GPGPU origins and GPU hardware architecture

Practical term report from Stephan Soller

High performance computing center Stuttgart[1]

Stuttgart Media University[2]

1. http://www.hlrs.de/
2. http://www.hdm-stuttgart.de/

Introduction

General Purpose calculations on Graphics Processing Units (GPGPU) is a term that refers to using the graphics processing unit (GPU) for general purpose calculations instead of graphics rendering. The term in itself implies a form of contradiction since it shows that a device (the GPU) is used for work it was not meant to be used for.

Nevertheless todays GPUs are very powerful calculation devices. They have long passed the point where they were only usable for graphics rendering. Many efforts to use that calculation power are centered around the term “GPGPU“ or similar terms like “GPU Computing”.

Origins of GPGPU

How GPUs evolved

The origins of GPUs are fast realtime graphics. First and foremost it was all about drawing graphics on the screen as fast as possible. In the 1980s and early 1990s this kind of work was often done by the CPU which, due to its architecture, was not very efficient at that task. As with all very demanding and domain specific tasks this work was gradually offloaded to a dedicated processor: the graphics processing unit for PCs was born.

It should be noted that developments of that kind are not unusual. It happened before as DMA controllers started to offload the input/output work from the CPU. It happened later as the physics processing unit (PPU) was built to handle physics calculations (albeit without much success). A more current development in this regard is the offloading of the TCP network stack or parts of it to the network card. Again in a very specialized domain to handle large amounts of network traffic (e.g. 40 or 100 GBit/s connections).

At first GPU vendors focused on 2D acceleration for desktop systems as well as monitor resolution and the quality of the generated analog signal. However a new branch gained popularity over time: 3D acceleration. Graphics APIs like DirectX, OpenGL and 3dfx Glide, developed for computer games and data visualization, were designed with hardware support in mind. More and more calculation steps (or pipeline stages) of these APIs were moved to dedicated hardware. Thanks to the gaming market this area was in fact so popular that for a short time dedicated 3D accelerator cards like the 3dfx Voodoo were widely used.

From fixed function to general purpose

As the GPUs became faster in rendering basic computer graphics (like basic geometry and textures) demand for more advanced techniques grew. New pipeline stages for these effects were added to the APIs and quickly implemented in hardware. At that time most functionality was “fixed function”, that is for each effect dedicated API calls existed and were implemented in hardware. Each new effect (e.g. depth of field) resulted in API and hardware changes, greatly limiting the possibilities of graphics programmers.

This limitation resulted in a major architecture switch of GPUs. Highly specialized function units were replaced by small and simple moderately specialized processors. Such hardware contained some processors for vertex specific calculations, a larger amount of processors for pixel specific calculations (because there are more pixels than vertecies on a screen) and some fixed function parts e.g. for rasterization. Complete fixed function was replaced by simple and limited programmable execution units. This greatly increased the flexibility of the hardware and allowed the GPU vendors to better scale horizontally. Performance could be increased by adding more of these vertex and pixel processors as well as other fixed function hardware like texture units.

However balancing the number of pixel and vertex processors is difficult as the workload distribution depends on the individual application and scene. Usually there are much more pixels drawn for a frame than vertices are transformed or calculated. But in some situations it can happen that there are not much vertices in a scene but many pixels are drawn (e.g. particles). In such a situation almost all the processing power of the vertex processors would be useless since these are idle most of the time. The direct opposite is a scene with a very complex model where all vertex processors are utilized but cannot produce enough pixels for all pixel processors, leaving some of these processors idle.

Since this lead to a suboptimal and uneven work distribution this aspect of the architecture was refined further. The specialized processor types, each with its own special instructions, were unified into one less specialized processor type which was able to handle all these tasks. The processors became more complex but also more flexible. That refined architecture (usually called Unified Shader Architecture) stays efficient even on changing workloads as the different tasks can be distributed over all processors.

Advent of GPGPU

Early approaches to use GPUs for more general calculations date back to the year 2000[3]. However GPU hardware was purely fixed function at that time. All tasks had to be mapped to the computer graphics domain and solved there. Textures were used as memory and operations like blending performed the actual calculation. This approach is difficult for problem domains that do not map well to computer graphics. But using a texture as a data structure rather than an image is still well suited for general calculations very close to computer graphics. For example in normal mapping a texture (the normal map) is used to store normal vectors instead of color information. However the actual calculations are no longer performed with operations like blending.

