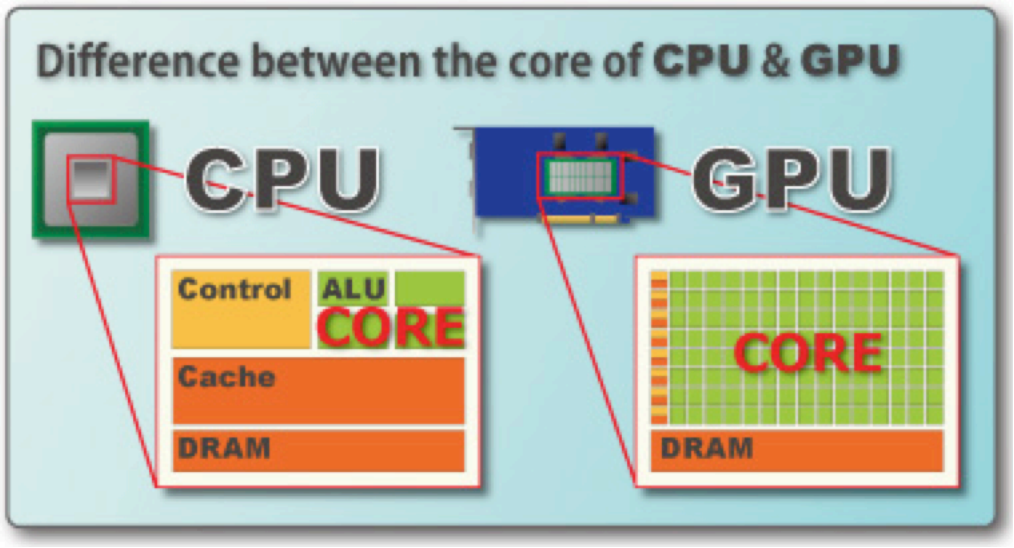
However, the result is not that linear. CPU and GPU have different architectures and different performances, because they designed for different purposes. Multiple cores CPU does not mean more efficient or faster.

The reason of that is about context, which is an environment that a thread happened. A context contains variables and address of particular program in a particular second. Threads are very important in CPU, when one thread was cut or waited for the resources; operating system has to save the context of those threads, and to load other thread. That makes CPU spend a lot of clock time to change the thread. That means more cores more time. GPU has not that problem. GPU does not need save the context or load it. When one thread in GPU was cut or waited, GPU can exchange it immediately. GPU can handle multi-thread with little delay.

In the architecture of a CPU, each single core can process its own instruction without anything about other core. The main core of CPU is Multiple Instruction Multiple Data (MIMD), which could make the cores to process. However, this methods limit the area of a CPU in a limited number of cores.

In GPU, the Streaming Processors (SP) act like cores in CPU. Each SP handles its own instructions. Despite the low frequency of processing, the number of SP in GPU is much more than the number of core in CPU, that makes GPU handle computing better than CPU. This architecture is what we called “Multiple Cores”. That is way GPU is the one that is more suitable for multiple thread programming, not CPU.

The difference of memory



So far, the memory controller of CPU base on two-channel technology and three-channel technology. Each channel has 64-bit bandwidth. That channel is what a CPU transfer data with memory. 64-bit bandwidth is double better than 32-bit bandwidth. The first 64-bit bandwidth smartphone is iphone5s. That indicates smart phone entry a 64-bit ear. On the other hand, GPU have lots of memory controller. All of these controllers have the ability of data accessing. That makes it has more than 512-bit bandwidth. In that way, CPU is 5 times powerful than CPU in memory access.

