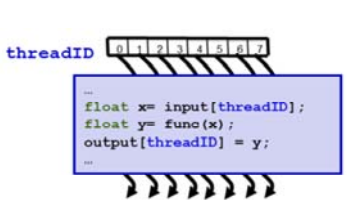
Parallel Programming



Programming Massively-Parallel Architectures (GPUs and accelerators)

2D Laplace Example on GPU

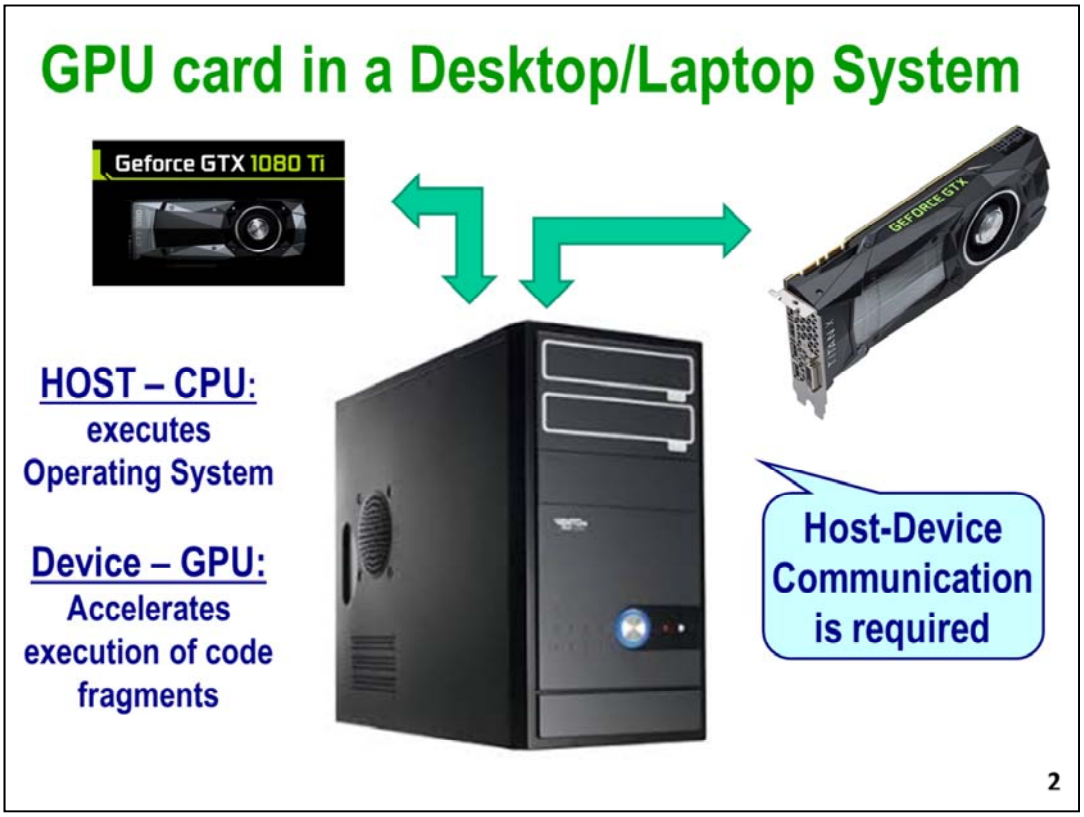
https://developer.nvidia.com/intro-to-openacc-course-2016
http://devblogs.nvidia.com/parallelforall/

juancarlos.moure@uab.es Office: QC-3034

This lesson introduces the architecture of GPUs (as execution accelerators) and analyses the programming language, tools and methodology required to adapt a program code example written in C language to be executed in a GPU.

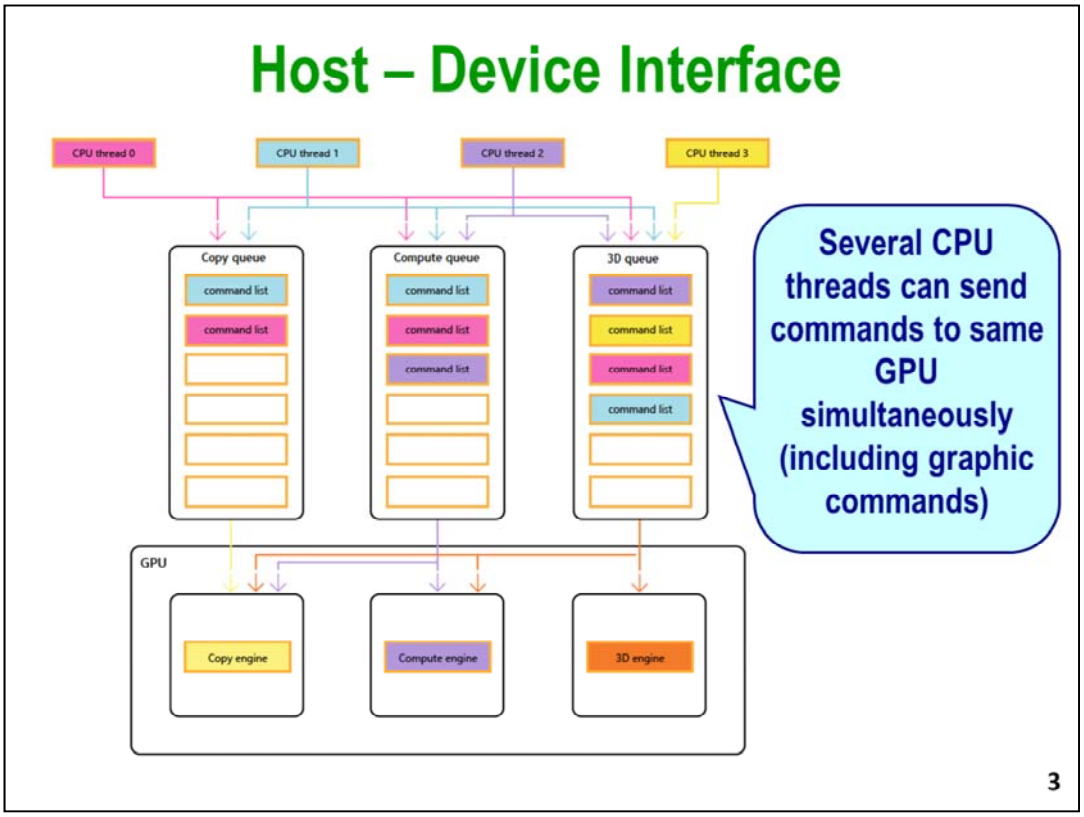
The two internet links provide lots of information about programming GPUs and accelerator devices. The first link is a recorded webinar (with videos and slides in PDF) that introduces the OpenACC programming language for massively parallel accelerator devices (like GPUs). The *leit motiv* of OpenACC is “more science and less coding”, which provides good inside about the purpose of this parallel language (which is very similar to OpenMP). You can register on the course and enroll in a QwikLab to practice with OpenACC remotely, without requiring any special equipment, apart from a web browser.

The second link is an NVIDIA web with multiple examples of applications of GPU, and with many code examples written using OpenACC but also using CUDA (a lower-level set of language constructs that provide access to the internal elements of the GPUs and, sometimes, is necessary for producing efficient code for GPUs).



A GPU is an accelerator device connected to a host CPU. The GPU alone cannot be used as the only processing device in a computer system, since it is not able to run a complete Operating System (it does not provide support for supervisor mode and user mode, interrupts, …). Then, the GPU device (sometimes referred to as the *device*) acts as a *coprocessor* of a central CPU, which is called the *host* and is responsible of the Input and Output operations.

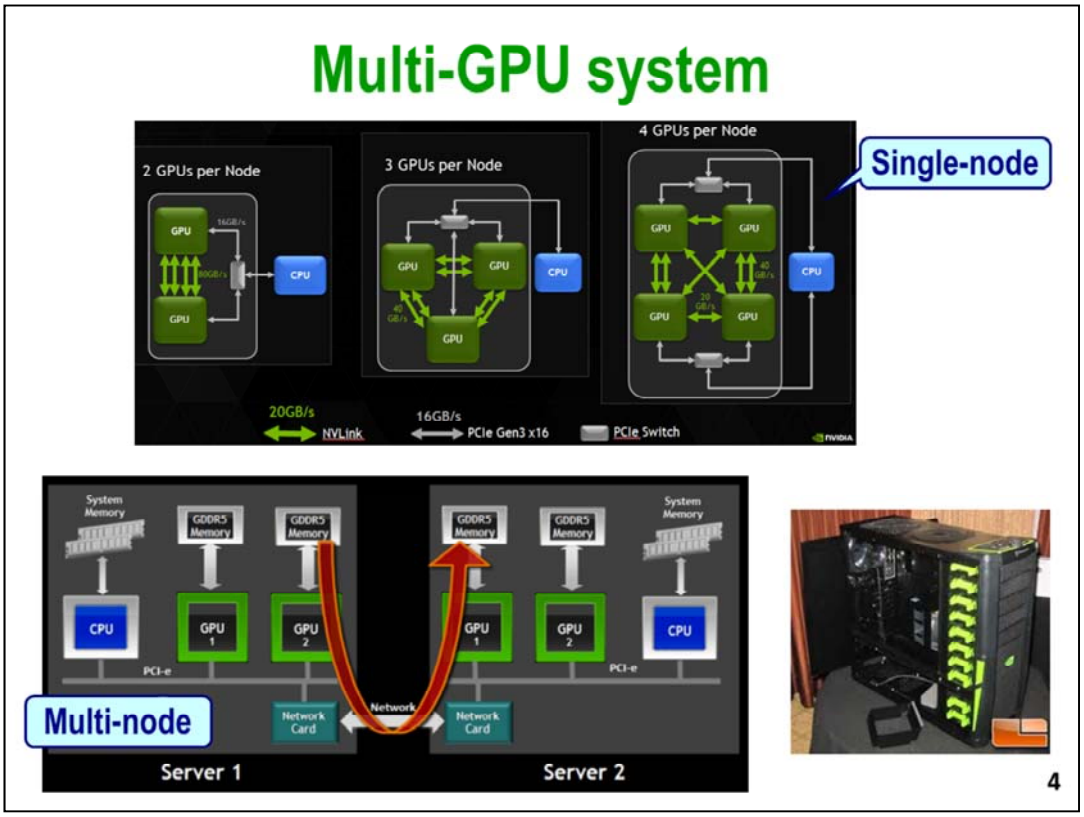
Most systems integrate a GPU into a Graphics card attached to the main board of the computer, as depicted in the slide. In those cases, the GPU is connected to its own memory (the device memory), while the CPU is connected to the main system memory (or the host memory). Connection between CPU and GPU is done through an I/O connection link using the PCI express protocol. This connection is used to move data between the *host* and the *device*, and to send commands from *host* to *device*.



A GPU device has general computation resources (inside the module depicted as Compute Engine in the slide), but also specific resources for graphical operations, like rendering and texture generation (3D engine), apart from resources devoted to move data between the device memory and the host memory (copy engine).

This slide shows that multiple threads running on a multi-core CPU can send commands to any of the three GPU engines described so far. Those commands are queued and processed in order. The high number of resources working in parallel that incorporates the GPU device allows to overlap de execution of two or more commands. For example, all three engines can work simultaneously, and the compute engine can even execute tasks corresponding to different commands at the same time (maybe coming from different processes and users).

The program executed on the host sends commands to the GPU device driver, that is, to the software module of the Operating System responsible for managing the GPU device, which interprets the command and translates it into a series of commands that are sent to the device. There are memory-copy commands (*memcpy*) generated by the host to move data between the host and the device memories. There are computation jobs sent from the host to the device, which are called kernels (not a very lucky word). Normally the host sends a kernel to the device and then waits for it to finish its execution, but it is also possible to send several kernels asynchronously and ask the driver if the jobs have finished without having to block the execution of the host.

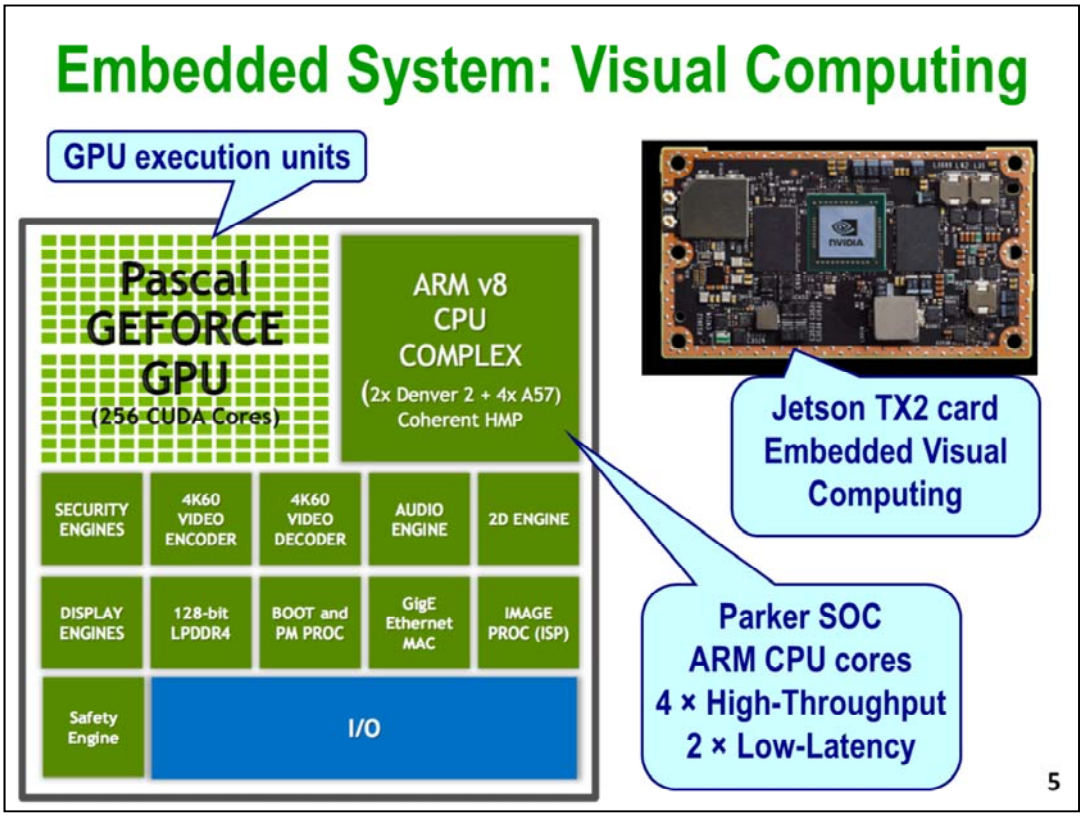


A GPU device provides very high computation performance with moderate energy consumption. This characteristic makes GPUs a very energy-efficient solution for High Performance Computing (HPC). Therefore, a good solution for building high-performance clusters or even super-computers is to populate the system with multiple GPU devices per host CPU.

The first example, on the top, shows three configurations using multiple GPUs, emphasizing the connections between the GPUs. Currently, the connection technology used by NVIDIA GPUs is called NVLink, and provides a peak bandwidth of 20 GBytes/s per link. The connection between devices and host is PCIe Gen3 x16, which provides a peak bandwidth of 16 GBytes/s.

The second example, on the left bottom, shows a configuration with two nodes. Since each node has its own system memory, then the system is called a distributed system. Communication between threads or processes running in different nodes requires explicit messages (like, for example, the messages handled by the MPI programming language). The good news regarding GPUs is that communication between GPUs can be performed directly, without the intervention of the host CPUs. For achieving that, a GPU-aware MPI system is required.

On the right bottom there is a picture of a computer rack containing several nodes, all of them populated with GPUs.



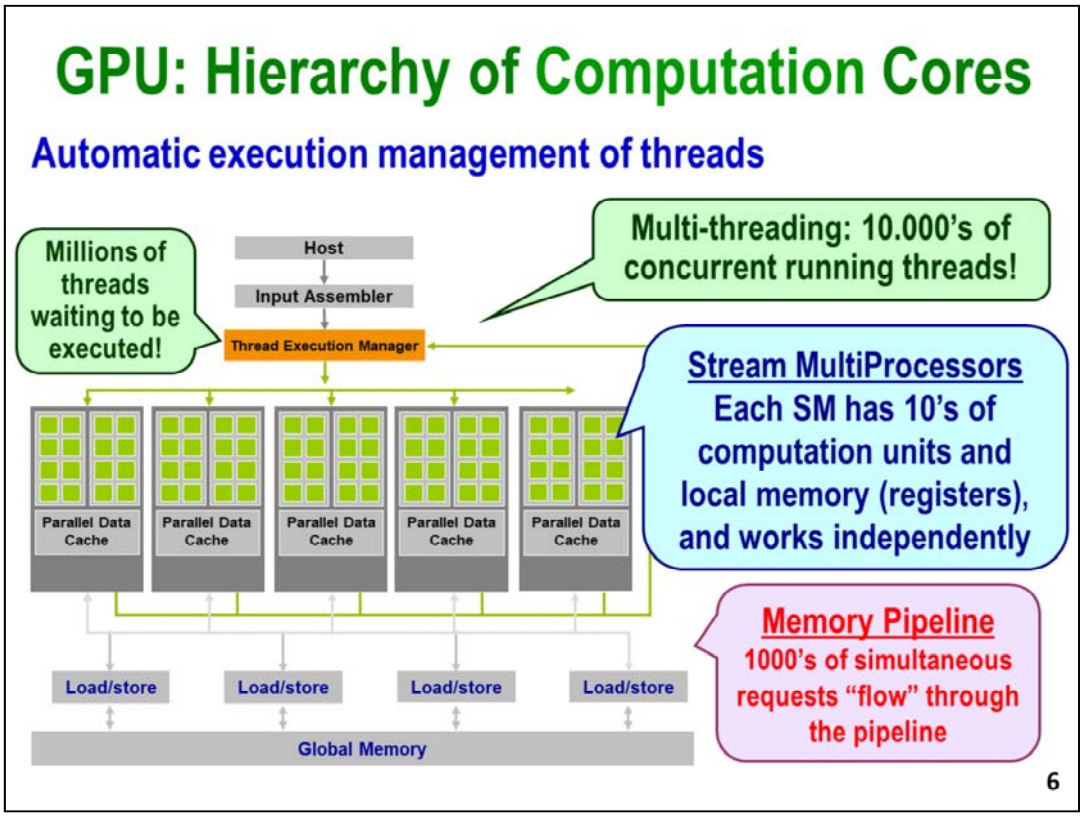
The Tegra X2 or Parker SOC is the current generation of the embedded SOCs (System On Chip) provided by NVIDIA. It is intended for embedded systems, and is an heterogeneous collection of elements, including GPU execution units (or cores), CPU execution units (or cores), and special-purpose H/W elements for video encoding and decoding, display controlling, and several communication protocols. There is a clear bet from many processor designers to support more and more powerful embedded systems to build smart low-consumption devices and integrate them in the Internet of Things (IoT).