Since then GPGPU grew with the abilities of the GPUs. For example in 2002 the vertex and pixel shaders were used for algorithms like matrix multiplication and 3-SAT[4]. In 2005 radiosity calculations were performed on the GPU[5].

With the marketing campaigns of the nVidia CUDA and Tesla brands a more widespread interest in the area came. Followed by a push of nVidia into the high performance computing sector. This introduced a wider range of programmers into the possibilities of GPGPU calculations and made them aware of the computational power GPUs offer.

The reasons for this push into the high performance computing sector are diverse and as complex as the computer graphics market itself. A proper analysis of these reasons would be a topic for a complete work of its own. However there are some likely reasons and one of them is related to the “uncanny valley”, an effect in the perception of human like objects.

Up to a certain degree of realism the object is still accepted as human. We even add missing details by our selfs. An extreme example of this case is a smiley or emoticon: a simple geometric form which is enriched with an emotional meaning by our perception. However with decreasing abstraction and at a certain level of realism our brain stops to induce missing information. Instead irritation is caused to draw attention to the fact that something is wrong with the object in question. The degree of realism where the uncanny valley begins depends on the kind of

3. C. Trendall and A.J. Stewart , "General calculations using graphics hardware with application to interactive caustics," In Rendering Techniques '00 (Proc. Eurographics Rendering Workshop), pp. 287-298. Springer, June 2000.. http://www.dgp.utoronto.ca/~trendall/research/msc2k/index.html
4. Chris J. Thompson, Sahngyun Hahn, Mark Oskin, "Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis," micro, pp.306, 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'02), 2002. http://www.cs.washington.edu/homes/oskin/thompson-micro2002.pdf
5. Greg Coombe, Mark J. Harris, and Anselmo Lastra. 2005. Radiosity on graphics hardware. In ACM SIGGRAPH 2005 Courses (SIGGRAPH '05), John Fujii (Ed.). ACM, New York, NY, USA, , Article 179 . DOI=10.1145/1198555.1198782. http://www.cs.unc.edu/~coombe/research/radiosity/

perception. For example still images are not that much affected by the uncanny valley, but movements like walking are heavily effected (we easily notice someone who doesn't walk or move properly). This is one reason why animations are that difficult. Even motion captured movement often needs to be fine tuned by experienced animators. Real time graphics lies somewhere between these two extremes but there are many more influencing factors. Scenes with an unnatural perspective are less affected by the uncanny valley. A strategy or building game where the player sees the world from an omnipresent or god like perspective for example. Scenes with an ego perspective on the other hand are more affected since they better resemble our natural perspective.

This effect made the production of photo realistic content very expensive and slowed the advance of the gaming market. Very much effort has to be put into many small details to make a scene believable. Traces of that can be observed in the current state of the gaming industry. This might be one of the reasons why hardware vendors pushed into alternative areas to open up additional revenue streams.

There are many other possible reasons for the direction of the video graphics hardware vendors: business strategy or relative similarity between CPUs and GPUs, to name just a few. However the uncanny valley might be one of the less obvious ones.

GPU hardware architecture

The hardware architecture of a graphics processing unit differs from that of a normal CPU in several key aspects. These differences originate in the special conditions in the field of realtime computer graphics:

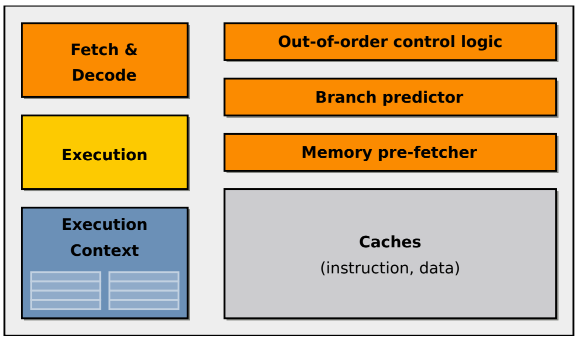
- Many objects like pixels and vertices can be handled in isolation and are not interdependent.
- There are *many* independent objects (millions of pixels, thousands of vertices, ...).
- Many objects require expensive computations.