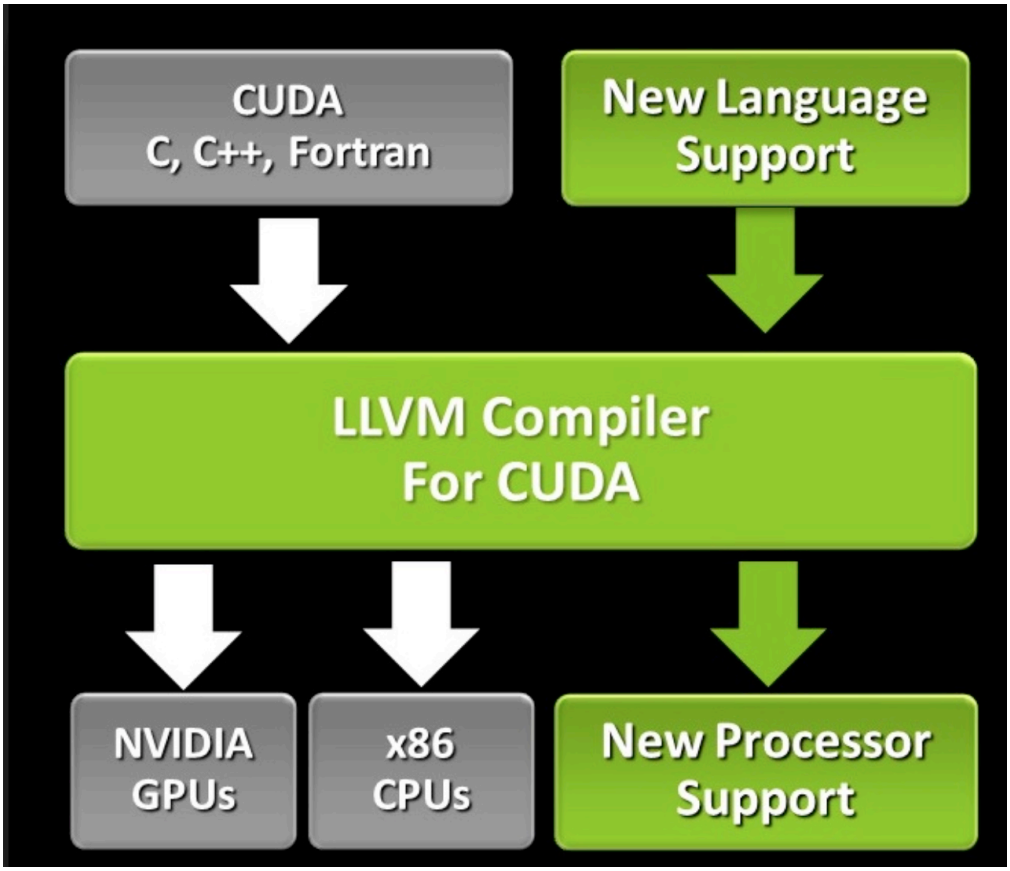
CPU has big catch in order to accessing delay and save some bandwidth. However, catch could be lapse in multiple thread situations. Every change of threads needs to build the context. Once the catch lapse, there would be a huge delay. Moreover, CPU has to make sure that the data in catch and data in memory are same. That needs complicated logic controlling. That catch technology results in low performance of CPU especially the ones have multi-core. However, there is no such catch technology in GPU, the catch in GPU has different meaning.

CUDA

The history of CUDA GPU

Since the 1990s, the graphics processor has been used for general-purpose computing. However, the number of applications based on graphics processor is small, because the renderer cannot be programmed. In March 2001, NVIDIA launched the first GPU with renderer which can ben programmed – GeForce3 (NV20). Late of the same year, ATI has launched renderer-programmable GPU – Radeon 9700. In that way, we can take basic loop and floating computing in dint of the pixel and vertex shades of GPU. January 2006, NIVDIA launched its first GPU based on CUDA architecture – G80. CUDA architecture is designed for general-purpose computing. It introduced a new parallel programming model and instruction sets, which made GPU programming more flexible without impacting the performance of graphics processing. Since 2006, more and more projects using the GPU general computing technology (GPGPU). In the top five supercomputers of last year, four of them adopted the CPU + GPU architecture.

CUDA is important in GPU history and in computer history. It indicates that common developers can program on GPU. Programmers do not have to know a lot of about graphics and hardware, before that they have to be experts if they want to program on GPU. The hardware of CUDA is more suitable for general computing as well. It has precision and efficiency. CUDA makes computer develop further and faster. It also makes GPU programming for everyone. Without CUDA, GPU was still considered much less important than CPU. CUDA changes people's opinion and makes them believe that GPU is as powerful as CPU.