General CPU-type computing units are needed for high-level control tasks, and for inherently serial algorithms. There are 4 ARM processor cores designed for High-Throughput applications and 2 special CPU cores aimed to reduce single-thread latency.

There are also specific DSPs (Digital Signal Processors) used for signal processing tasks, specific ASICs (Application Specific Integrated System) to do video encoding and decoding, and hardware interfaces for standard connectivity protocols (PCI, SATA, USB, CAN, Wifi, Bluetooth …) or for connection to input sensors, like cameras, radars, lidars … are also present.

A GPU accelerator provides high computation performance with very limited energy consumption. It contains GPU execution units (or cores) for the Pascal architecture, or Compute Capability 6.0. The GPU cores of an embedded system are usually less powerful than the GPU cores addressed to desktop computers or HPC systems, because they operate at around 10 watts, while high-end GPUs operate at 200-300 watts.

A promising target application for embedded GPUs is autonomous vehicles and robots: the GPUs provide the computational power required to implement the Computer Vision and Machine Learning algorithms that provide perception capabilities and intelligence to the vehicle or robot. For example, Convolutional Neural Networks (CNNs) or image depth computation are algorithms that can be implemented very efficiently on a GPU.

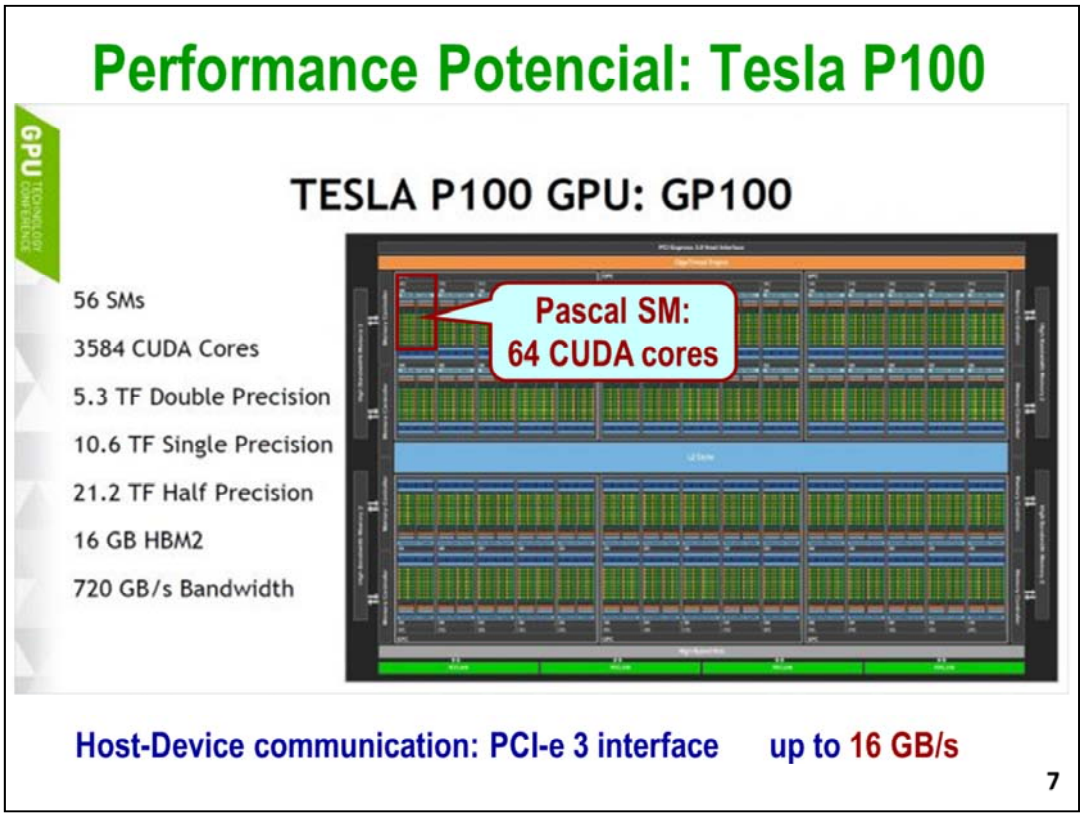


GPUs are specific-purpose processors (graphics) that have evolved to become general-purpose performance accelerators; they are coupled with traditional CPUs to form an heterogeneous system that is very appropriate for a wide range of applications. In the last years, the addition of new characteristics to the internal GPU architecture widens even more the range of applications that are suitable for GPU-acceleration.

The first remarkable issue about GPUs is that they are able to execute tens to hundreds of thousands of threads of execution simultaneously: there is space in the GPU for storing the context of such a high number of threads (hundreds of thousands of registers storing thread-private data, memory addresses, and pointers to the current instruction into the program that each thread has to execute …). A GPU program (kernel) may consists of millions to thousands of millions of threads of execution, waiting on the queues of the execution manager to be issued for executed into the GPU.

A common-sense H/W design decision is to group a subset of the compute and memory resources into elements called Stream Multiprocessors (SMs). Each SM contains multiple compute units that can simultaneously execute 10's to 100's of machine instructions selected on every clock cycle from up to 2048 threads that reside in the SM. The SM also contains different types of small and very fast memories. The consequence of this H/W hierarchy is the hierarchical parallelism exhibited by the CUDA programming model (the basic language for accessing to the full potential of a GPU).

One of the most remarkable characteristic of a GPU is that the memory system is designed to support thousands of individual accesses to different memory blocks (64 Bytes) at the same time. The memory system is like a pipeline, with memory requests sent from the SM's entering the pipeline, thousands of pending requests flowing through the pipeline, and data being delivered to the SMs at the end of the pipeline. Requests enter (and exit) the pipeline at a certain maximum issue rate, which is substantially lower than the latency of the requests. For example, tens of new requests can be made every nanosecond, while the response of the request (the data) can arrive after more than 300 nanoseconds. The result is a very high memory bandwidth (100's of GBytes per second).



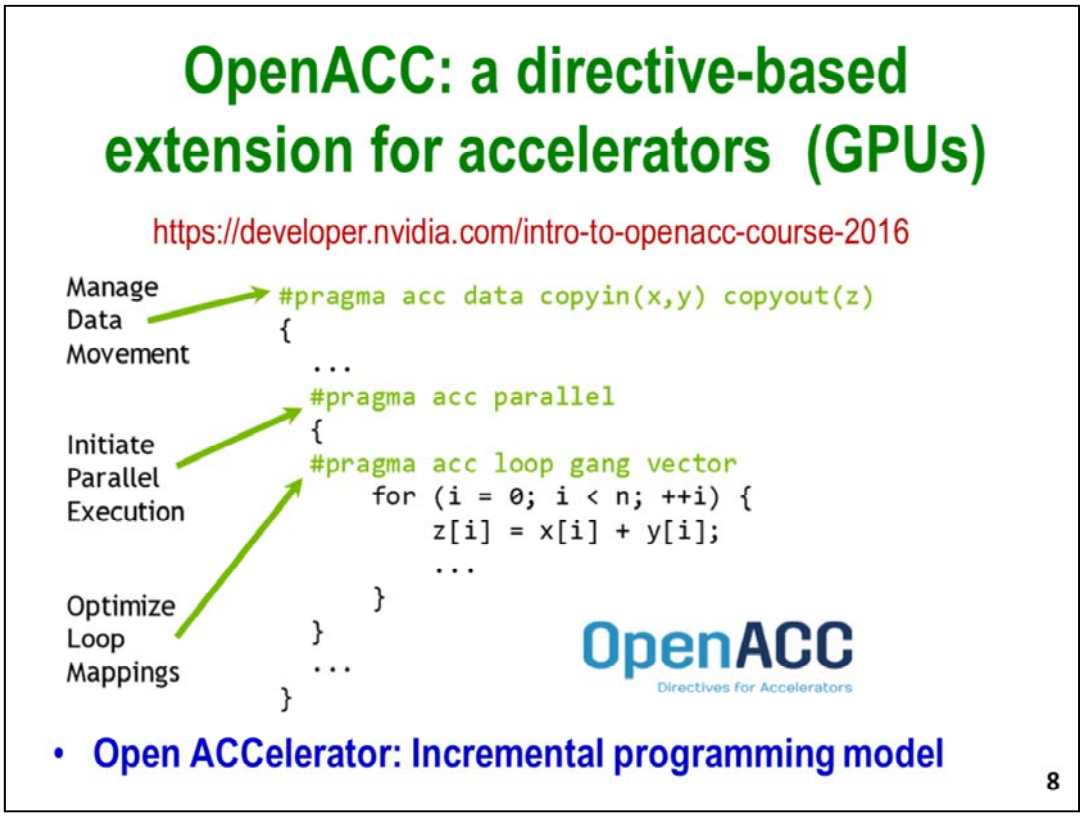
The GP 100 GPU is a recent example (late 2016) of a high-end GPU. It contains 56 SMs (Stream Multiprocessors), and each SM contains 64 CUDA cores (a total of 3584 CUDA cores in the GPU). A CUDA core is a hardware unit that performs one 32-bit compute operation per clock cycle (either using integer or floating-point numbers). The architecture of the SMs is nick-named Pascal, and also receives the more technical definition of CUDA Capability 6.0 (or CC 6.0). Different SM architectures contain a different number of computation resources and thread capabilities.

The peak performance of the GPU is 10.6 TeraFLOPs for Single-Precision operations and 5.3 TFLOPs for Double-Precision operations (Peta= 10^{15}, Tera= 10^{12}, Giga= 10^{9}, Mega= 10^{6}, Kilo= 10^{3}). This capability is similar to that offered by the most powerful supercomputer only 15 years ago, and today costs less than 3.000 Euros.

A new issue of this architecture is the support for half-precision (16-bit) floating-point operations. This kind of operations are very useful for an specific but very important application: machine learning using Convolutional Neural Networks (also know as Deep Learning). GPU cards specifically oriented towards the graphical market often have substantially lower peak performance for double-precision operations, since graphical applications generally do not need this precision.

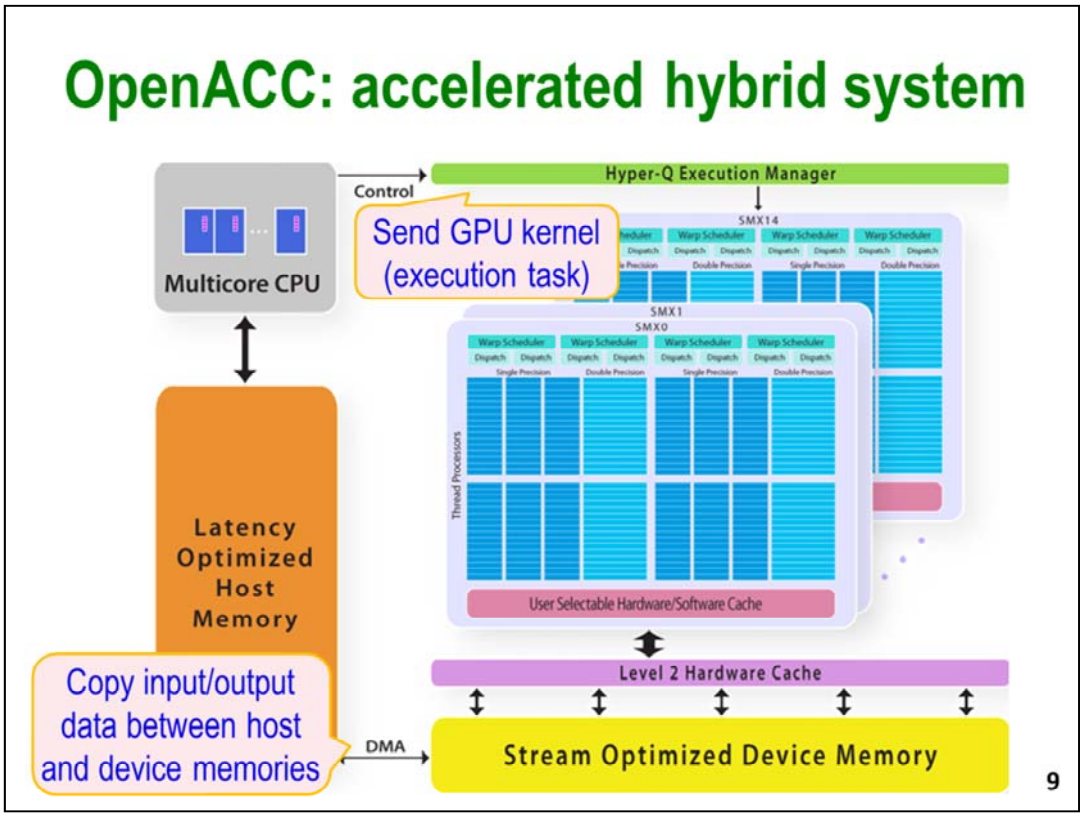
An amazing performance characteristic is a peak bandwidth with the GPU external memory of 720 GBytes/s. This is around 10 times higher than the bandwidth provided by a comparable high-end CPU. The connection between devices and host, however, has a peak bandwidth of 16 GBytes/s (assuming a PCIe Gen3 connection).

The ratio between peak compute throughput and memory bandwidth is 10,600 / 720 = 14.72 FLOPs / Byte (the lower is the better). Applications with low arithmetic intensity (ratio of compute operations with respect to Bytes read from memory) benefit from processors with a high memory bandwidth, compared to the peak computation throughput.



OpenACC is a standard for parallel programming on CPU / GPU *heterogeneous systems (Open ACCelerators*), developed by Cray, NVIDIA and PGI. Like in the case of OpenMP, the programmer annotates the source code using compiler directives (#pragma) in the C, C++ and Fortran languages in order to identify the areas of code that can be accelerated. The code specified by the programmer can be compiled to be executed in an accelerator *device* coupled to the CPU (acting as a *host*): for example the GPUs of NVIDIA and AMD, or the Xeon Phi accelerator from Intel. This code is downloaded into the *device* at run time, when the program running in the *host* invokes the execution.

The link provided in the slide is a recorded webinar (with videos and slides in PDF) that introduces the OpenACC programming language for massively parallel accelerator devices (like GPUs).

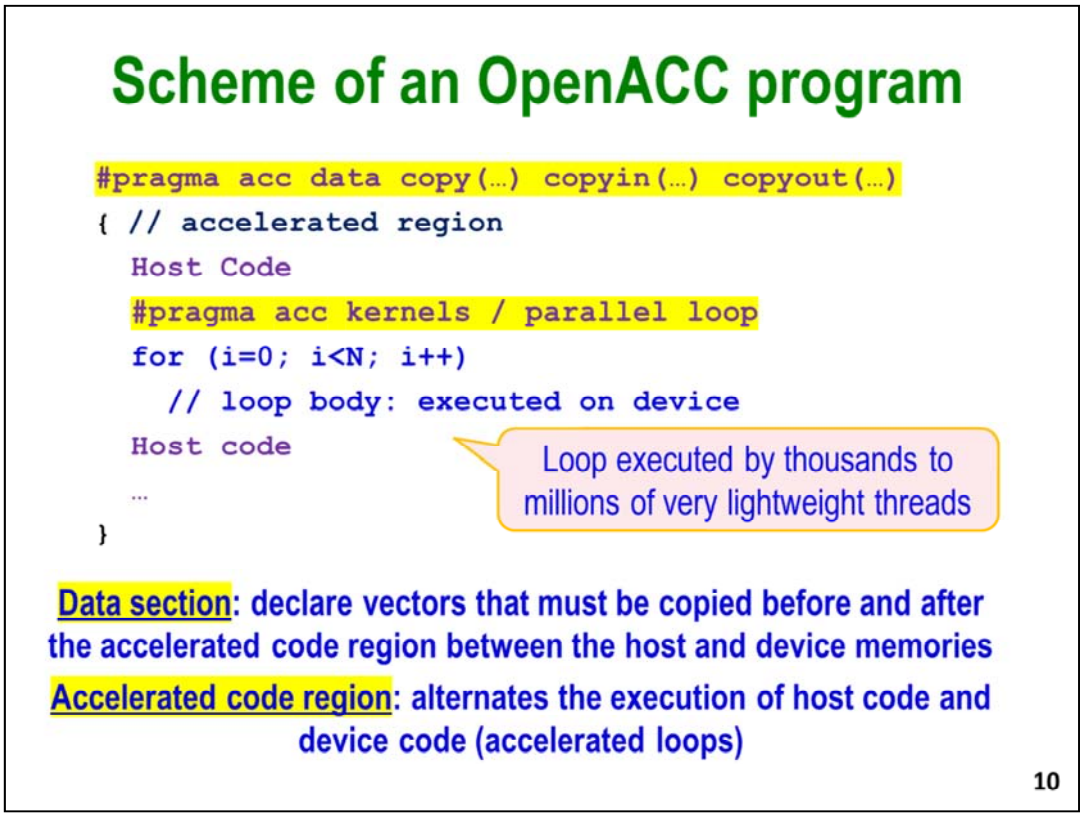


The slide shows the model of the CPU-GPU hybrid system considered by OpenACC. A program cannot be executed directly on the GPU (*device*), which is a coprocessor of a traditional CPU (*host*). Among others, the device does not incorporate the necessary H/W support to implement OS tasks that allow protection between processes (priorities) or the management of Input / Output devices using interruptions.