The GPU architectures evolved to meet these requirements.

To better understand the differences between CPU and GPU architectures we start out with a CPU architecture and make several key changes until we have a GPU like architecture. The rest of the chapter is partially based on a presentation of Kayvon Fatahalian (Stanford University) from the SIGGRAPH 2009: “From Shader Code to a Teraflop: How Shader Cores Work”.

Modern CPUs

Basic CPU subsystems



Most modern CPUs that are used in high performance computing as well as servers and desktop systems are much more complex than the simple calculations machines they are often programmed as. To be able to calculate something a CPU needs:

- To fetch & decode instructions from the memory
- An execution unit that does the calculations (ALU, FPU, ...)
- Some kind of execution context (registers)

With that alone however a CPU would be very slow due to some effects:

- Memory latency: Fetching data from the off chip main memory is a very time consuming operation. If an execution unit needs some data for a calculation it's stalled until the data is available, potentially waiting for a very long time.
- Suboptimal program flow: The way a program uses the execution units of the CPU may be inefficient and leaves some of the execution units idle.

CPUs contain several additional complex subsystems in order to overcome these problems and to boost the performance:

- Out-of-order execution: The instructions are reordered to better utilize the internal execution units of the CPU. Integer arithmetic (ALU) and floating point operations (FPU) are executed in parallel for example. After execution the original order needs to be restored since this optimization is transparent to the program.
- Branch prediction: When a program branches the CPU doesn't know the next instructions and which data they need until the branch condition is computed. Usually the CPU would be stalled immediately after the branch because it now needs to fetch the instructions and data for the taken branch. To limit the impact of branches the CPU tries to predict which branch will be taken and loads the instructions and data for that branch. After that the CPU performs a speculative execution of the predicted branch. If the prediction was right a pipeline stall has been avoided, if it was wrong the program execution is stalled and can only resume as soon as the required instructions and data is available.
- Memory pre-fetching: Based on the characteristics of a program the CPU loads data from the main memory that *might* be needed by the next instructions.
- Cache hierarchy: To reduce the memory latency a CPU utilizes several caches.

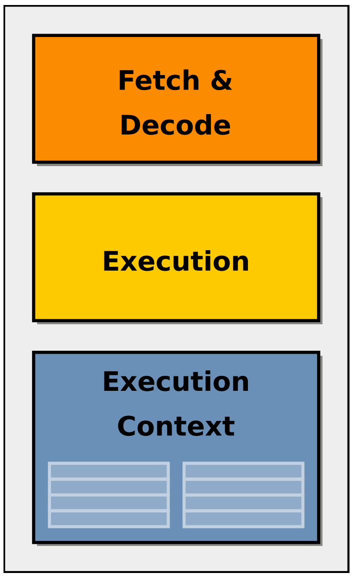
The purpose of all these optimizations is to improve the performance of a single instruction stream. Since CPUs traditionally run only one program at a time this was the only performance that mattered for a long time. However the CPU architecture is aware of the fact that multiple programs are run in a time sliced manner, e.g. by providing a memory management unit and a translation look aside buffer to efficiently implement virtual memory. During the last years this fact has also been used to optimize the performance by adding hardware to handle a second instruction stream (e.g. a second ”Fetch & Decode” block). If one instruction stream is stalled because of a mispredicted branch the second stream can be fed to the execution units, resulting in a better over all utilization of the execution units.

This approach was first introduced with later Intel Netburst architectures (Pentium 4 HT) and called "Hyper Threading". The Netburst architecture had an extraordinary long pipeline and therefore pipeline stalls had a very high penalty. Hyper Threading reduced the impact of these but to the required operating system support limited the usefulness of this optimization.

The Niagara architecture (UltraSPARC T1) of Sun Microsystems takes this idea even further. It automatically manages 4 threads per core and an instruction from a different thread is executed every cycle. This interleaving hides memory latency. Later versions of this architecture increased the number of concurrent threads even further to 8 or 16.