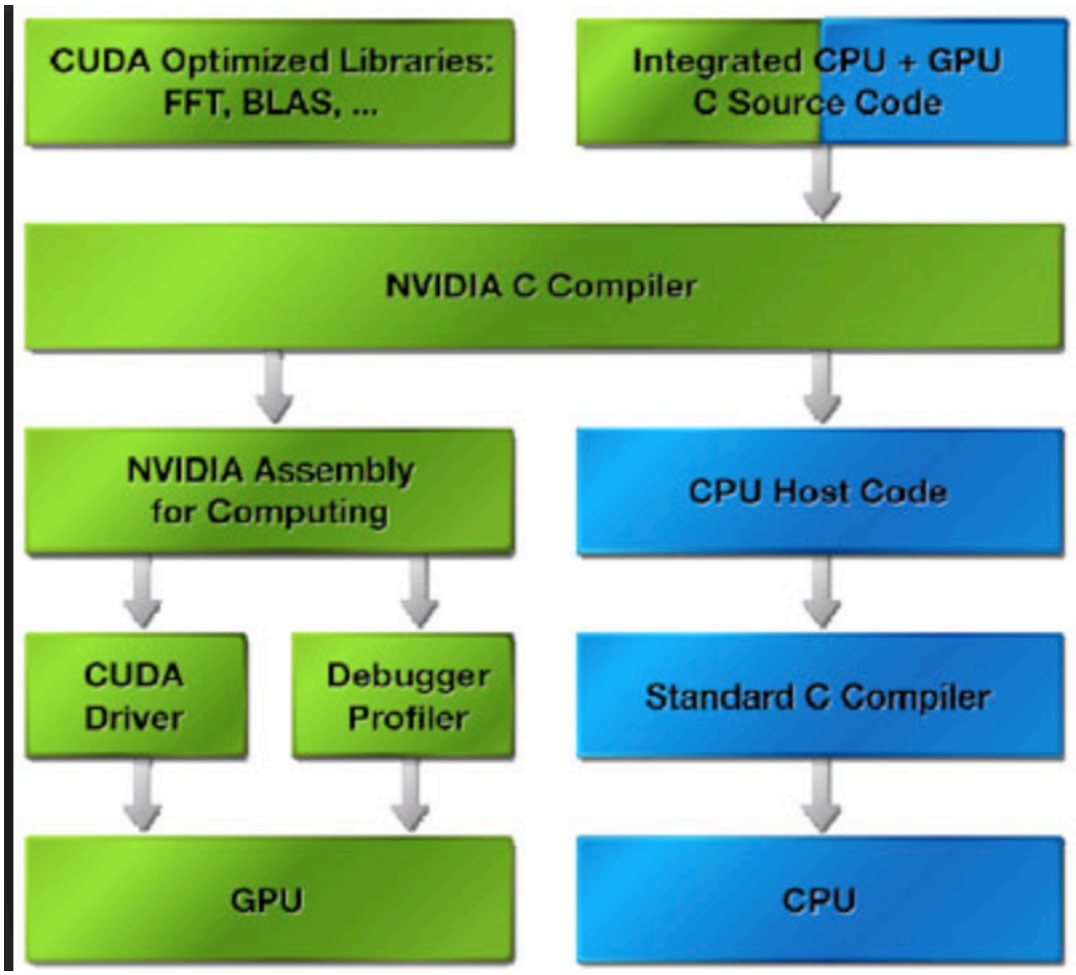


In the very beginning, GPU allowed imprecision because it is designed for processing graphics, and that is OK if there are some imprecision in graphics (the graphic on screen would not be different). However, that was a disaster for general computing (a tiny wrong with number would make the result huge different). This is one reason that GPU was not accepted before 2006.

Another reason is about algorithm. The software and development environment are immature for GPU. Developers cannot use it completely. They had to make a detour by using graphics interface to do program. That made programming on GPU very hard.