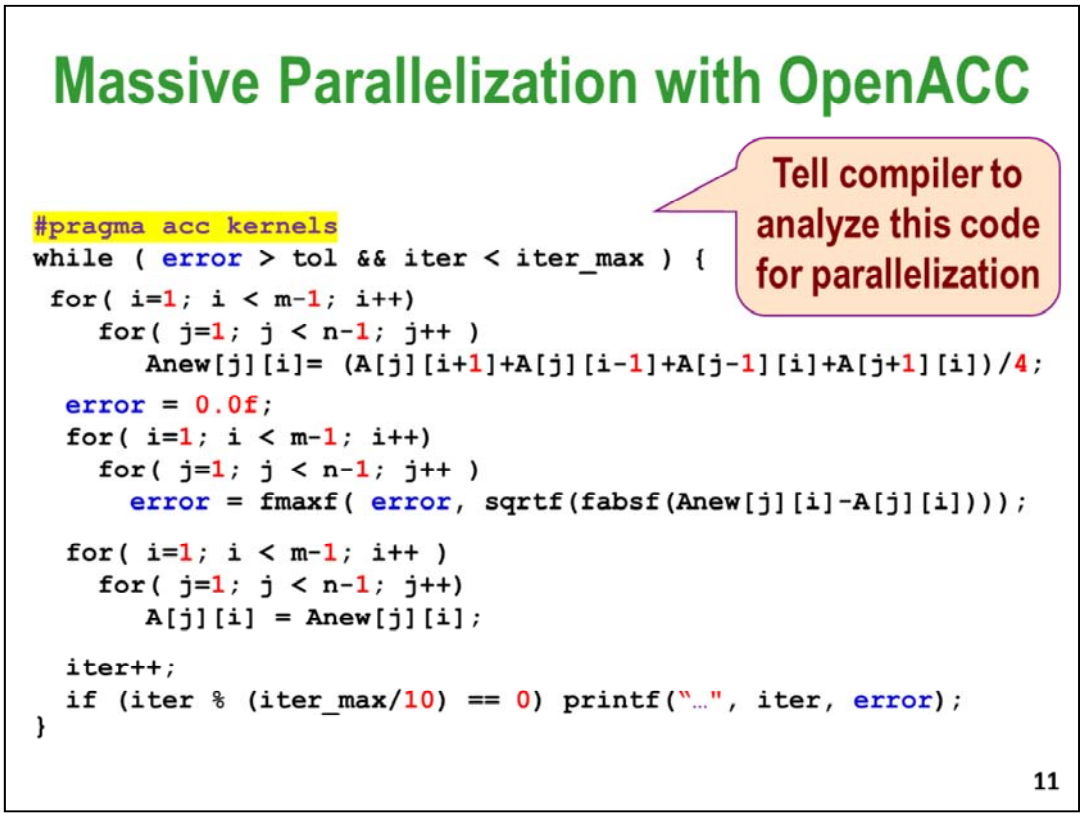
Host and *device* use to have separate physical memories, optimized for different metrics: (1) the *host* memory is optimized to reduce the access latency and offer the maximum storage capacity; (2) the *device* memory is optimized to maximize the memory bandwidth for sequential accesses and, for cost reasons, must sacrifice storage capacity and memory latency. These physical memories can be handled as separate memory spaces or can be part of a virtual shared memory known as *unified memory.* Another typical feature of many accelerators is that there are small S/W programmable cache memories: the programmer decides which variables are stored in these caches during the entire execution of a kernel. Currently, the use of the S/W cache memories is not possible using OpenACC and can only be controlled using CUDA.

A kernel running on a GPU is made up of thousands, millions, or billions of threads. A thread is a program that specifies the sequential execution of a series of instructions. All threads execute the same code, in what is called an SPMD (Single-Program Multiple-Data) model, but each thread receives a unique identifier that allows it to differentiate the input data it has to use, the results that it must generate, and even the parts of the code that must be executed or not. The threads that make up the kernel share the total work that must be done, and may have to collaborate by sharing data in common variables and synchronizing explicitly.

As shown in the figure, the millions of threads that compose the kernel wait in a general queue (Hyper-queue Execution Manager), before being assigned to one of the computation modules of the device (or Stream Multiprocessor or SMs). Each SM has its own queue associated with thousands of threads competing to execute instructions using the SM's compute resources. Once a group (block) of threads is assigned to a SM, the threads can no longer be evicted, and remain in the same SM until their execution is completely finished. NVIDIA GPUs group threads from 32 to 32 to run in SIMT (Single-Instruction Multiple-Threads) form, that is, synchronized. This scheme represents a different way of programming what we know as SIMD execution and the use of vectorized instructions.



The slide shows the structure of an OpenACC program: (1) data sections (first line) indicate which data has to be moved between *host* and *device*, and also command the reservation of memory in the *device*; (2) parallel directives control the regions and for loops that can be parallelized for execution in the *device*.

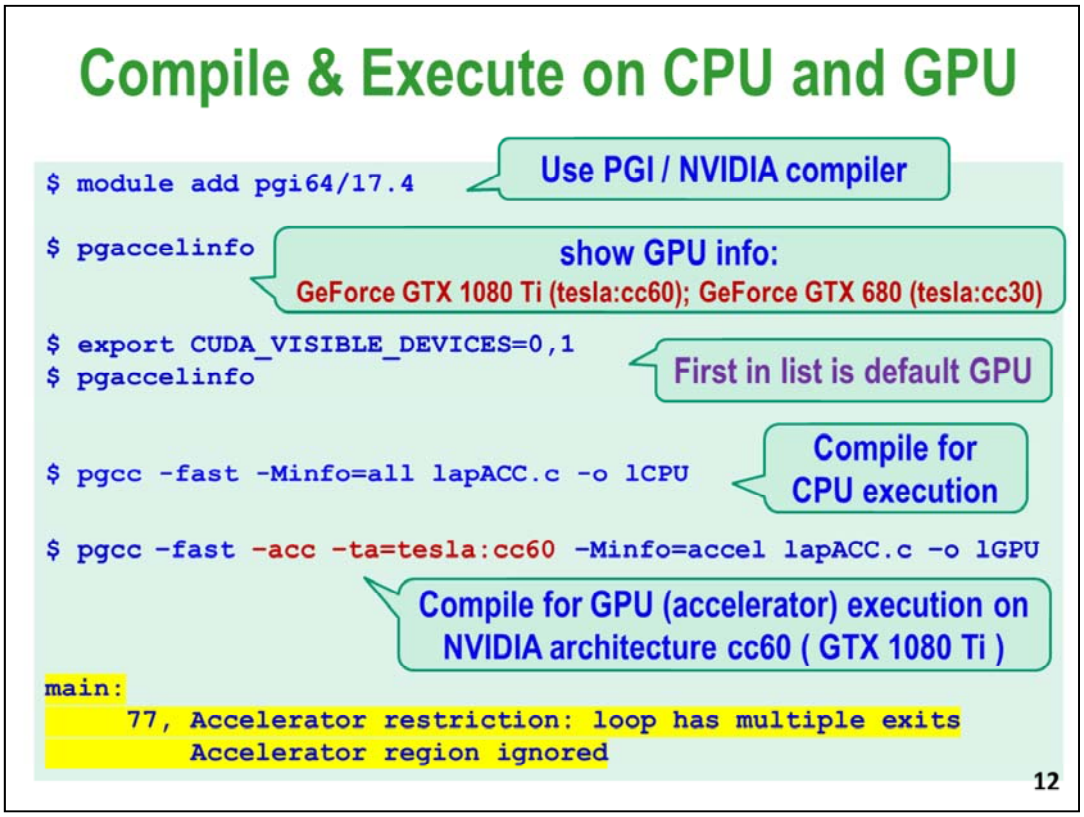


This slide shows a first attempt to accelerate on a GPU the execution of the 2D Laplace program using OpenACC. The "`#pragma acc kernels`" directive affects only the following statement, which in this case is the while loop statement. The intention of the directive is to tell the compiler to analyze all the code in the while loop. Basically, the compiler does all the work of parallelizing the code, but previously the compiler needs to "understand" the code in the annotated accelerated region:

(1) deduce which are the data structures (vectors, matrices, scalar variables) involved and how they read and/or written

(2) Identify the loops that can be potentially parallelized and check if they involve independent loop iterations or if there are recurrent data dependences that prevent the parallel execution of the loop iterations

The next step is to compile the code with an OpenACC-aware compiler.



Not all the compilers are OpenACC-aware: for example, the gcc compiler will ignore the "`#pragma acc kernels`" directive and generate executable code exclusively for CPU.

We need to use a new compiler, the free PGI compiler that has been recently acquired by NVIDIA, one of the leaders in the manufacturing of GPUs. In the laboratory, the compiler must be configured using the **`module`** utility each time a new terminal is created, as shown in the slide. If you install the compiler in your own computer you do not need to use the **`module`** utility for configuring the compiler environment.

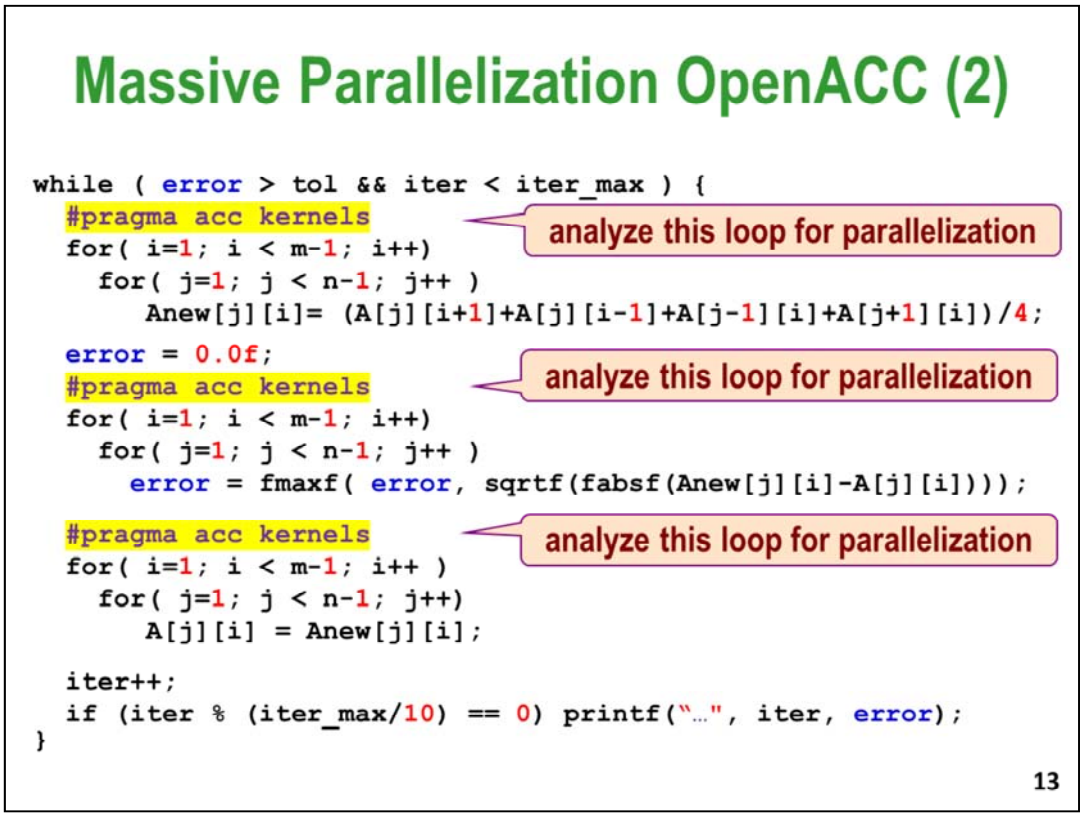
The command **`pgaccelinfo`** provides information about the GPU cards installed (and visible) in the computer. The environment variable **`CUDA_VISIBLE_DEVICES`** can be used to declare the numerical order of the GPUs (0, 1, 2 …), and to make the GPUs visible or not visible. The first GPU in the list is the one that will be used by default when running an application requesting a GPU. This is important, since the GPUs in the laboratory belong to different architectural families (or Compute Capabilities) and older architectures cannot execute code compiled for newer architectures.

We show the commands used to generate code for CPU execution and code for GPU execution (**`-acc`**). The flag **`-ta=tesla:cc60`** indicates the compute capability of the destination or target GPU. The GPUs in the laboratory computers mostly have a 2.0 compute capability. The examples shown in these slides use the aolin21 machine, containing a very new and powerful GTX 1080 Ti GPU card.

The flag **`-Minfo=accel`** is very useful to obtain information about how the compiler generates the code for GPU. The flag **`-Minfo=info`** provides information for the code generated for CPU.

Sometimes you need to execute the **`nvidia-smi`** command to check all the GPUs available in your computer. This command is configured by adding the environment for the CUDA tools, using the module utility: **`module add cuda/7.5`**

The PGI compiler complains about the OpenACC directive: the while loop is not a valid construct that can be massively parallelized for GPU execution. We must look for a different alternative …



The slide above shows a different approach, where the "`#pragma acc kernels`" directive is placed just before tall he three **`for`** loop statements in the code. Notice that the directive is applied to the outermost loop, but affects all the nested loops inside the outermost loop. In the next slide we will analyze the effect of these directives.

An alternative way of achieving the same results is as follows, where the code of the three internal loops is grouped into a compound statement (inside brackets {…}). Notice that the **`if`** statement must be moved outside the scope of the **`#pragma`** directive or otherwise the compiler will complain. The "**`iter++`**" statement can be put inside the scope of the directive, but its execution will be performed on the host side.

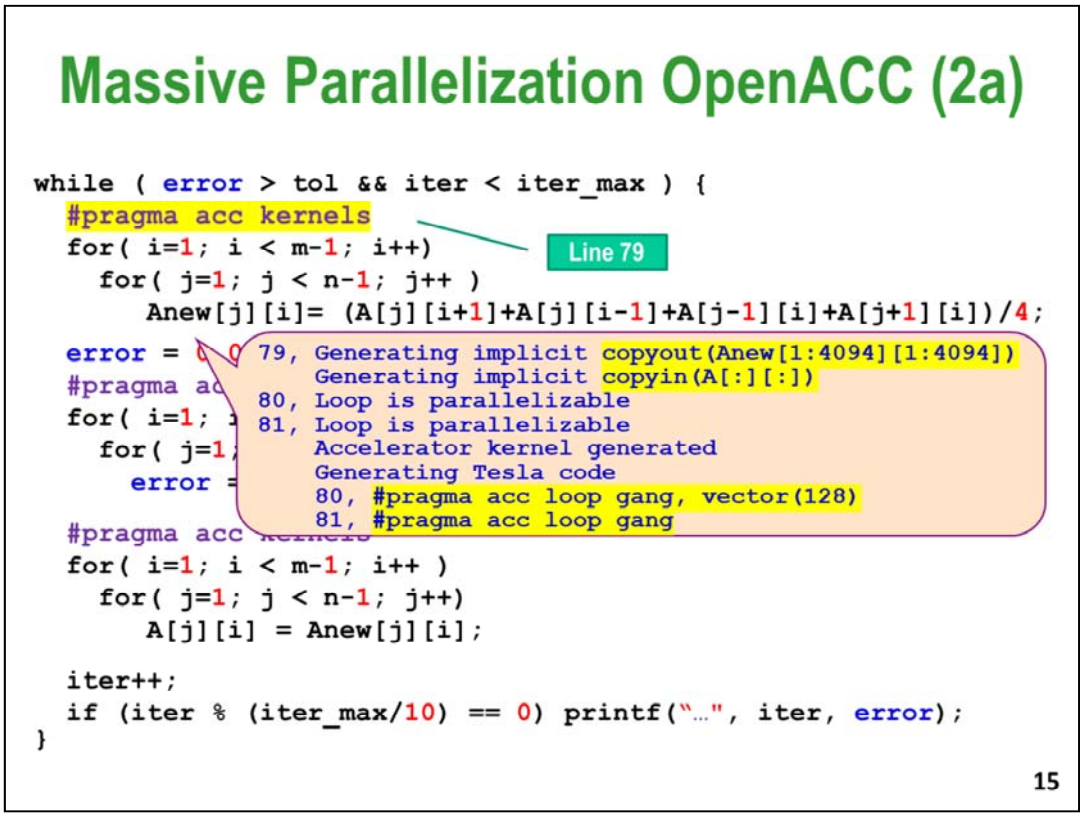
```
while ( error > tol && iter < iter_max )
{
  #pragma acc kernels
  {
    for( i=1; i < m-1; i++)
      for( j=1; j < n-1; j++ )
         Anew[j][i]= (A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i])/4;
    error = 0.0f;
    for( i=1; i < m-1; i++)
      for( j=1; j < n-1; j++ )
        error = fmaxf( error, sqrtf(fabsf(Anew[j][i]-A[j][i])));
    for( i=1; i < m-1; i++ )
      for( j=1; j < n-1; j++)
        A[j][i] = Anew[j][i];
    iter++;
  }
  if (iter % (iter_max/10) == 0) printf("…", iter, error);
}
```

Compile on GPU

```
$ pgcc -fast -acc -ta=tesla:cc60 -Minfo=accel lapACC.c -o lGPU
main:
     79, Generating implicit copyout(Anew[1:4094][1:4094])
         Generating implicit copyin(A[:][:])
     80, Loop is parallelizable
     81, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         80, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         81, #pragma acc loop gang /* blockIdx.y */
     85, Generating implicit copyin(Anew[1:4094][1:4094],A[1:4094][1:4094])
     86, Loop is parallelizable
     87, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         86, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         87, #pragma acc loop gang /* blockIdx.y */
         88, Generating implicit reduction(max:error)
     90, Generating implicit copyin(Anew[1:4094][1:4094])
         Generating implicit copyout(A[1:4094][1:4094])
     91, Loop is parallelizable
     92, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         91, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         92, #pragma acc loop gang /* blockIdx.y */
```

14

This is the full output from the compiler. It indicates which code has been accelerated and which host-device data movement tasks have been inserted on the program's execution. The line number of the statements is listed. Next we will describe the optimizations one by one.

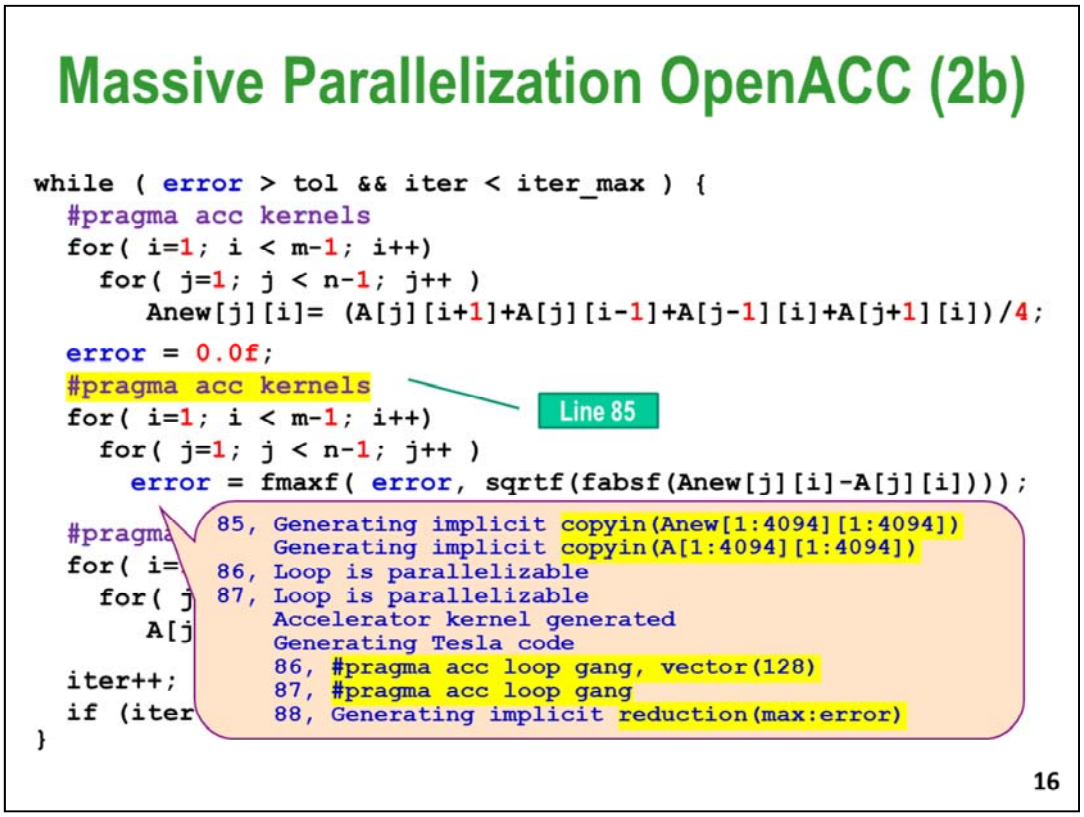


The code behind the pragma in line 79 (the first nested **`for`** loop) has been accelerated, as shown in the log text inside the box.