No focus on single instruction stream performance

A simple processing core



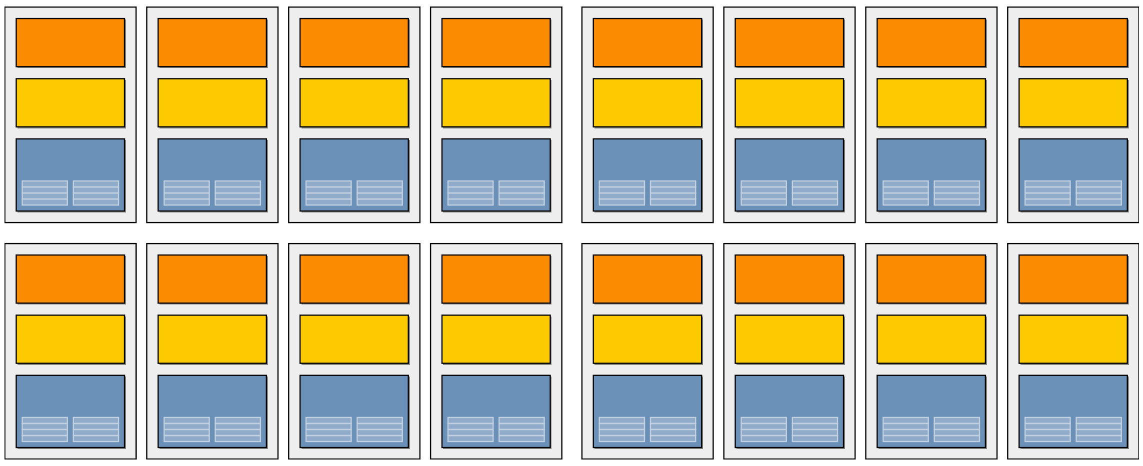
Tasks in computer graphics usually offer good parallelization potential. Many operations need to be done on all pixels of an image or on all vertices of a scene. These operations can usually be computed more or less independent of each other. From the hardware architecture point of view this kind of workload can be distributed over many cores and does not require strict sequential operation like many CPU based algorithms do.

In order to achieve a large number of cores these cores need to be simple. Therefore we reduce the processing core to the absolutely required minimum:

- Instruction fetch and decode
- Execution unit
- Execution context

We have effectively removed all logic that boosts single instruction stream performance but gained the ability to put more cores on a chip. This increases the potential calculation power of our architecture. For example we can now put 16 simple processing cores together and process 16 instruction streams in parallel.

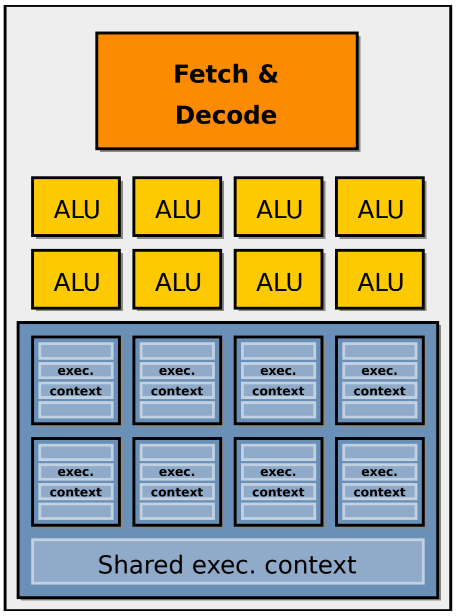
GPU with 16 processing cores



Share instructions between streams

When executing a program that calculates a pixel the same instructions has to be applied to a very large amount of data. On a picture with the dimensions of 1280×1024 the same program has to be executed 1.310.720 times, once for each pixel. In our current 16 core design we would process all these pixels in blocks of 16 at a time.

A 8 units wide SIMD core



However most of these 1.310.720 instruction streams do exactly the same. They are executing the same instruction just on different data. This gives us the opportunity for another architecture optimization: single instruction multiple data (SIMD) processing. Instead of one decode & fetch unit per instruction stream we reuse the decode & fetch unit for several instruction streams. With this optimization our new SIMD processing core consists of one decode & fetch unit and a number of execution units, each with an associated execution context. For example an 8 units wide SIMD processing core would consist of one decode & fetch unit, 8 execution units and 8 execution contexts.