The third reason is about software. Because the iteration of GPU is fast, there is no general software for uniform GPU architecture. The GPU programming languages cannot be accepted and popular.

However, CUDA makes GPU computing possible. It not only takes GPU as general parallel processor, but also makes the next GPU generation capable for it.



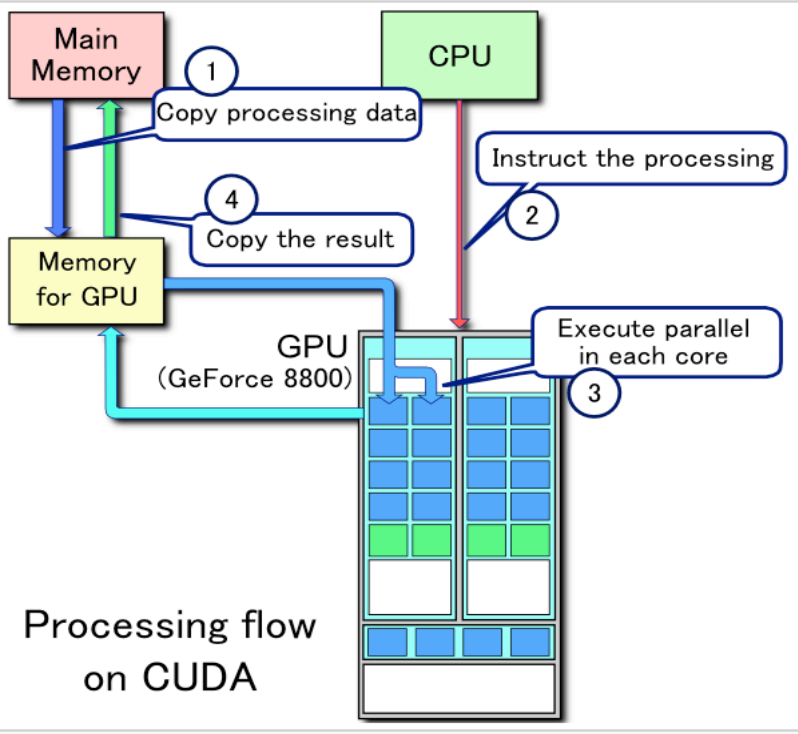
The advantages of GPU computing in CUDA

As we all know, in a computer system, a GPU is a coprocessor of the CPU. It helps CPU to do works CPU could not do. Initially used for computer graphics acceleration, handling graphics conversion, light, the building of triangular, cutting image and rendering work. Because of that, GPU has a different design purpose with CPU. The architectures of them are very different. In

order to meet the need of parallel computing and high volume of the graphic image applications, GPU optimized its hardware architecture for the throughput. In that way, it has a lot more lines and memory throughput compared to CPU, which makes it better at the completion of the work of massively parallel computing. Take the NVIDIA GPU TESLA K20 for example. It can reach 1 TFlop/s or more of double precision computing, and 3 TFlop/s or more of single precision computing, 250GB/s or more of memory throughput. All of these are much higher than the CPU nowadays. Although, much better throughput means better computing. But it gives a vision that how powerful it could be. That is way people work hard on it to make it possible. In other words, huge throughput indicates that GPU has great potential in general computing.