First, explicit copies between host and device have been arranged: the full contents of matrix A are copied from host to device before executing the accelerated code (**`copyin(A[:][:])`**), and the contents of matrix Anew except for the first and last rows and columns are copied from device to host after executing the accelerated code (**`copyout(Anew[1:4094][1:4094])`**).

Then, the outermost loop (in line 80) is identified to be parallelizable and therefore it is accelerated, using gang and vector parallelism, as shown by the directive **`#pragma acc loop gang, vector(128)`**. This directive is not completely syntactically correct and then cannot replace the kernels directive. Vector parallelism is like SIMD parallelism (Single-Instruction Multiple Data) and indicates that threads work tightly coupled and synchronously, as if each thread was a lane of a vector instruction. Gang parallelism is like MIMD parallelism (Multiple-Instruction Multiple Data): decoupled and asynchronous threads, executing at different times, maybe without overlapping their execution.

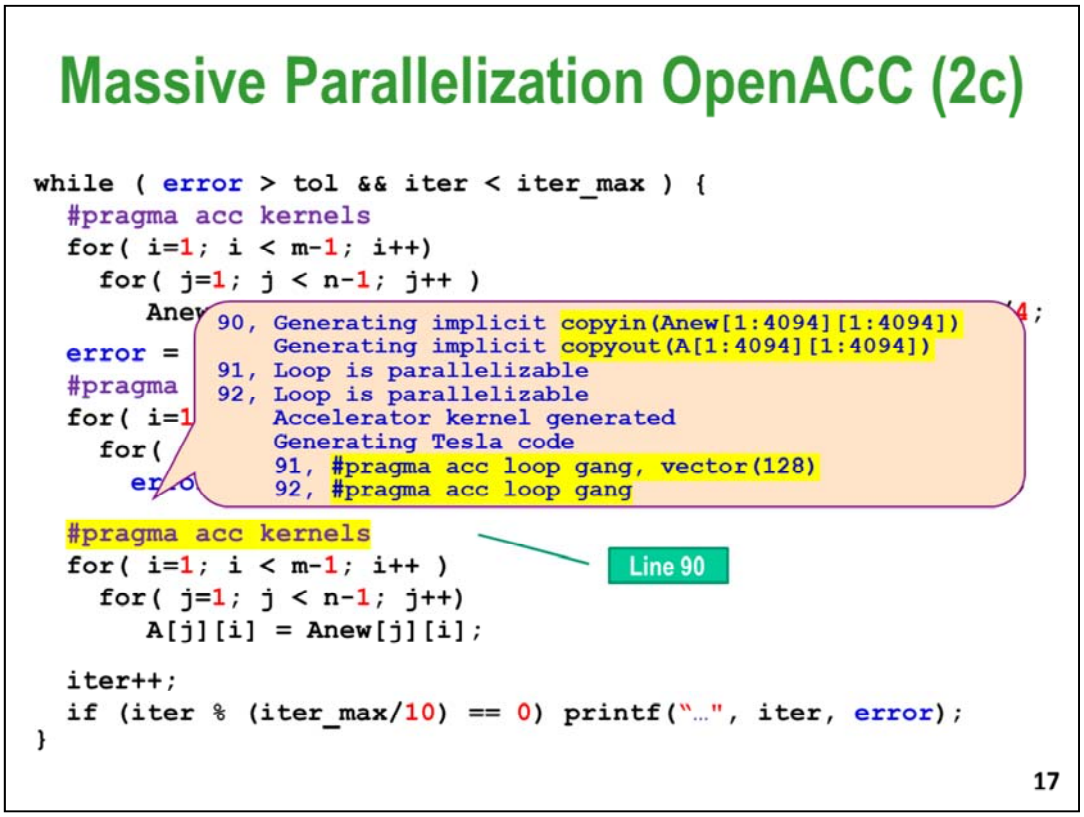
Finally, the innermost loop (in line 81) is also identified to be parallelizable and it is accelerated, using only gang parallelism, as shown by the directive **`#pragma acc loop gang`**. This directive is syntactically correct and can be place before line 81, but only if line 80 uses a **`parallel`** directive.



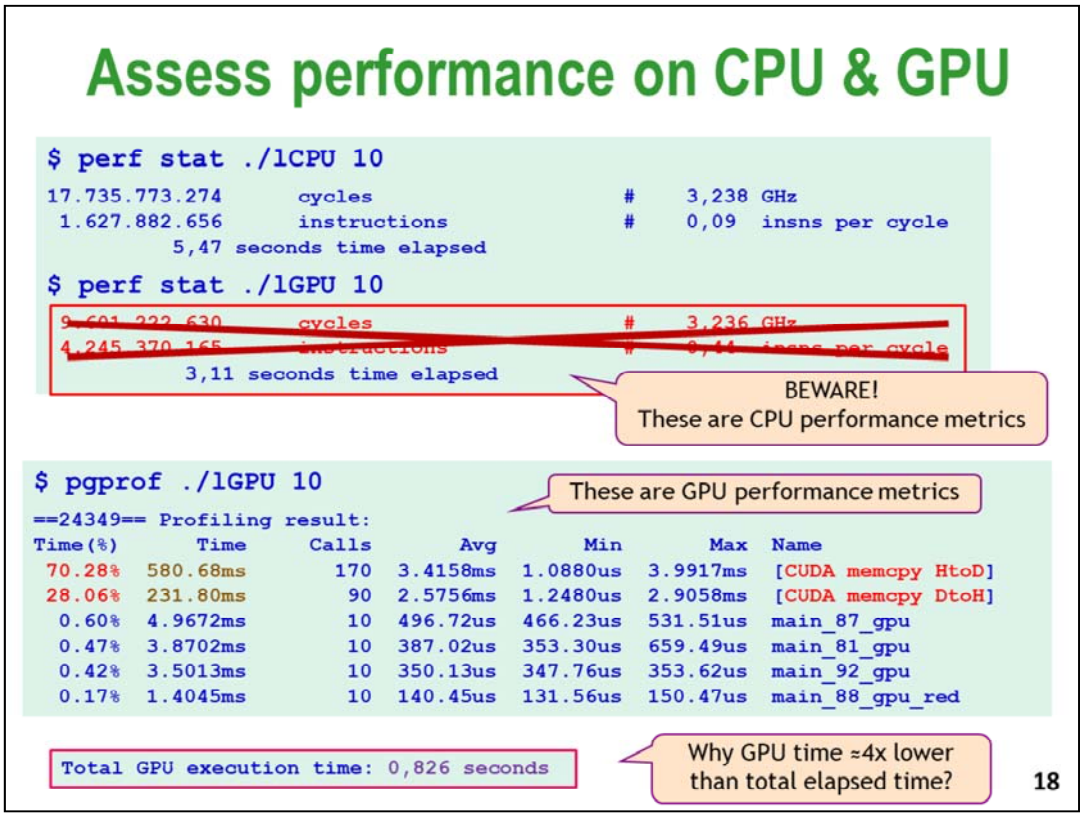
The code behind the pragma in line 85 (the second nested **`for`** loop) has also been accelerated, as shown in the log text inside the box.

Again, explicit copies between host and device have been arranged: all the contents of matrix A and Anew, except for the first and last rows and columns, are copied from host to device before executing the accelerated code (**`copyin(A[:][:])`**, **`copyin(Anew[1:4094][1:4094])`**). Since the accelerated loop does not modify those two matrices, there is no **`copyout`** directive. Instead, the value of the **`error`** variable is copied in and out, before and after the execution, but this is not shown explicitly in the log, but implicitly in the **`reduction(max:error)`** directive at the end of the text box.

The outer and inner loops (lines 86 and 87) are parallelizable and are accelerated, using gang and vector parallelism in the same way as for the previous accelerated loop (described in the previous slide). The compiler identifies a reduction operation on the accelerated code that computes the error (see the text for line 88) which involves the creation of an additional accelerated code that must reduce (compute the maximum of) the partial results of each of the gangs to generate a single reduced value. As explained before, this value must be copied in and out to allow the host code to check the actual error of the convergence loop and decide if the program must continue or not.



The code behind the pragma in line 90 (the third nested **`for`** loop) has been accelerated in the exact same way as the first nested loop. Now the roles of matrices A and Anew are reversed when copying data between host and device (Anew is copied in, and A is copied out).



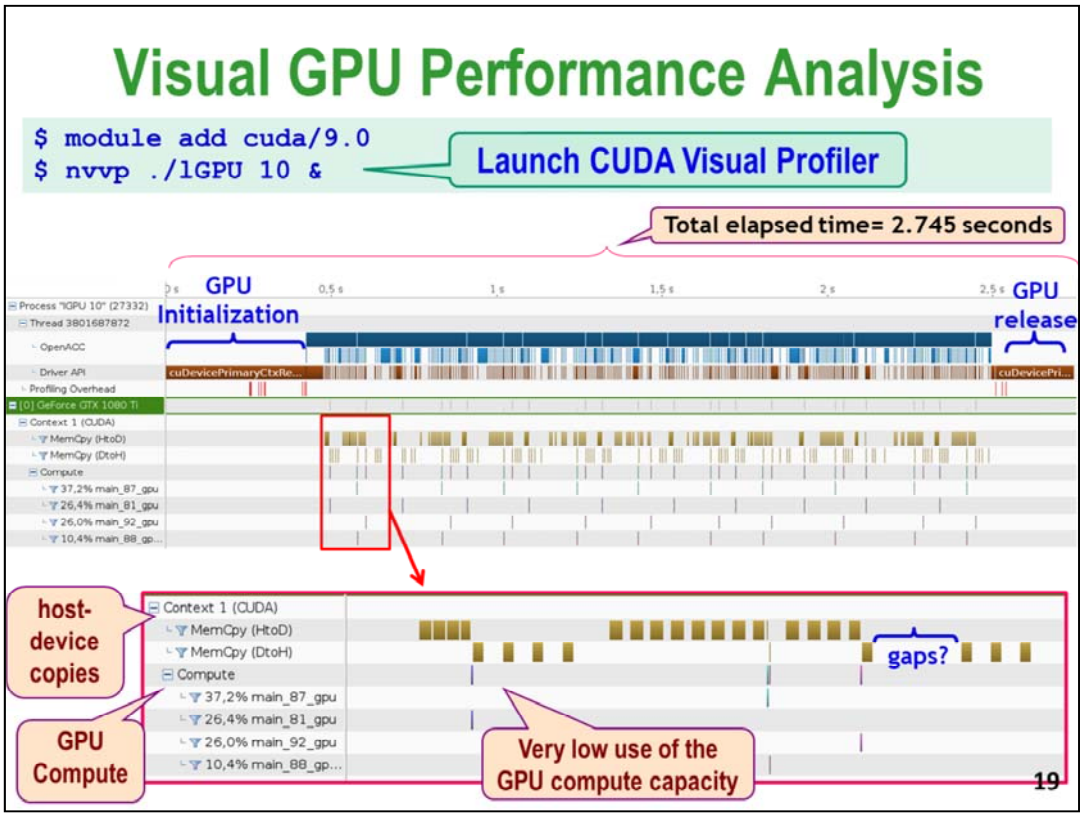
The **`perf`** tool measures execution performance only on the host part (CPU). Therefore, **`perf`** can only be used to measure the total elapsed time of the accelerated code. The **`pgprof`** tool (on the PGI compiler toolset), or the **`nvprof`** tool (on the CUDA toolset) can be used to profile the execution on the device. The slide shows that six types of tasks are performed on the GPU:

- two types of data memory copies ("*memcpy HtD*" means memory copy from Host to Device, while "*memcpy HtD*" means memory copy from Device to Host). 170 memory copy tasks have been performed from Host to Device and 90 memory copy tasks from Device to Host.
- four different accelerated codes (kernel) are executed, for lines 81, 87, 88 and 92. The codes for lines 81, 87 and 92 correspond to the three nested loops, while the code for line 88 is the special reduction code for computing the error. Each kernel is executed 10 times, which corresponds to the 10 iterations of the convergence loop that are specified as an input parameter in the command line.

We notice two main conclusions from this results:

1. Most of the GPU execution time is spent on copying data in and out (almost 99%)
2. The aggregated time of the GPU tasks is around 25% of the total elapsed time of the execution

We can understand the reasons for those results by exploring the Gantt Diagram provided by a visual profiling tool …



The NVIDIA Visual Profiler (**nvvp**) tool provides many information that is very useful to understand the performance of an application running on a GPU. The first tool is the Gantt Diagram of the execution, which identifies the tasks executed on the Host and the Device, identifies Memory copy tasks and Compute tasks, and provides the elapsed times for all those tasks and kernel executions.

The commands needed for using the tool are the following (it is contained on the CUDA toolset; the PCs in the laboratory are too old and cannot run version 9.0 of the CUDA toolset).

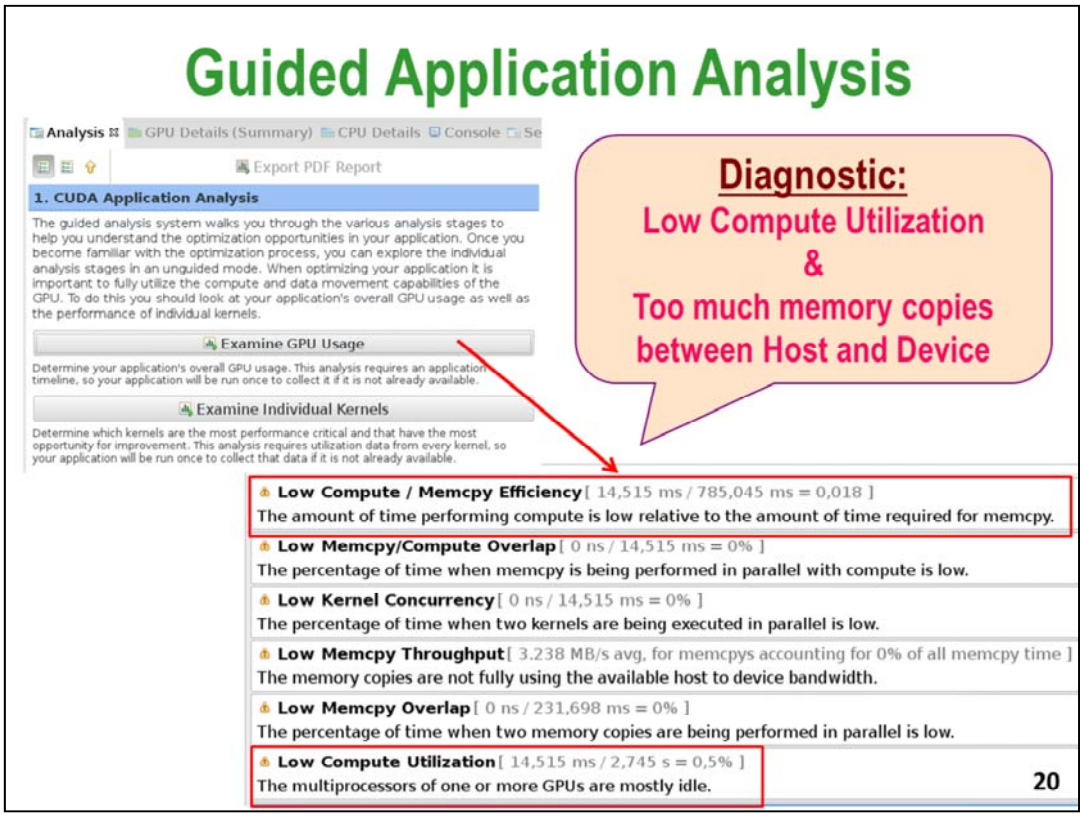
```
$ module add cuda/7.5    // use this version on the LAB
$ nvvp ./lGPU 10 &       // launch program and return
```

The diagram on top shows the whole execution, which indicates that the computation on the GPU takes barely 1% of the total execution time (or elapsed time). Most time is devoted to data movement between host and device. Also, the first 0.4 seconds of the execution are used by the host CPU to do the GPU initialization.

The slide shows on the bottom a zoom of a region dominated by data copies from host to device. You can have complete information of the performance metrics of the memory data transfer (elapsed time, amount of data, and average bandwidth) by clicking on a memory task in the diagram.

From the visual inspection of the Gantt's Diagram it is fairly straightforward to diagnose that there is too much unrequired data movement between host and device. Also, the initialization and finalization stages of the execution are responsible of a significant part of the time.

The next slide shows how to make the visual tool to provide a similar diagnostic of the program's execution



There is an option called “guided analysis” that examines the GPU usage and provides different metrics that can be useful for identifying possible performance bottlenecks and recommendations for solving them. The first step consists in doing a performance analysis of the whole execution.

The most relevant results are the ones with a red square: (1) there is too much time devoted to data movement between host and device in comparison to the amount of time devoted to computation; and (2) the GPU resources (multiprocessors) are almost idle.