If all 8 execution units share the same instruction all works well and we get 8 times the performance without much organizational overhead. But if the shared instruction stream contains branch instructions it can happen that not all 8 execution units continue with the same instruction. A branch instruction like a conditional jump continues the control flow of the program at different positions depending on a runtime condition (e.g. the value of a variable or pixel coordinate). With such an instruction it can happen that some execution units of one SIMD core need to continue execution on another position than the remaining execution units. In that case we have to execute the instructions for both branches. The execution units not in the currently executed branch then need to ignore these instructions.

The amount of branching in the shared instruction stream limits the usefulness of SIMD optimizations. The more branching occurs within the same SIMD core the the lower the SIMD performance gain becomes.

If we update our current 16 core architecture that way we get 16 SIMD processing cores, each able to do the same operation on 8 data streams. In the optimal case we can now process 16 × 8 = 128 programs in parallel. Note that we can handle one unique instruction stream per SIMD core without performance loss since the SIMD cores are independent of each other. If however the control flow within one SIMD core varies we lose performance since the SIMD core has to execute the instructions of each branch sequentially.

Impact of SIMD on the programming model

There are two different ways to represent a SIMD architecture to programmers:

- Explicit by using vector instructions
- Implicit by representing each execution unit as thread

Both approaches are widely used and have their advantages. Explicit SIMD for example is most common on CPUs in the form of SSE instructions. One instruction can operate on 4 single precision floating point values there. On GPUs this explicit approach is used by Intels Larrabee with 16 component vectors. The big drawback of that approach is that it's not transparent to the programmer. While writing the source code the programmer has to use vector data types and instruction or all but one execution unit are unused.

Implicit SIMD on the other hand is transparent to the programmer. While this is usually more convenient the danger lies within the illusion that each thread can follow its own control flow. The programmer needs to be aware of the architecture characteristics but does not have to explicitly use vector data types and instructions. This approach is currently used by nVidia.

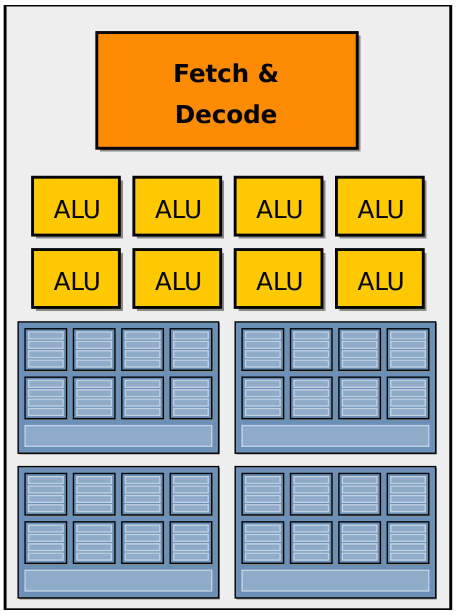
AMD chips currently are a hybrid of both approaches. Many execution cores share the same fetch & decode unit. However each execution core can operate on a 4 or 5 component vector by itself.

Interleave streams to hide latency

In normal computer graphics applications a very large number of independent objects (e.g. pixels, vertices, ...) need to be calculated. With thousands or millions of instruction streams available we can use this concurrency to mask memory latency. If one instruction stream cannot proceed because of missing data we switch to one of the other available instruction streams. If that stream gets stalled, too, we switch to the next one, and so on. With an sufficiently large number of instruction streams the execution units can be kept fully utilized until the data for the first stalled instruction stream is available.

Finishing one instruction stream may take a long time (depending on how often it gets stalled), but when looking at all streams together the overall throughput is maximized.

A SIMD core with execution contexts to interleave 4 instruction streams



Switching between instruction streams needs to be very fast to effectively implement this. We achieve this by storing more than one execution context in the execution unit. Instead of swapping the registers on each context switch with a stored execution context we just keep all execution contexts in registers. This requires a larger amount of registers but allows instant switching between instruction streams.

This architecture has an interesting consequence: A simple program requires only a small execution context (few registers) to operate upon. Therefore more of these execution contexts can be kept on chip and the instruction stream can be better interleaved, hiding memory latency very effectively. More complex programs require a bigger execution context (more registers) to operate resulting in decreased interleaving. Usually optimization leads to more complex programs and because of that in some cases simple programs can perform better in terms of memory access than "optimized" ones.