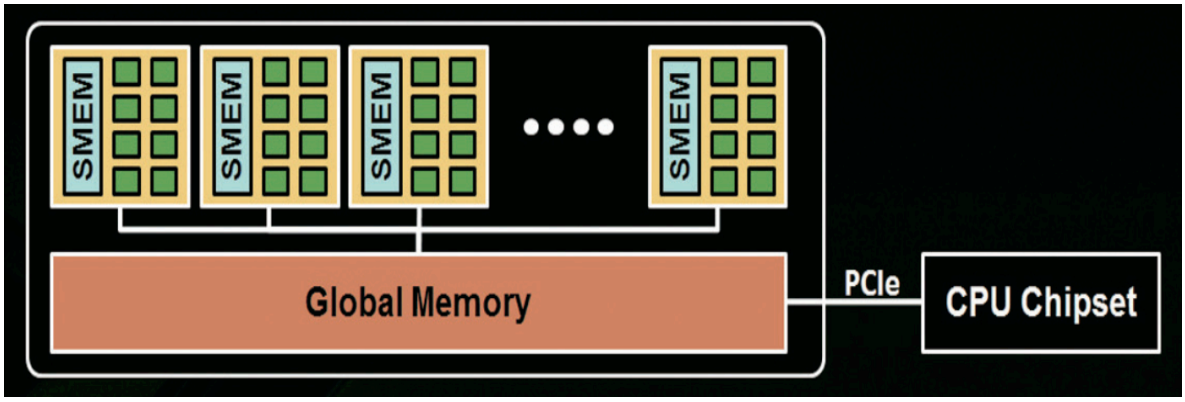
Therefor, from 2006, in that year NVIDIA launched its remarkable GPU, witch means the programmable GPU comes true, there are more and more developers take advantage of high-performance GPU to program software. They try to make general programming possible on GPU by lots of efforts. Gradually, they make GPU programming come true. It becomes powerful, easy to write and easy to learn for novices. It makes that language more standardized. The GPU-programming covers CAD, financial, structural mechanics, fluid mechanics, intelligence, automation, materials science, animation, image editing, molecular mechanics, mathematical analysis, oil and gas, physics, quantum chemistry, visualization, weather forecast and so on. Developers in different fields are able to do complicated and sophisticated program task, what CPU cannot do, on GPU. There is a lot of GPU software that meet their needs. These applications are far better that running on CPU. That makes personal computer more powerful than ever.

CUDA Programming Languages



Due to wide range of industries, different software has different requirements of development cycle (the speed, precision of developments), besides the programmers have different background and different programming skills, NVIDIA and its partners developed a wide variety of GPU programming techniques: CUDA C/C++, CUDA Fortran, OpenACC, HMPP, CUDA-x86, OpenCL, JCuda, PyCUDA, Direct Compute, Matlab, Microsoft C++ AMP etc. All have committed for developers to find the right one for themselves and their projects to do the GPU programming. However, in the future, these languages may become uniform. Just like CPU programming. Developers do not have to know what hardware in their computer and limited by one or two program languages. What they have to do is just focus on languages.