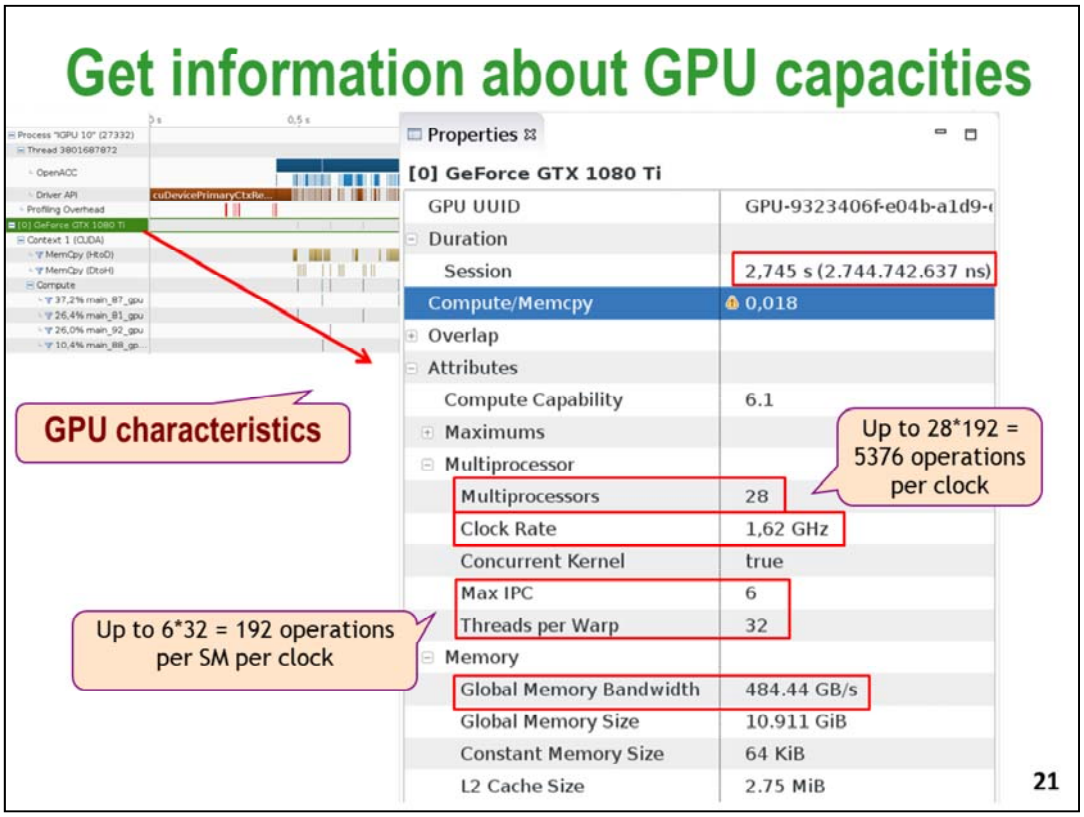
The first metric (*compute / memcpy efficiency*) suggests that memory copies between host and device should be reduced to a minimum, compared to the time needed for computation. In fact, we will soon show how to modify the code to reduce this data movement.

The last metric (*GPU compute utilization*) indicates that the time for computing on the GPU is a very small fraction of the elapsed time. This could be fixed by increasing the total number of iterations of the convergence loop.

The second metric (*memcpy / compute overlap*) and fifth metric (*memcpy / memcpy overlap*) suggests that if the amount of memcopies cannot be reduced, at least they must be overlapped either with computation on GPU or with other memcopies (some GPUs allow copying in both directions (DtH and HtD) simultaneously.

The fourth metric (*memcpy throughput*) evaluates if the data transmissions reach an acceptable bandwidth. The message is very misguiding, since some copies achieve a very few bandwidth (around 3 MB/s, compared to a peak sustainable bandwidth higher than 4 GB/s), but these copies account for almost 0% of the total time spent on doing memcopies.

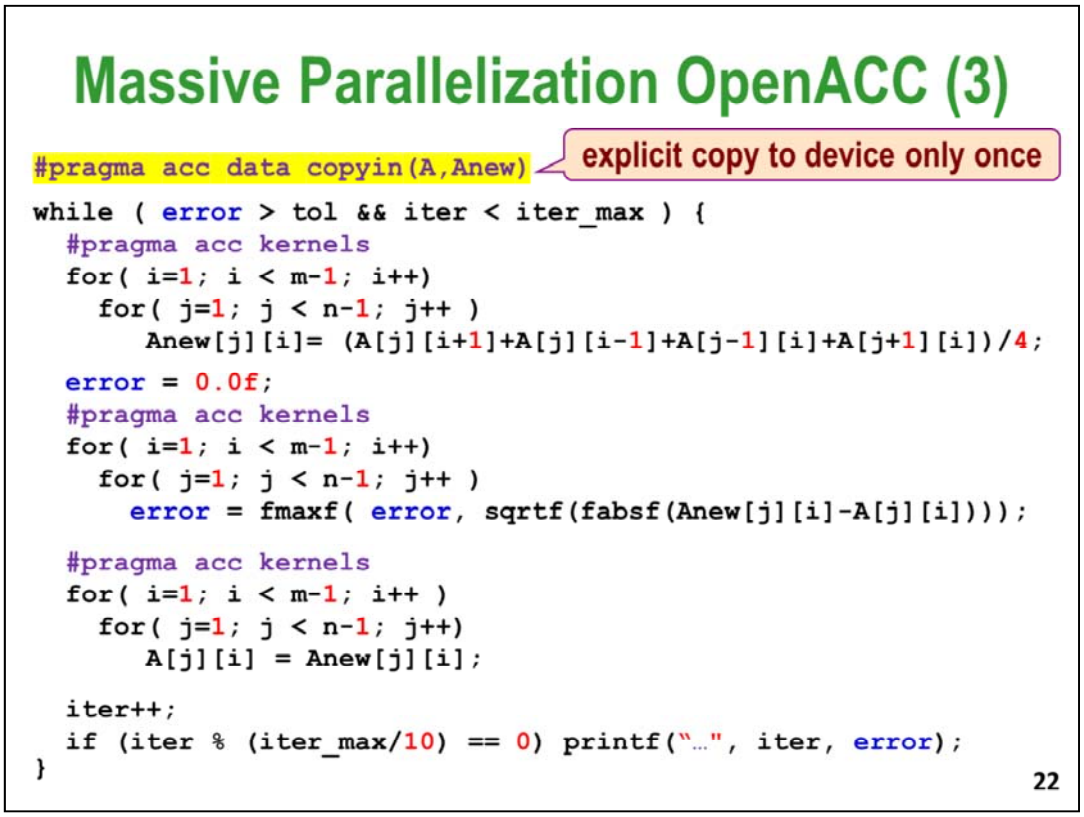
The third metric (*kernel concurrency*) suggests that overlapping the execution of several computation kernels could provide higher performance. This is true only for some cases, and it is not relevant for this application.



The GPU characteristics can be checked by clicking on the GPU name on the left side of the Gantt's Diagram.

Beware: the GPU shown here (plugged into the aolin21 computer) is very powerful compared with the GPUs that you will find in the laboratory. The most relevant numbers are:

- Up to 5376 operations per clock cycle (at 1.62 GHz corresponds to 8.7 TFLOPs)
- A peak device memory bandwidth of 484 GB/s



From a deep analysis of the algorithm one can infer that it is not necessary to copy the data in the matrices A and Anew between host and device on every execution of a GPU kernel and on every iteration of the converge loop. The data of the matrices need only to be copied from host memory to device memory before the convergence loop, and from device memory to host memory after the convergence loop. The convergence loop can be implemented by executing accelerated kernels exclusively on the device and reading and writing data only from or to the device memory. The only data that must be interchanged between host and device during the execution of the convergence loop is the estimated error.

The pragma `#pragma acc data copyin(A,Anew)` has been inserted in the program before the convergence loop to move the contents of both matrices from host to device before (`copyin`) the convergence loop. Since the host code does not use the final result in the matrices it is not necessary to copy back the resulting matrices from device to host. This can be done by changing the clause `copyin` by `copy`.

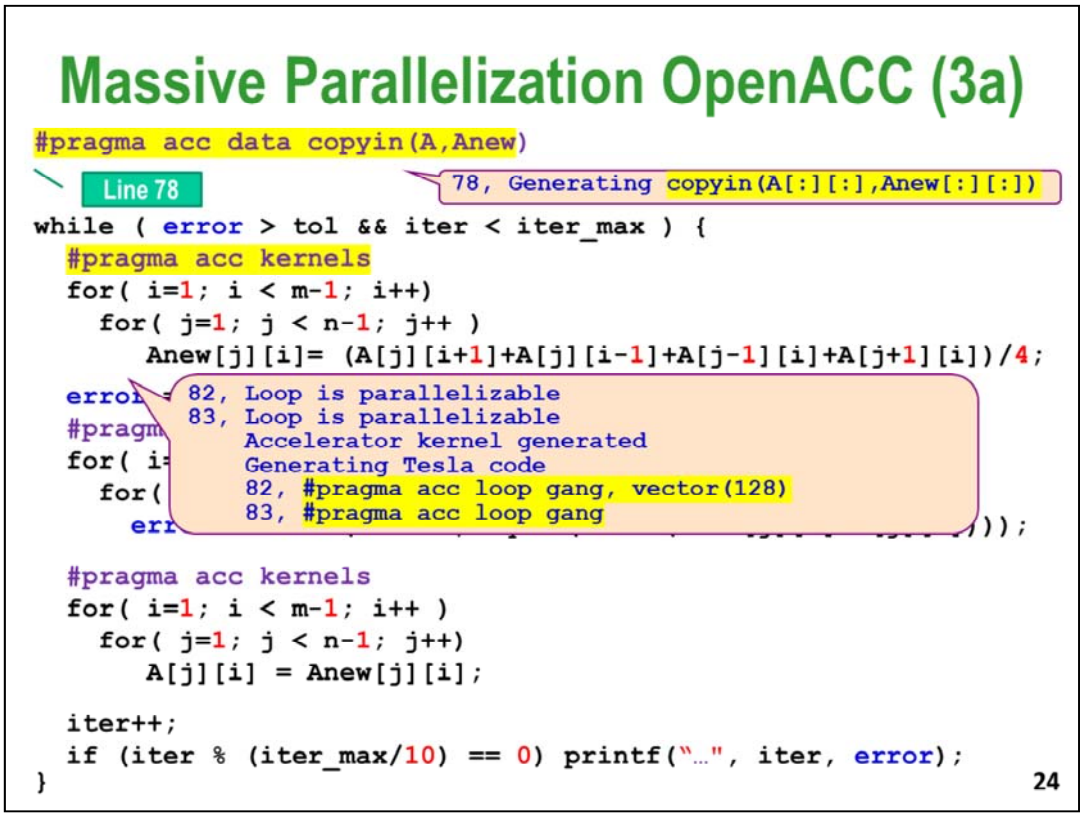
Let's analyze the log text provided by the compiler to understand how the executing code is now performing.

Compile on GPU

```
$ pgcc -fast -acc -ta=tesla:cc60 -Minfo=accel lapACC.c -o lGPU2
main:
     78, Generating copyin(Anew[:][:],A[:][:])
     82, Loop is parallelizable
     83, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         82, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         83, #pragma acc loop gang /* blockIdx.y */
     88, Loop is parallelizable
     89, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         88, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         89, #pragma acc loop gang /* blockIdx.y */
         90, Generating implicit reduction(max:error)
     93, Loop is parallelizable
     94, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         93, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         94, #pragma acc loop gang /* blockIdx.y */
```

23

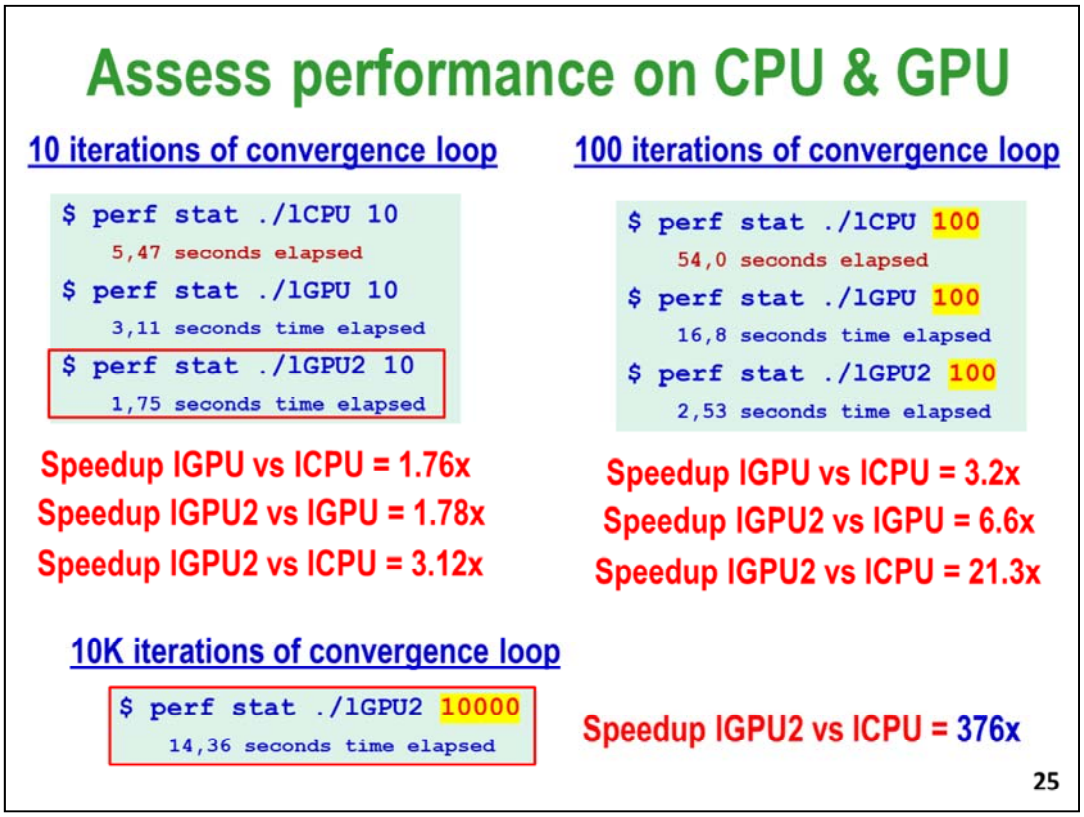
Again, the result from the compiler indicates which GPU tasks are arranged, but it is better to explain these results by analyzing also the code.



The first pragma, on line 78, does what it is expected to do.

The second pragma, on line 81, produces the same computation kernel as before, but now the compiler identifies that the data required by the kernel is already in the device memory and then skips the memory copies from host to device.

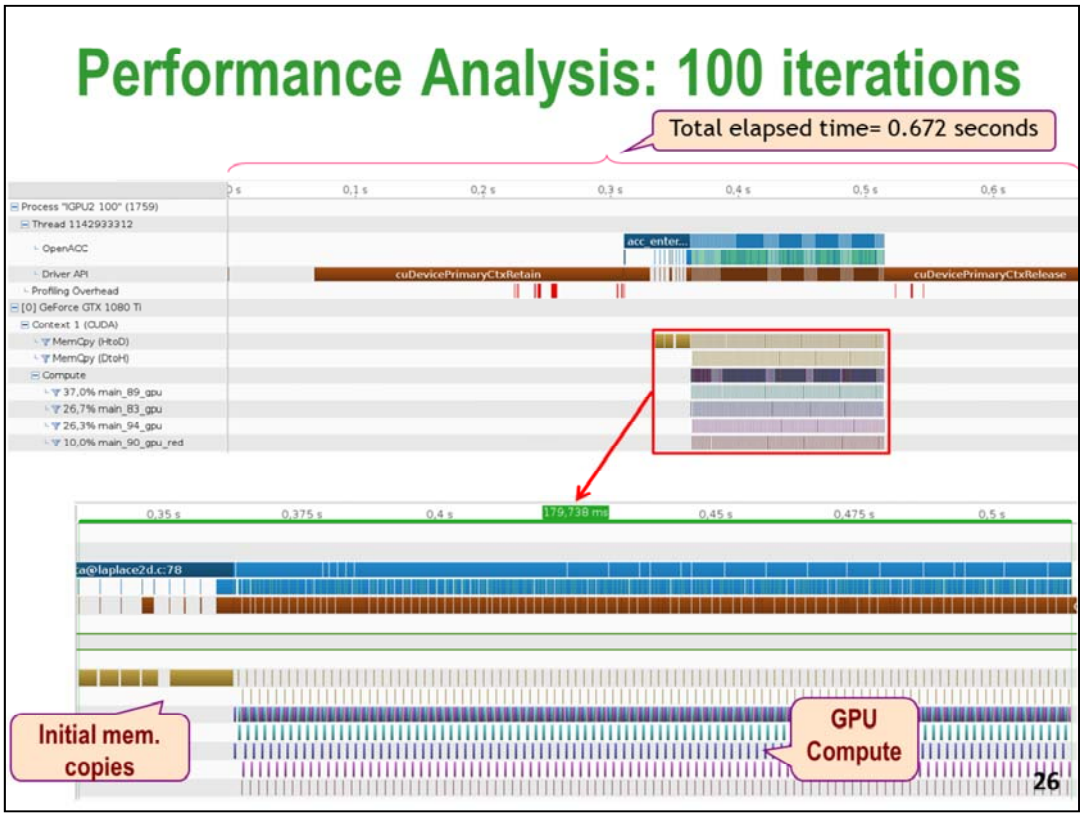
The same happens for the remaining loops: the kernel completely removes any memory copy that was performed inside the convergence loop. Only the data movement associated to the error variable needs to be moved in and out of the device memory. There is no explicit message by the compiler to indicate this movement, since the error variable is associated with a reduction clause.



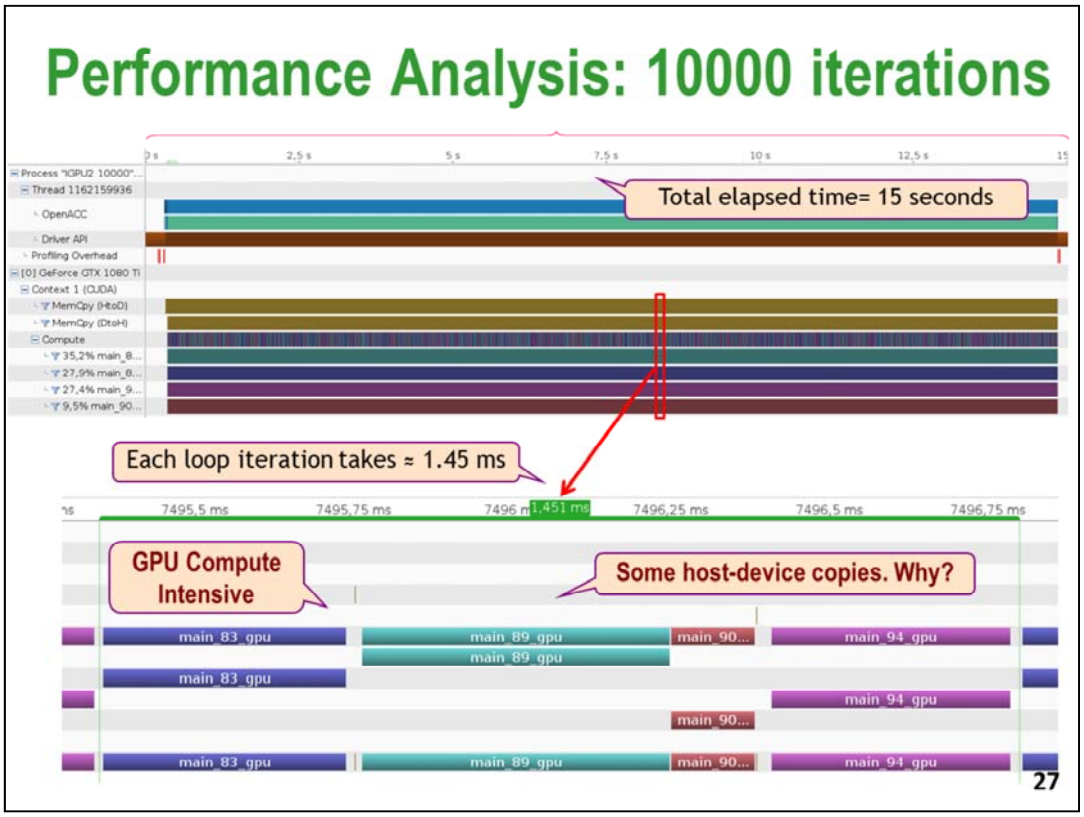
This slide shows some performance results of the three program versions (CPU-only, GPU version 0, and GPU version 1). On the upper-left there are results for a convergence loop of 10 iterations. The performance speedup of the best GPU-accelerated version compared to the single-core CPU version is 3.12x. This is apparently a good result … if you do not know the real potential of the GPU.