Memory access

Modern CPUs use a sophisticated caching hierarchy to overcome the high latency to the main memory. If data is present in one of the caches it can be accessed relatively fast. The CPU itself maintains the caches and takes care of moving data between the caches and the main memory. Since programs usually show some kind of locality in their memory access patterns these caches are efficient for most tasks. The hardware takes care of everything these optimizations are transparent to the programmer. But for high performance applications it can still be of use to know about the semantics of the cache hierarchy. Especially when working with multiple threads since that can require the caches to be synchronized which can have a negative performance impact.

GPUs on the other hand usually do not operate on the computers main memory. Often a GPU is connected to its own off chip memory which is used for texture and vertex data. The size of this graphics memory varies but historically it has been around fourth or half the size the main memory. Before the GPU can start to work on something that data first needs to be moved to the graphics memory. The speed of that operation depends on the connection between main memory and the graphics board. Therefore it varies heavily but a rough value to get some perspective is about 5 GiByte/s.

GPU programs that work on this graphics memory have been relatively short in the early days. The was no locality in the access patterns, except for texture access. If one pixel fetched the texture data at a specific position chances are high that the next pixel also fetches texture data near the same location. Because of that GPUs did not provide a sophisticated caching hierarchy. Instead they offer a high bandwidth to the graphics memory and provide texture caches. Texture caches are basically overlays over a specific block in the graphics memory. Access to such memory blocks with special instructions (texture fetches) are cached. This reduces the latency when working with texture data.

Memory access which is not done though the texture cache suffers the full latency of the graphics memory. We need to rely on the instruction stream interleaving in that case. The memory bus of GPUs is organized in a way to efficiently handle large chunks of data. Usually a chunk is large enough to feed all execution units of a SIMD core with one float value (the most common data type in computer graphics today). If a program accesses memory in a way that all execution units in a SIMD core can get their required data with one bulk transfer we can utilize the full bandwidth of the graphics memory. Currently this is about 150 GiByte/s. But if a program shows more random memory access patterns these bulk transfers cannot be used efficiently and some or most of the bandwidth is wasted.

In oder to overcome these limitations in memory access most GPU programming APIs provide some sort of chip local memory (usually registers). The program can use this memory as a manually managed cache. First moving data from the graphics memory to the chip local memory and using that memory in any further calculations. That limits the impact of graphics memory latency to the one time access on program start. However the on chip memory is not very large, usually some KiByte at most for an entire SIMD core.

A drawback of that approach is that this chip local memory would usually be used to store additional execution contexts of interleaved instruction streams. The more chip local memory is used as manually managed cache the less instruction streams can be interleaved to hide latency. Because of that the programmer needs to balance the use of that chip local memory.

Comparisn between CPU and GPU architecture

CPU and GPU architectures share the same basic execution model. Fetch and decode an instruction, execute it and use some kind of execution context or registers to operate upon. But the GPU architecture differs from the CPU architecture in the three key concepts introduced in the previous chapter:

- No focus on single instruction stream performance
- Share instructions between streams (SIMD)
- Interleave streams to hide latency

These concepts originate in the special conditions of the computer graphics domain. However some of the ideas behind these concepts can also be found in modern day CPUs. Usually inspired by other problem domains with similar conditions.

The MMX and later the SEE instruction sets are also SIMD (single instruction multiple data) based for example. SEE allows to work on multiple integer or single precision floating point values simultaneously. These instructions were added to allow faster processing of video and audio data. In that case the raw volume of multimedia data forced new optimizations into the CPU architecture.

However most CPU programs do not use these optimizations by default. Usually programming languages focus on well known development paradigms. Vectorization or data parallelism unfortunately isn't such a well known paradigm. In the contrary, it is usually only adopted when necessary because it adds complexity to a program. Therefore very much software does not use SIMD style instructions even if the problem domain would offer it. Especially in a time where development time is considered very expensive and performance cheap it's hard to justify proper optimization. This situation gave SIMD instructions on the CPU an add-on characteristics instead of being the instructions of choice to efficiently solve certain problems.

The SIMD architecture on GPUs was adapted out of necessity. Otherwise the large amounts of data would be more difficult to handle. We also save space by not needing a fetch and decode unit for each execution unit. This saved space makes the chip more compact and cheaper to produce (more chips per silicon wafer).