GPU architecture and programming model

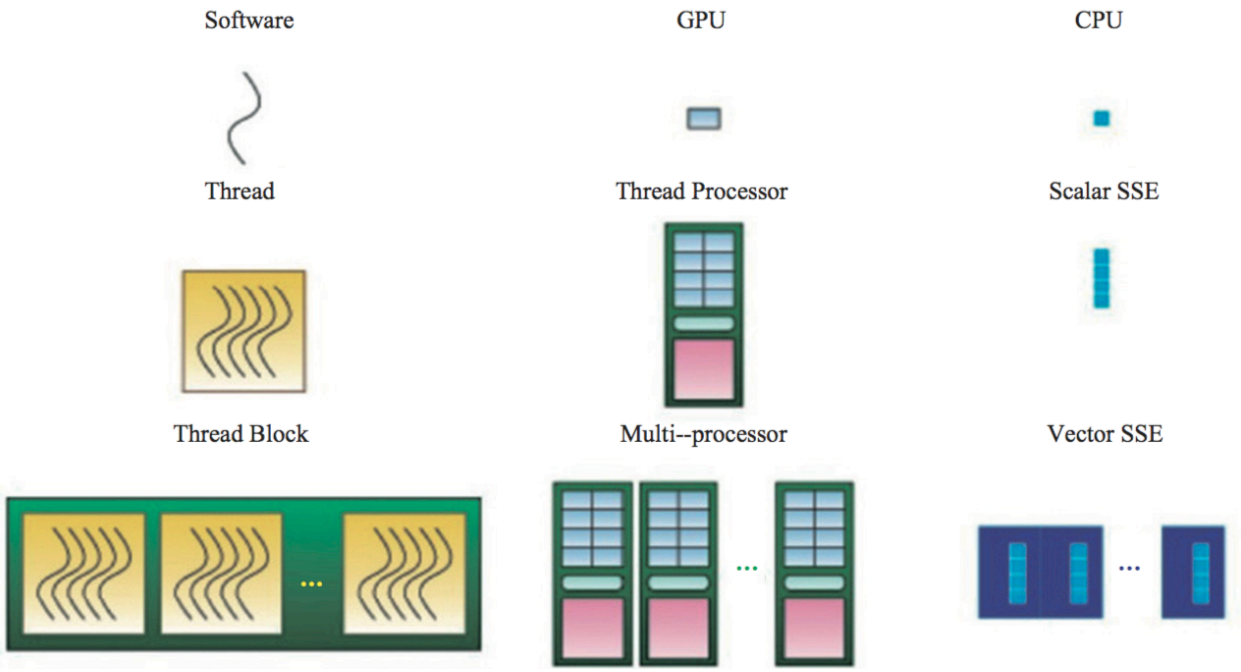


GPU is a processor, which designed to optimize the performance of parallel multi-threaded programming. This leads that GPU has significantly more cores than CPU in hardware design. This is called multi-core architecture. GPU with multi-core architecture, on instruction throughput and memory throughput, is far better than CPU, which makes it good at dealing with large-scale parallel computing tasks. In computer architecture, GPU is responsible for assisting CPU to handle high throughput multi-threaded tasks, which CPU is not good at. GPU connected to the CPU via PCI-E bus. Instructions and data executed are passing through this bus from CPU to GPU.

Not like complicated CPU, the internal architecture of GPU is very simple, which mainly consists of two parts: storage area and calculation area. The part of calculation consists of many streams multi-processor (also known as SM), which response for the implementation of GPU instruction. The part of storage called global memory, which response for data storage. What used for connect that two parts is SIMD. There also is a bus used for communication with CPU. The general architecture has a stable pattern. That means it was not change frequent. That also means the architecture makes designing of program languages more easily.

Programming technology of GPU

There is a lot of programming technology for GPU. The famous ones are: CUDA C/C++, CUDA Fortran, OpenACC, HMPP, CUDA-x86, OpenCL, JCuda, PyCUDA, Direct Compute, Microsoft C++ AMP etc.



Among these, CUDA C/C++ is designed by NVIDIA, and others are developed by NIVIDIA and its partners. For the languages, CUDA C/C++, OpenACC, HMPP, CUDA-x86, OpenCL, Direct Compute, Microsoft C++ AMP, are based on C/C++. CUDA Fortran, OpenACC, HMPP are based on Fortran language. JCuda is based on JAVA. PyCUDA is designed based on Python.

There are two kinds of multi-thread programming languages. One is compile guide, which means adding compile instructions into serial program to lead compiler to make the serial program to multi-thread program. Another is explicit thread model, which allows programmers to code parallel program directly, and use GPU threads explicitly. Compile guidance is more

easier compared with explicit thread model. It is easy to learn for novice, and convenient for optimization, and has short development life cycle. However, since it is autocompleted by compiler, handling GPU is relatively not flexible for threading model by it.

OpenACC and HMPP are compile-guidance languages. They bear resemblance with OpenMP, which is explicit thread model. CUDA Fortran can do in both way.

CUDA C/C++

CUDA C/C++ is a set of programming language designed for CUDA architecture GPU. It was published by NVIDIA in 2006. For now, CUDA 7.0 has already launched. CUDA 7 is a huge update to CUDA platform. CUDA is based on language C and C++. It is easy to learn for developers who code in C/C++.

CUDA C/C++ includes libraries for a variety of fields. That is convenient for the development to use. In CUDA 7.0, the libraries include cuFFTT (one library for Fourier transform), cuBLAS (one library for linear algebra), cuSPARSE (one library for sparse matrix operations), cuRAND (one library for generating random numbers), NPP (one library for image signal processing), Thrust (one library for handing sorting, scanning, conversion, reduction and other algotithms) and standard math library.