When the problem size is increased to 10,000 convergence loop iterations then the full potential of the GPU is revealed, reaching a performance speedup of 376x. Notice that the CPU code is not optimized and can be improved, but not so much.

The best way (and the highly recommended way) to assess the performance of the GPU-accelerated code is by profiling the execution. Let's do it.



When the program performs only 100 iterations of the convergence loop, most of the time is still spent on GPU initialization. Also, the initial copy of data from host to device takes a significant amount of time. Therefore, it is clear that the potential of the GPU requires a bigger problema size.

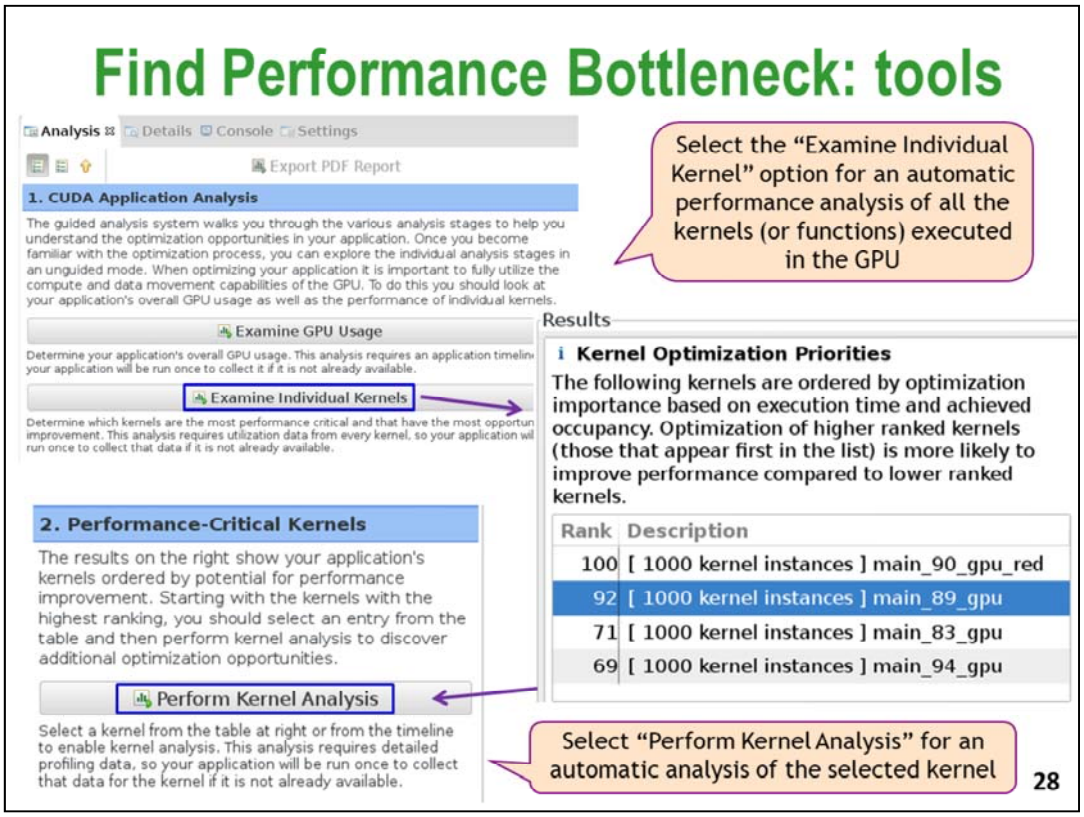


10,000 convergence iterations is enough to make the initialization and ending stages almost neglectable, and make the GPU execution time to dominate the whole program's execution time.

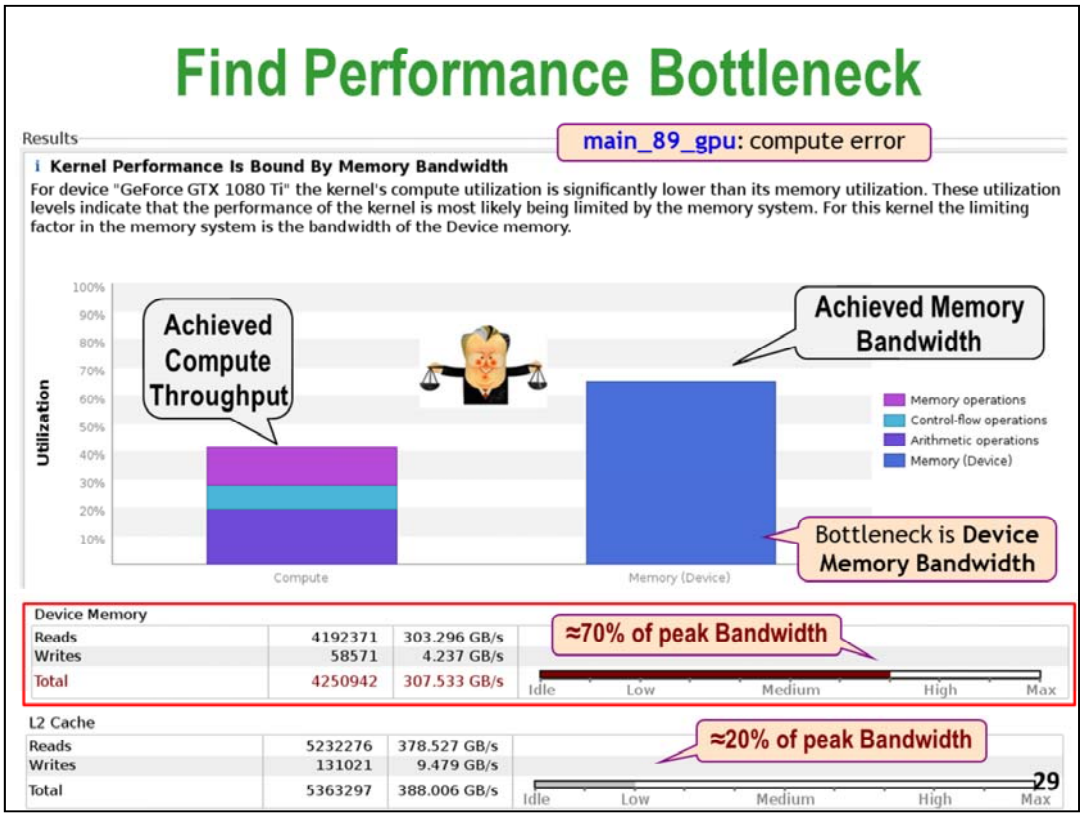
The detailed execution fragment (bellow) identifies the tasks done by the GPU for each of the convergence loop iterations:

- 4 kernel functions (compute) are executed, three of them take most of the time, and the most time-consuming corresponds to the code in line 89, which is the code computing the error (reading data from both matrices and computing the square root arithmetic operation).
- two data movements (first from host to device and then from device to host) are performed, but they take very few time. In fact, if we click on those rectangles we can check that only 4 Bytes are transferred, which correspond to the variable error.

Very few time is spent between GPU executions and, then, the most promising way to further improve performance is by reducing the execution time of one of the four kernels. But in order to make changes it is worth understanding the performance bottleneck for each kernel.



The slide shows the steps to analyse the performance of the kernels' execution (the compute part): before making changes in the code it is worth understanding the performance bottleneck for each kernel. We first select the most time-consuming kernel, which is the one with more potential for improving the whole application's performance.

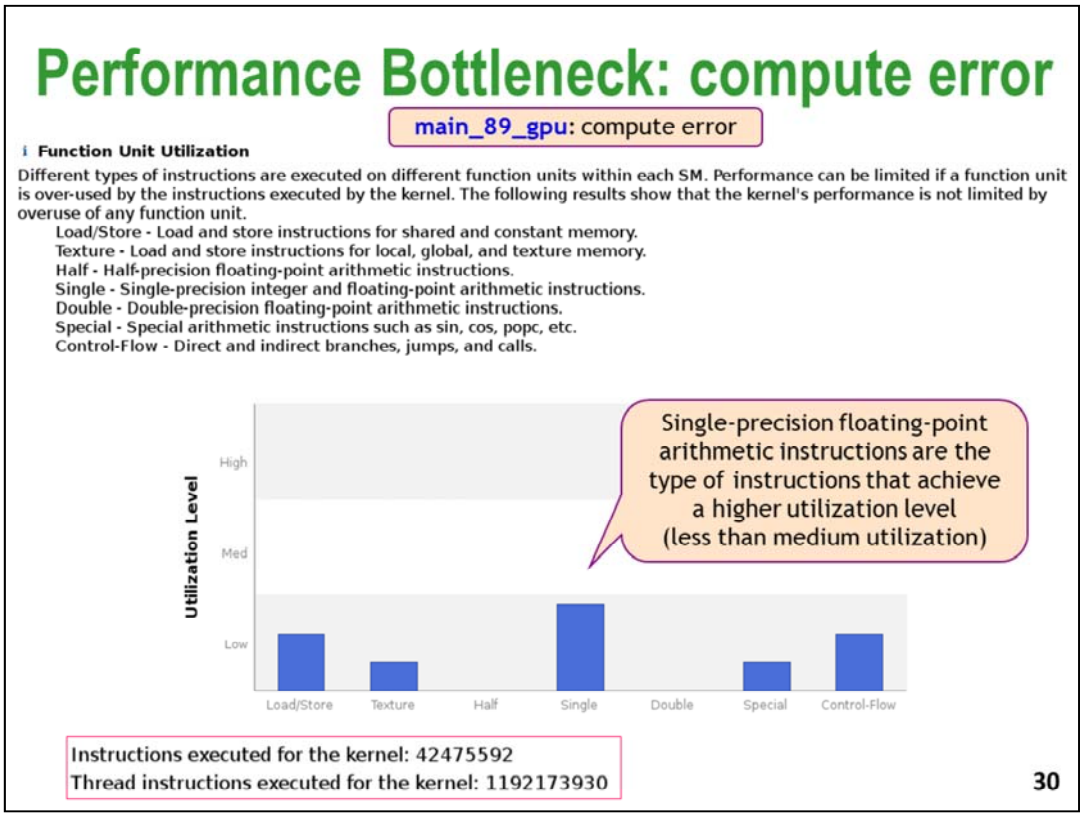


The automatic analysis made by the profiling tool is done for the kernel computing the error. The result indicates that the compute resources are used less than 45% of their capacity, and the memory resources achieve an utilization of around 65%. In this case, the memory resource with the higher utilization percentage is the device memory (notice the label below the bar), but in other cases the bottleneck could be the L2 cache, which is shared by all the multiprocessors (or SMs), or the L1 cache, which is private for each multiprocessor.

The diagnostic of the performance tool is that the performance of the kernel is most likely limited by the bandwidth of the device memory (see the main title of the window and the text below explain the diagnostic). Therefore, it seems that the computation of square roots is not a performance problem: you can check this with the optimization strategies that you have to test.

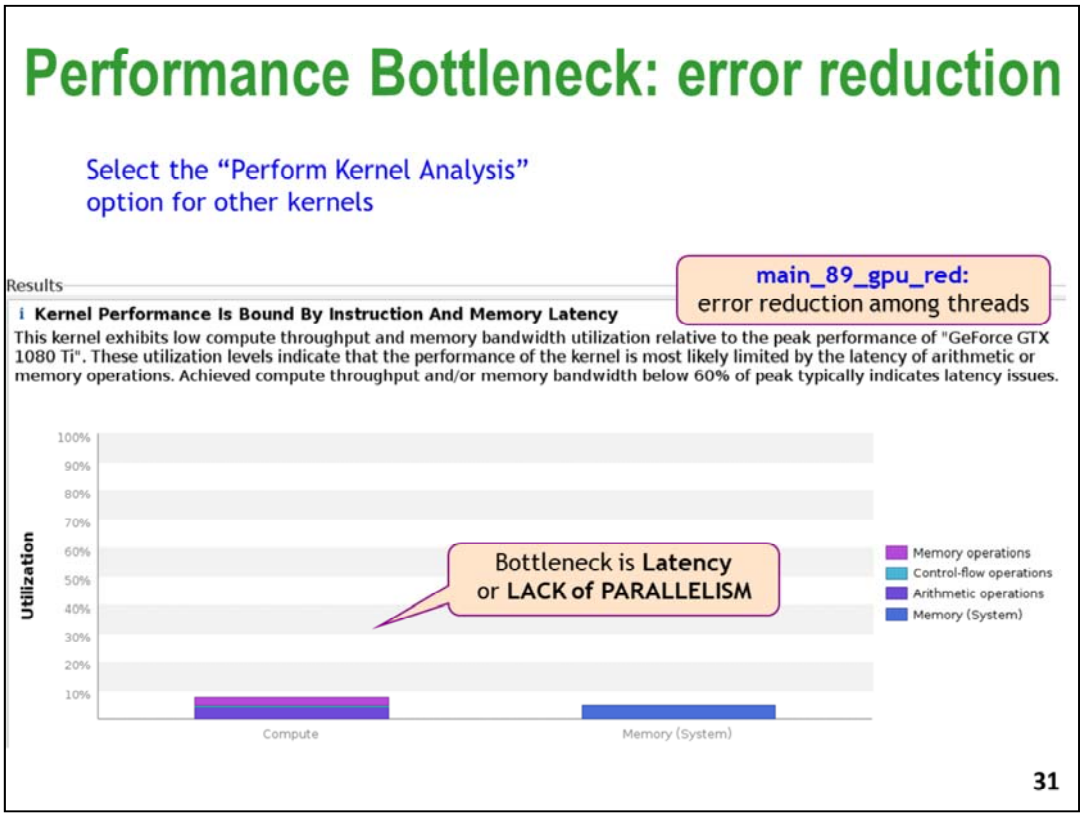
But there is still more …

We can command the tool to give you more information about the memory bandwidth usage and the result is the information shown bellow in the slide, which indicates an achieved effective bandwidth on the device memory of around 307 GB/s, which represent around 65% of the peak bandwidth (484 GB/s). The effective bandwidth achieved on the L2 cache is higher, around 388 GB/s, but only represents around 20% of the peak bandwidth.

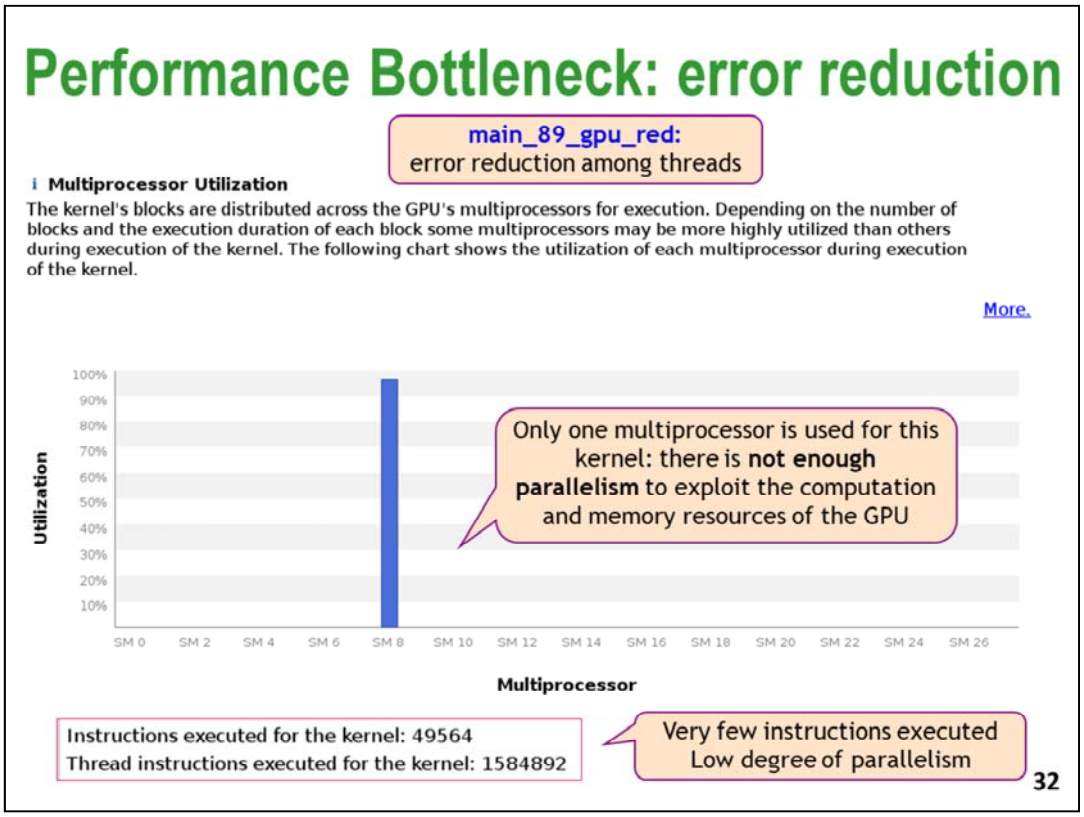


You can also get information about the usage of compute units. There are different execution capacities for different types of instructions. The previous chart indicates the utilization percentage for each operation type.