Out of different motivations CPU and GPU architectures moved into the same direction in regards to SIMD optimization.

Another similarity between CPUs and GPUs it the current development towards multicore and manycore CPUs. Out of certain physical limitations (speed of light, leakage voltage in very small circuits) CPUs can no longer increase the performance of a single instruction stream. CPU frequency and with it the instructions per second cannot be increased indefinitely. The increased power consumption of high frequencies will damage the circuits and require expensive cooling. The Netburst architecture (Pentium 4) was designed for very high frequencies of up to 10 GHz. However the excessive power consumption limited the frequencies to about 3 to 4 GHz. Frequencies of up to 7 GHz were achieved under special cooling conditions for a very short time (30 seconds up to 1 minute before the CPU burned out).

This left the CPU vendors no other choice but to scale horizontally. Adding additional cores theoretically multiplies the performance. However one single instruction stream (or thread) can no longer use that performance. In order to use the performance of all cores for one task a program has to coordinate several instruction streams on different cores. However this usually requires the threads to synchronize on various occasions which makes programming and debugging more difficult.

In order to integrate more and more cores onto a single CPU speed and complexity of a single core is usually reduced. Therefore single core CPUs usually have a higher performance for one instruction stream than the more modern quad core CPUs. This is similar to the idea used in the GPU architecture: many simple processing cores are more effective than one large core. While CPU cores are still far more complex than GPU cores they might develop into more simple layouts to allow better scaling. GPU cores on the other hand might evolve into more complex layouts to provide more developer friendliness. However the idea of horizontal scaling is present on both architectures.

Programming models for GPUs

There are a variety of APIs that can be used to perform general purpose calculations on the GPU:

- ATi CAL (Calculation Abstraction Layer)
- nVidia CUDA (Compute Unified Device Architecture)
- OpenCL (Open Compute Language)

There are more smaller frameworks but CUDA and OpenCL are currently the most used ones. ATi CAL is an older low level interface to ATi GPUs. It required to write low level assembly like code (ATi Intermediate Language) for the parts that were executed on the GPU. AMD has replaced CAL by OpenCL which can be programmed in a C like high level manner. Therefore ATi CAL is likely to vanish from the GPGPU sector except for very specialized or low level purposes.

The high level APIs CUDA and OpenCL both map the architecture of multiple SIMD cores directly to a programming model:

- Each SIMD core is represented by a group of threads.
- The whole GPU with its many SIMD cores is represented by a number of thread groups.

The exact terminology of these things differ between APIs. In nVidia CUDA for example a thread group is called a block.

All threads of a thread group will be executed on the same SIMD core. Therefore a thread group should at least contain a thread for each execution unit in the SIMD core. Since interleaving is used to hide latency we also need to provide enough threads for effective interleaving. In our example architecture above we added enough registers to interleave 4 instruction streams. Combined with 8 execution units per SIMD core this makes a minimal thread count of $8 \times 4 = 32$. Thread groups with fewer threads will waste calculation performance since execution units will be idle. In practice even more threads are needed in a thread group to effectively hide latency.

In the CUDA and OpenCL programming models the programmer writes code for a single thread. Whenever this code is executed on a GPU the programmer needs setup the number of threads and thread groups. Therefore algorithms targeting the GPU need to be build with this two level parallelism in mind.

Further reading

The above chapters were just a introduction into the wide topic of GPGPU. For practical usage of GPUs detailed knowledge of the GPU architecture is not strictly required albeit very useful. More important is the knowledge of the used API like CUDA[6] and OpenCL[7]. Programming paradigms or software architecture of code that uses the GPU is another research area. Programming models like GRAMPS[8] will provide further insights into these topics.

6. nVidia CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html
7. OpenCL. http://www.khronos.org/opencl/
8. Sugerman, J., Fatahalian, K., Boulos, S., Akeley, K., and Hanrahan, P. 2009. GRAMPS: A programming model for graphics pipelines. ACM Trans. Graph. 28, 1, Article 4 (January 2009), 11 pages. DOI = 10.1145/1477926.1477930. http://doi.acm.org/10.1145/1477926.1477930