CUDA C/C++ can be run on Windows, Linux and Mac, and supports Visual Studio, Eclipse, and other integrated development environment. Windows developers can use Nsight tools to do some debugging. Meanwhile, Linux developer can use the CUDA-GDB to debug. The CUDA-memcheck can be used for memory errors checking on all of these platforms.

Developers can download the CUDA 7.0 development kit from NVIDIA official website, and start their work. That kit contains CUDA C/C++ nvcc (CUDA compiler), analyzer, programming manuals, GPU drivers, and some examples showing typical applications.

Programming Method of CUDA C/C++

Just like other GPU programming languages, CUDA C/C++ has two parts: one is the data transportation from GPU to CPU; another is the internal parallel algorithm. Here's an example of 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_aa<<<n/128,128>>>(d_x,d_y,n);
    cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
    cudaFree(d_x);
    cudaFree(d_y);
}
```

The programs written by CUDA C/C++ are pretty similar with regular C/C++ run on CPU. In the example above, we have two functions; one is main function the entry of program; the other is vec_add, which is the interface of GPU marking with "__global__". In the main function, program use cudaMalloc to allocate memory, and give the address to memory points. Second the function named cudaMemcpy transfer data from CPU to GPU. Then, vec_add is responsible for allocating threads and running program on the GPU. Finally, cudaMemcpy deliver the result of computing from GPU to CPU and release the memory.

Function vec_add includes two instructions; the first one use thread number and thread block number to calculate the only global thread number; the second instruction use every thread to sum numbers up in one direction, therefor all threads can finish the sum of all directions.

CUDA Fortran

CUDA Fortran is a GPU programming language launched by NIVIDIA and its partner GPI. The latest version of CUDA Fortran is Release 2013. It contains a compiler of Fortran2003. The programming pattern of CUDA Fortran is similar with CUDA C/C++. Fortran developers and CUDA C developers are able to learn the skill very quick.

Just as CUDA C/C++, CUDA Fortran is also accessible to CUDA libraries such as cuBLAS. It's available on LINUX, Mac OS X, and Windows. It can be written in Visual Studio and Eclipse (needs some plugins). CUDA Fortran is not supportable for debugging. However, it has a simulator called Mcuda-emu using CPU to simulate GPU for debugging.

The first thing of programming in CUDA Fortran is to download PGI CUDA Fortran Kit from PGI official website (latest version is PGI Release 2013). The Kit includes CUDA Fortran complier pgfortran, OpenACC compiler pgcc, CUDA-x86 compiler pgcpp, analyzer pgprof. adjustor pgdbg, information spec and help documents.

The programming methods of CUDA Fortran

CUDA Fortran perfectly merges CUDA GPU programming and Fortran. Anyone who familiar with CUDA C/C++ will be familiar with CUDA Fortran. They just have to

learn some of the basic syntax. Also, the Fortran developers have to learn some multi-thread knowledge to manipulate CUDA Fortran. Here is an example of CUDA Fortran.

This program has the same meaning of the program written by CUDA C/C++. Here are some features. First, in CUDA Fortran, we add attribute (global) before subroutine to show that this program is going to run in GPU. Second, we add 'device' before variables declarations to indicate that the variables are allocated in GPU. The names of allocate and deallocate functions are the same in CPU and GPU Fortran languages. Then, we just need a '=' symbol to transfer the data between CPU and GPU.

```
module cuda_kernels
contains
arributes(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
```

OpenACC

OpenACC is a set of open standard established by NVIDIA and its partners CRAY, PGI, and CAPS. It is very different from CUDA C/C++ or CUDA Fortran. It is a guidance language, and support for C/C++ and Fortran for now. It is able to make a loop or code block move to GPU and operate it. The syntax of OpenACC is quite similar with Open MP.

Compared to CUDA C/C++ and CUDA Fortran, OpenACC is more capable for paralleled programs. It makes developers to focus on the CPU code without considering how GPU implements it. The disadvantage is that it is not easy to do more sophisticated programming. Because of that, CAPS company's OpenACC program language have a conversion function, witch can convert OpenACC to CUDA C/C++ automatically, and optimize the code.

In order to do OpenACC program, Linux users can download OpenACC or HMPP3.2 from CAPS's official website. Meanwhile, one can download PGI Kit from PGI website (the latest version is PGI Release 2013). That kit has OpenACC compiler pgcc.