Also, we can get the number of instructions executed: instructions are SIMD with 32 lanes (the CUDA programning model provides an abstraction where each lane of the SIMD instruction is considered an independent thread). The number of thread instructions (1.19 billions) is then 32 times the number of SIMD instructions (42.5 millions). You can use this value to check if further optimizations help reducing these number or not.

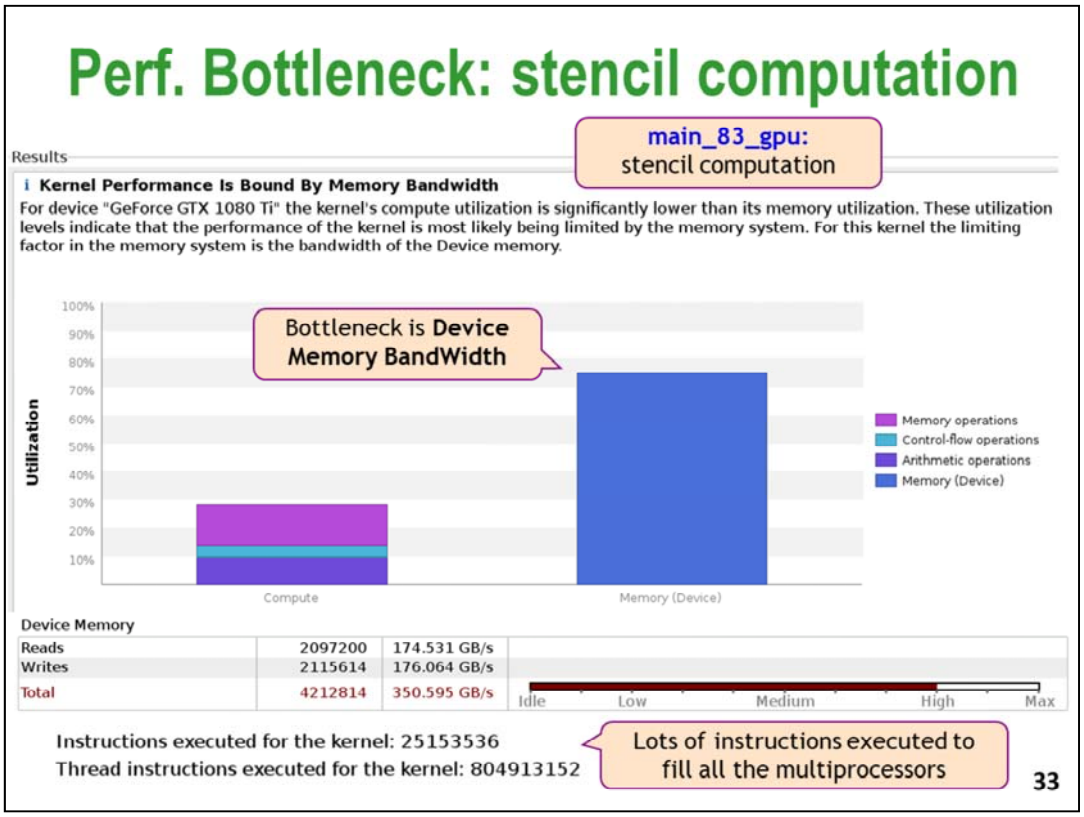


The performance of this kernel is rather different: when the utilization of all the resources is low (both compute and memory resources) then we asume that performance is bounded by latency. An alternative way to say the same is to that there is a lack of parallelism.

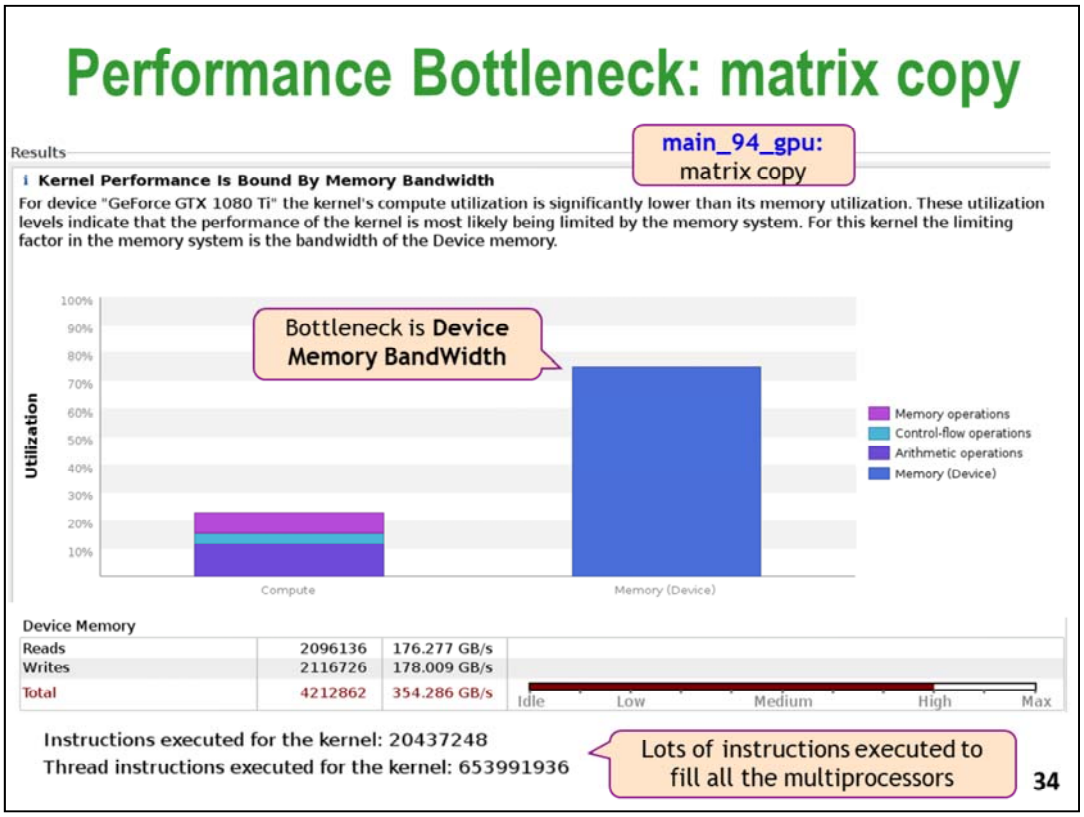


By navigating through the different analysis options we can notice that very few threads are created, and only one SM (multiprocessor) is occupied. Also, the SM is not used efficiently, because the execution comprises a very small number of instructions.

The task performed by this kernel consists of reducing the few values generated by the previous kernel by computing the máximum of all these numbers. We can find the size of the kernel and find out that only 256 threads are used. This is a very small number of threads for such a powerful GPU, and makes the resources to be highly underutilized.

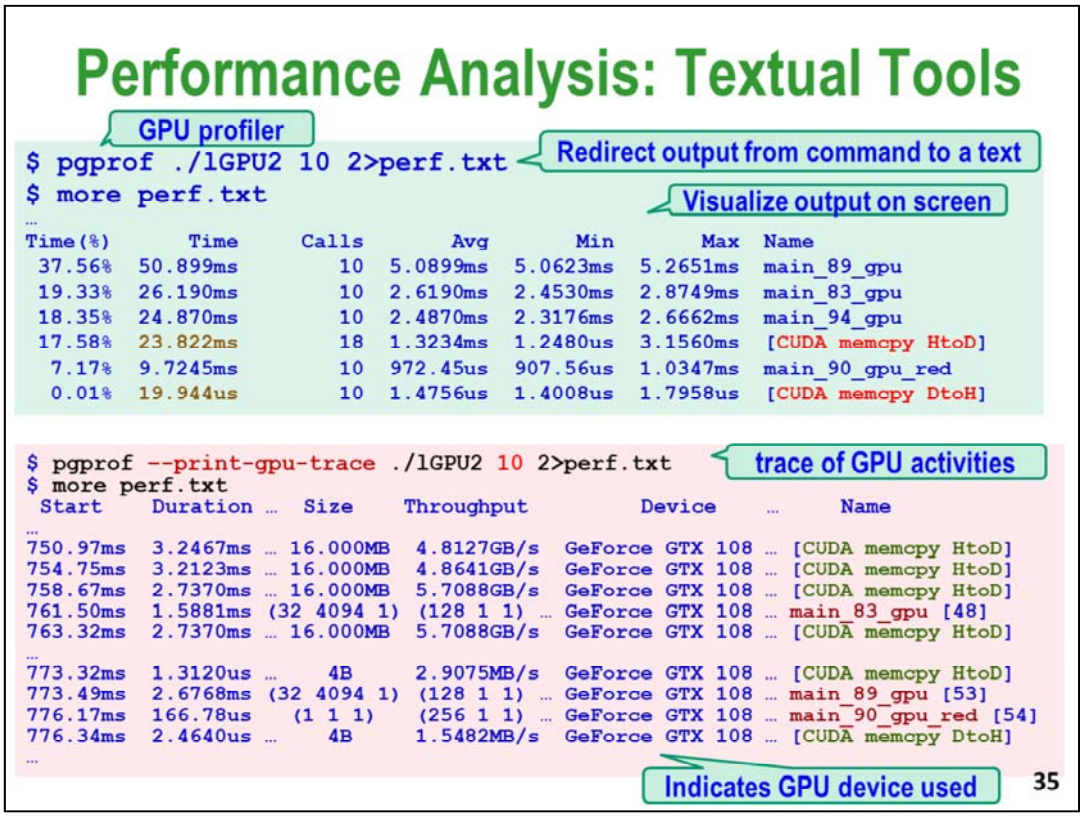


The kernel doing the stencil computation is also bounded by memory bandwidth: while the computation resources are utilized lower tan 30%, the device memory bandwidth approaches 75%. That means that the computation cost is smaller than the cost of moving the data of the matrices between the GPU computation resources and the device memory.



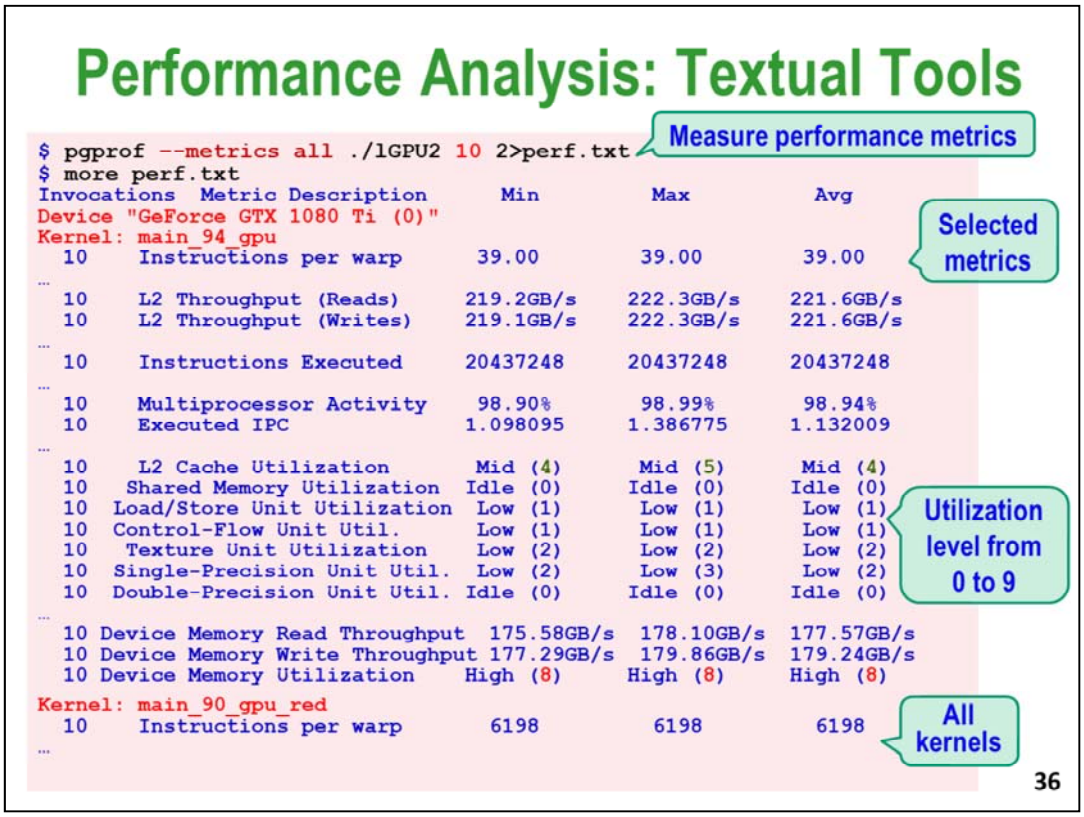
Finally, the matrix copy kernel is, as expected, bounded by memory bandwidth. The computation cost (almost zero) is smaller than the cost of moving the data of the matrices from and to the device memory.

This kernel performing copies, where each element is read and written once, takes almost the same time as the kernel performing the stencils, where each element of the matrix is read four times, some operations are done and the final result is written. In other words, the computation of the stencil and copy kernels is too low to cause any performance problem. This is important to take into account when proposing and analysing the effect of certain optimizations.



This slide shows how to execute the program and obtain several performance metrics about the execution. The first command (**`pgprof`**) provides a summary of the execution time taken by each kernel and by all the memory copies to and from the device.

The second command (**`pgprof --print-gpu-trace`**) provides a trace of all the GPU activities, indicating the duration of each activity, and some parameters, like the size of the memory block copied between host and device, or the amount of threads used by a kernel. For example, the kernel **`main_83_gpu`** uses 32×4094 blocks of threads, where each block contains 128 threads, which means a total of 32×4094×128 ≈ 16 million threads. We will discuss this issue in the next slides.



Finally, this slide shows how to get performance metrics (**`pgprof --metrics all`**) for all the kernels. A subset of selected metrics is shown on the slide. They are the most relevant on most of the applications.

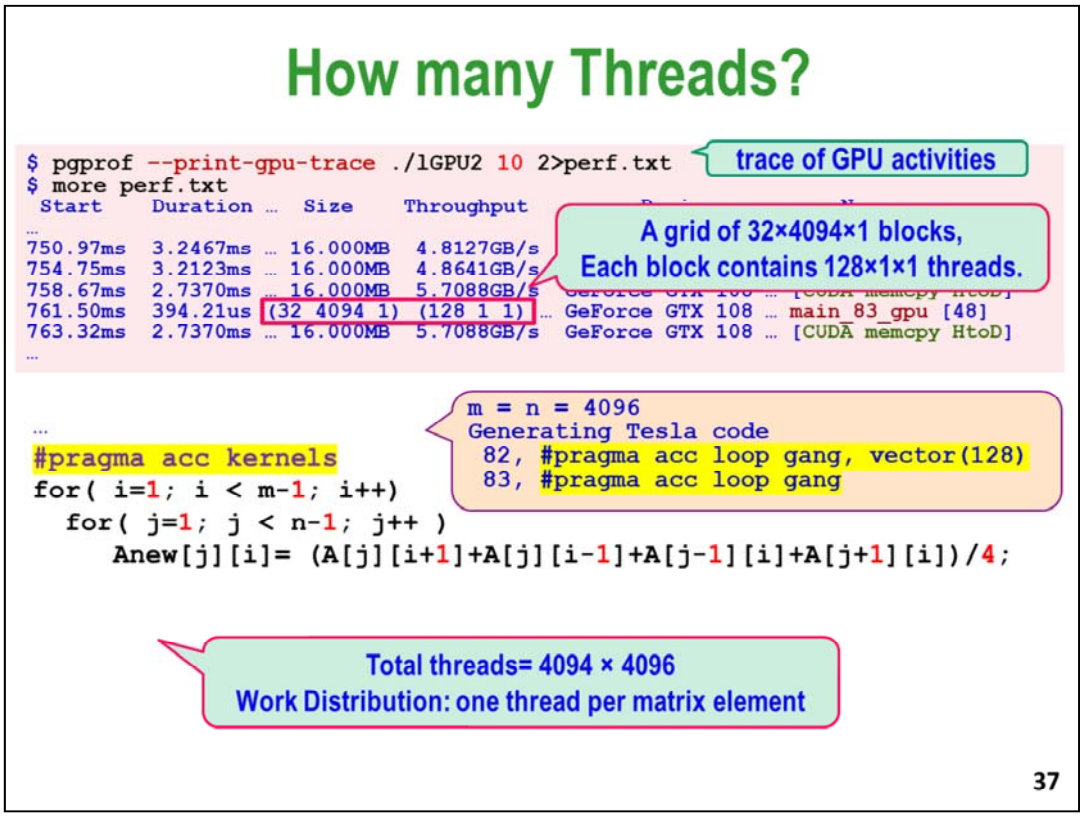
Resource utilization: a high value (7 to 9) indicates that the resource (L2 cache, device memory, Load/Store unit, Single-Precision unit …) may be a performance bottleneck

Multiprocessor Activity and Executed IPC: indicate the utilization of the multiprocessors (SMs) of the GPU (should be near 100%) and the amount of instructions executed per clock cycle (should be close to the peak IPC, but in general achieving 20 to 40% of the peak IPC is a good result).

L2 and device memory throughput: indicate the amount of data accessed from L2 cache and device memory and can be used to estimate how much data is reused in the L1 and L2 caches. It is a measure of the locality of the data accesses. This values can be used to check if too much data is read or written.

Instructions executed: indicates the total number of operations executed by the kernel (the number of operations is the number of instructions multiplied by 32). It is not easy to assess this number from a single program, but it can be useful to compare the effect of modifications / optimizations on the code.