```
_global_ void vec_add(Float *x, Float *y, int n)
{
    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_aad(x,y,n);
    free(x);
    free(y);
}
```

The methods of programming in OpenACC

The example above is a very simple OpenACC program. There is only one instruction that is different from CPU program, which is called compiler guidance. The compiler guidance is used to allocate n threads to GPU and receive the data from CPU, and transfer the result to CPU after computing. All of these operations are completed for users automatically.

CUDA-x86

CUDA-x86 is a compiler launched by PGI Company, designed for running the same code simultaneously on CPU and GPU. OpenACC transfers code from CPU to GPU via compiler. By contract, CUDA-x86 transfers code from GPU to multi-core CPU, so that it could be run on it.

CPU architecture and GPU bear some resembles. For multi-core CPU, one core communicates others via MIMD paralleled architecture usually among 4 to 12 cores. In every core, it process AVX or SSE instruction in a parallel way. Hence, we can see SM in GPU as single core in CPU, and see single core of CUDA as Scalar SSE processor unit.

In that way, CUDA-x86 maps CUDA blocks in the CPU cores. It also makes SSE or AVX SIMD processing units to do CUDA tread works. Finally it optimizes the code, so that users can run it both on CPU and GPU.

CUDA-x86 supports Windows, Linux and Mac OS X platform. Developers can use the pgcpp compiler in PGI Release kit to compile CUDA code, so that it would be possible to be run on multi-core CPU.

Conclusion

With the development of personal computer, graphics processing is becoming common from station to PC. Meanwhile, due to its architecture, it is to be found that more suitable to handle multiple threads tasks. We do not just focus on the graphics processing, but also focus on the general computing. GPU is no long just a Graphics Processing Unit, but is a second central processor.

Based on its advantages of architecture (huge volume of throughput and no delay paralleled computing), GPU companies make it more suitable for general computing. Since 2006, GPU programming becomes possible for everyone. CUDA is a set of GPU programming languages launched by NVIDIA and designed for NVIDIA GPU. Developers can use CUDA to write general-purpose code without knowing how the hardware works and without learning anything about graphics.

There are four popular and common CUDA language, CUDA C/C++, CUDA Fortran, OpenACC and CUDA x86. Each of them has their own advantages and disadvantages. Developers could choice what they need.

In the future, GPU will be more important than ever. Some experts even say that GPU will replace CPU. The GPU languages also will become objected and standard.

Reference

[1] Han Bo and Zhou Bingfeng, “A performance Model for General-Purpose Computation on GPU”, Institute of Computer Science and Technology, Peking University, Beijing, 2009.

[2] Wang Zehuan, Wang Peng, “Introduction to GPU Parallel Programming Technology”, NVIDIA Co. Beijing, 2013.

[3] Geforce 3 Product Overview 06.01v1. http://www.nvidia. com/page/geforce3.html, NVIDIA, 2001.

[4] GeForce 256. http://en.wikipedia.org/wiki/GeForce_256 . Wikipedia, Oct 2012.

[5] Tesla C2050/C2070 GPU Computing Processor Supercomputing at 1/10th the Cost, NVIDIA, Jul 2012.

[6] GPU-Accelerated applications, NVIDIA, Jun 2012.

[7] CUDA Fortran Programming Guide and Reference, PGI, Jun 2012.

[8] The OpenACC API QUICK REFERENCE GUIDE Version1.0a, CAPS CRAY NVIDIA PGI, April 2012.

[9] PGI CUDA-x86: CUDA Programming for Multi-core CPUs, PGI, November 2010.

[10] Enhua Wu and Youquan Liu, “General Purpose Computation on GPU”, Laboratory of Computer Science, Institute of Software Chinese Academy of Sciences, Beijing, 2004.