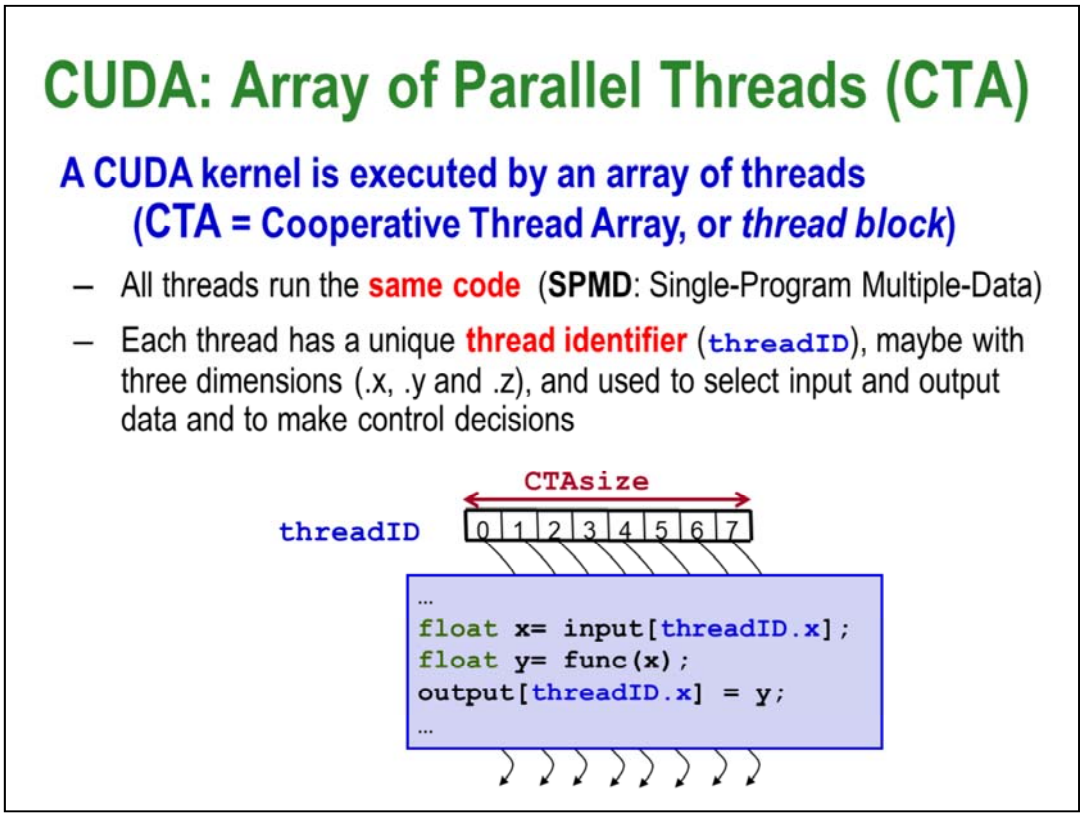
Instructions per warp: number or instructions executed by each thread. A very small value may indicate high instruction overhead (most of the instructions executed are used to initialize the thread task and only a few instructions are devoted to actually performing the task)



The command **`pgprof --print-gpu-trace`** provides a trace of all the GPU activities, and specially the amount of threads used by a kernel. In this case, the kernel **`main_83_gpu`** uses 32×4094 blocks of threads, where each block contains 128 threads, which means a total of 32×4094×128 ≈ 16 million threads.

From that, we can deduce that each thread will be responsible of computing the result for a single cell in the output matrix Anew[][]. The issue of work distribution, i.e., how the whole task in the parallel loop is distributed among the threads is very important to determine the performance of the execution.

In order to understand this question, we must describe the CUDA threading execution model and the OpenACC threading execution model.

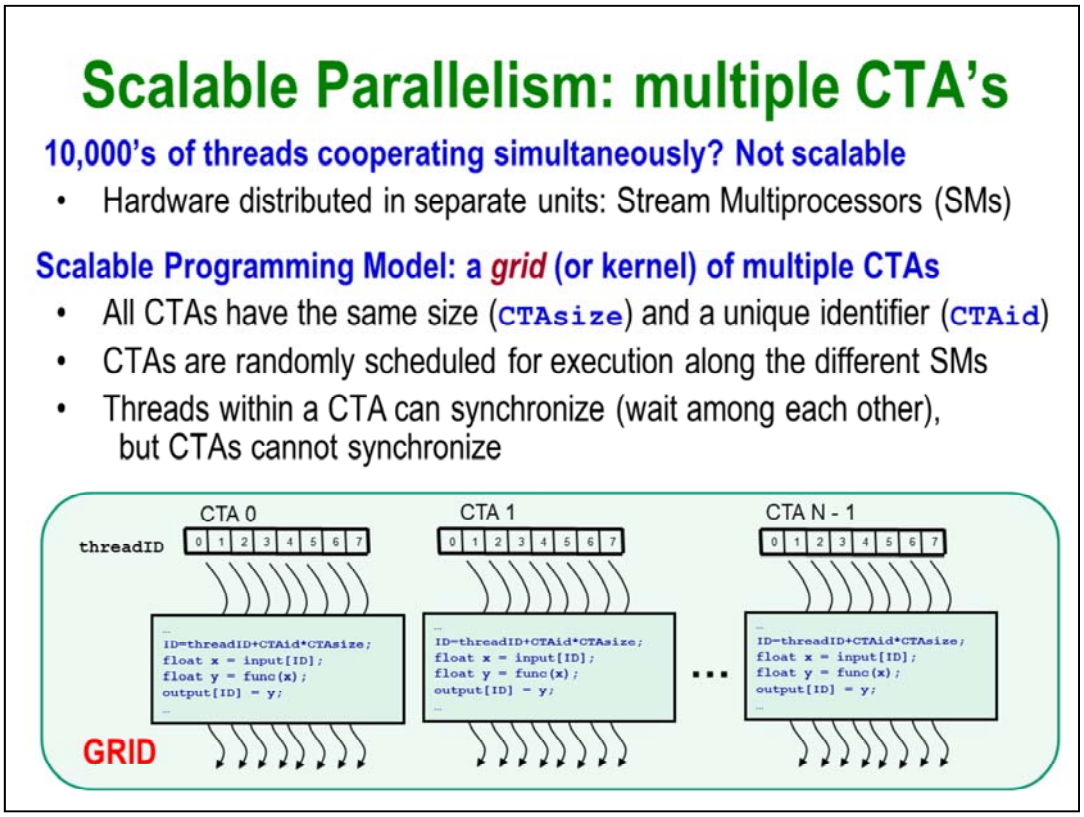


CUDA uses a parallel programming model denoted SPMD = Single-Program Multiple-Data. Each thread, among the thousands or millions created, has a unique identifier (composed of up to six numbers). A CUDA program is a sequential code (the kernel code) that is executed by all of the threads, potentially at the same time. Each thread specializes its processing task by using its own identifier to select the input data that it must process and the output data that it must generate.

The example in the slide shows a CUDA code corresponding to a set of 8 threads that compose a Cooperative Thread Array (CTA), or thread block. The thread block nomenclature is commonly used in the literature of CUDA, but NVIDA technical documents define a CTA as an alias of thread block. We prefer the term CTA, since it is not confused with other usages of the word "block" (like data block, instruction block, data blocking optimization, …). Each thread uses its identifier, threadID, as an index into an input vector and an index into an output vector. The total number of threads in the CTA, CTAsize, is a value that is available for the program at run time. Potentially, the threadID has three dimensions (id.x, id.y and id.z), but often only the first dimension (id.x) is used.

It is convenient to repeat that GPUs implement very fast mechanisms to create and manage threads on H/W, which allows the efficient execution of as few as 20 machine instructions per thread. A CPU requires the execution of hundreds of thousands of instructions per thread in order to achieve high compute efficiency.

Having a single CTA is not scalable, as we will explain next, and therefore a CUDA program consists of multiple CTAs. Each CTA has also a unique identifier, blockID, and the total number of CTAs, GridSize, is also a value that is available for the program at run time. Again, the blockID has three dimensions (.x, .y and .z), but often only the first dimension (.x) is used.

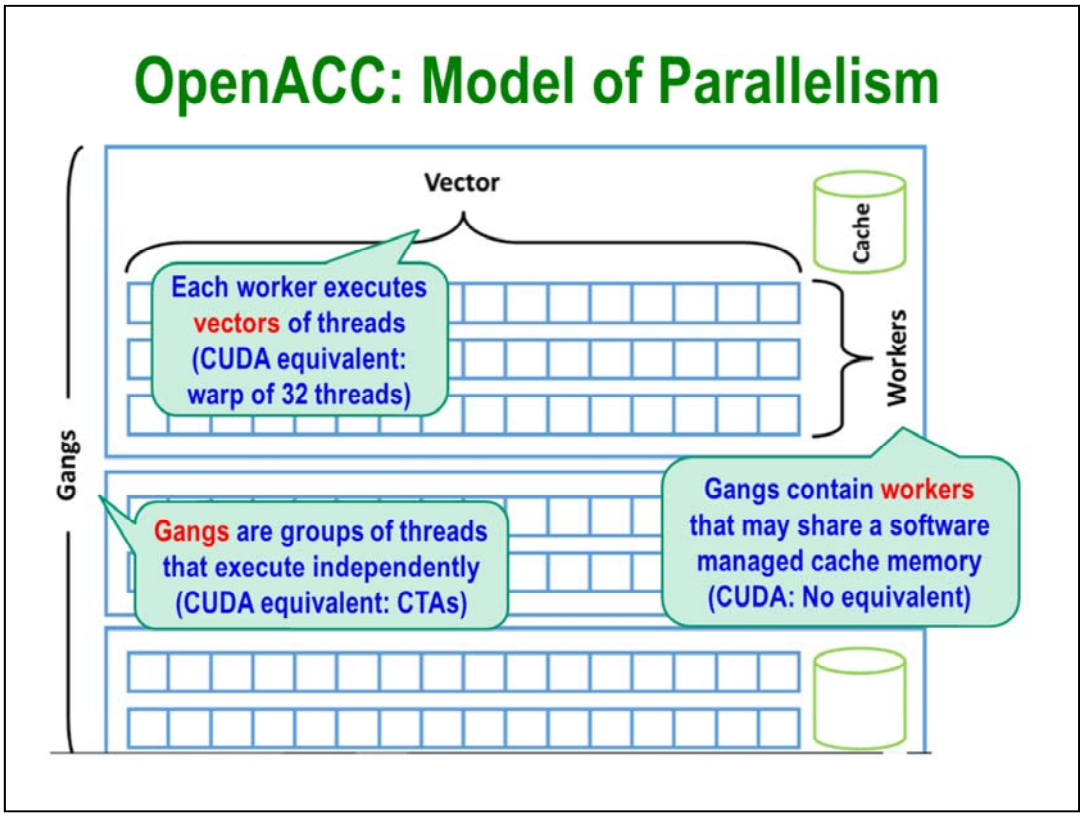


The model of using a single CTA (a single *array of cooperating threads*) does NOT SCALE performance-wise. It is not possible for many thousands of threads to cooperate among them in an efficient way (both in terms of speed and energy consumption). Cooperation requires both low-latency data exchange and low-latency synchronization. Some applications do not need any kind of cooperation among threads (this kind of parallel computation pattern is called a MAP parallel pattern). For example, adding two long vectors into a third vector can be parallelized into a large collection of threads, where each one is devoted to a single addition and does not need to cooperate with any other thread. However, some applications do require collaboration, like aggregating all the elements of an array (REDUCE parallel pattern), or sorting the elements of an array (a combination of cooperative parallel patterns).

Current GPU architectures define the maximum CTA size as 1024 threads. This could change in upcoming GPU architectures, but a substantial change is not expected. The reason behind this decision comes from the technology of building processor chips: there is no way to store a large amount of information in a memory module and at the same time being able to perform multiple and very fast simultaneous accesses to these data. A shared storage space cannot be both large and fast, and have very high access bandwidth. You cannot hold the state of the private variables (between 20 and 200 Bytes) of a large number of threads and at the same time allow any thread to exchange its internal data with another thread very fast (in a few clock cycles), or make a large number of threads to synchronize at a particular execution point.

A common-sense H/W design decision is to group a limited subset of the compute and memory resources into elements called Stream Multiprocessors (SMs). Each SM contains multiple compute units (up to 192) and is able to simultaneously execute a small number of CTAs (currently, up to 16 CTAs), accounting for a certain maximum number of threads (currently, up to 2048 threads). All the threads belonging to a CTA run simultaneously in the same SM and can share data with low-latency and high data bandwidth and can synchronize with low latency. Since different CTAs can run in different SMs or in the same SM but at non-overlapping time intervals, it is not allowed to synchronize CTAs, i.e., make one CTA to wait on another CTA.

To summarize, the CUDA parallel programming model defines a 2-level hierarchy of threads: a grid of Cooperative Thread Arrays (CTAs), each one composed of up to 1024 threads.

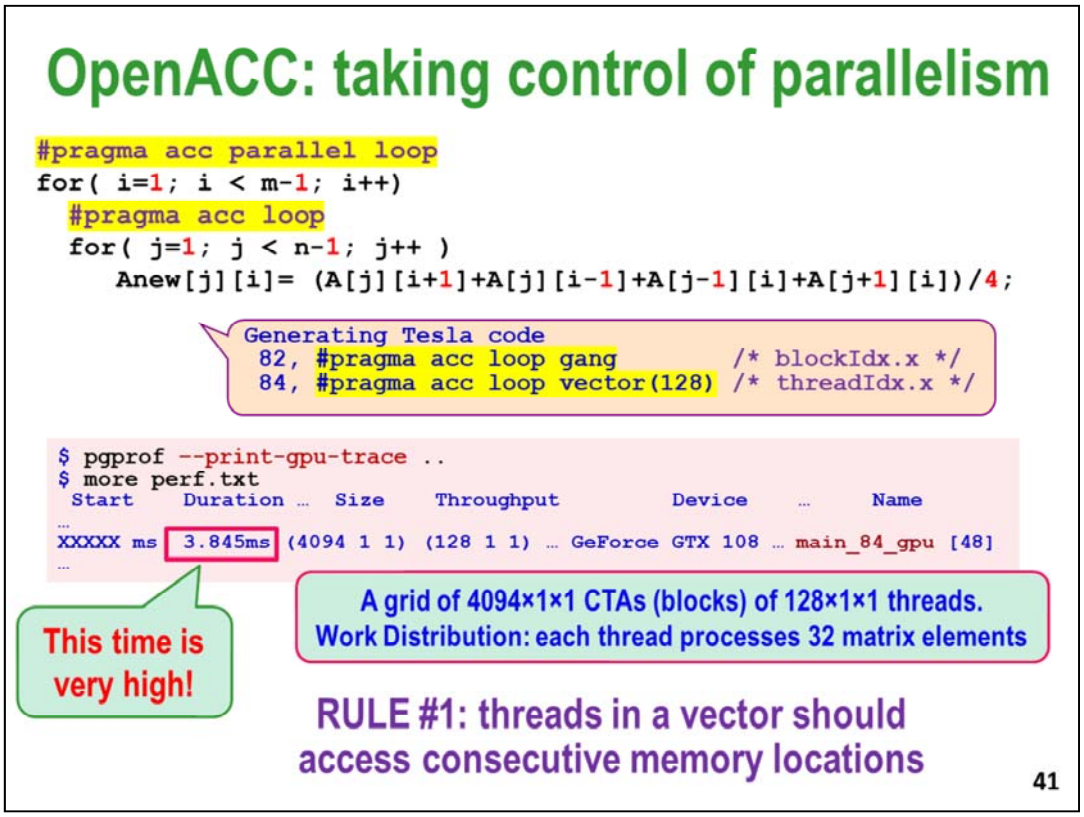


The hierarchy of threads considered in OpenACC (or the threading execution model) is as follows:

A gang of threads, where each gang is executed independently, and we cannot make any assumption about when different gangs will be executed, either at non-overlapping or overlapping moments of time. Threads in the same gang share the same software managed cache space. The CUDA equivalent of a gang is a thread block or Cooperating Thread Array (CTA). Gangs are expected to be executed in independent processing cores (in the case of GPUs the independent processing cores are the SMs)

Gangs are composed of workers. There is no equivalent in CUDA for workers. The workers in the same gang are expected to be executed in the same processing core (SM) but using the multithreading capability of the core, that combines and merges the execution of instructions of different workers along time in order to make a better usage of the computations resources. When a worker executing in a SM must wait for some result or resource, another worker can be used to issue instructions for execution, reducing the time periods where the computation resources are idle.

Each worker executes a vector of threads. The equivalent in CUDA for vectors are warps. In CUDA, warps have 32 vector lanes. Threads in a vector (warp) are assumed to be executed at the same time, synchronously. I.e., the same instruction is executed for all the threads simultaneously. In other words, the threads in a vector must be considered the lanes of a vector instruction: for instructions reading from or writing to memory this is very important, since efficient memory accesses require that the memory addresses generated by the threads in the vector (warp) are consecutive (or to very close locations).



The OpenACC **parallel** directive tells the compiler that the following code should be parallelized.

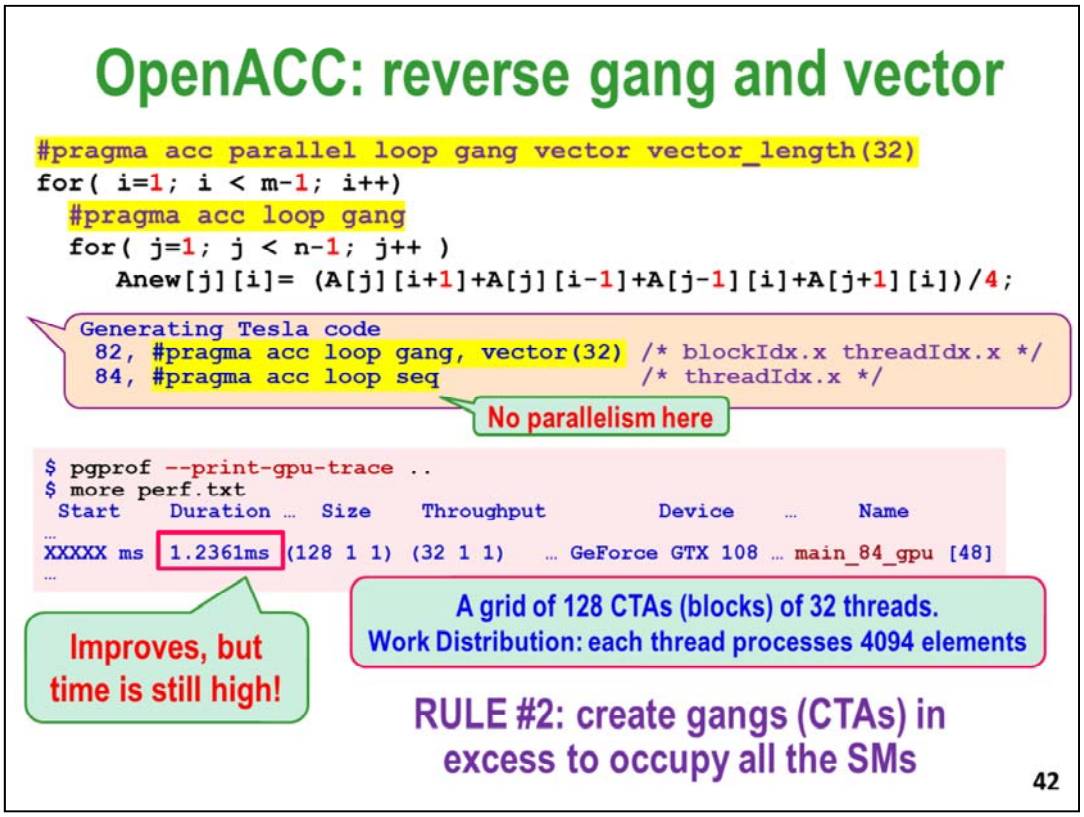
The OpenACC **loop** clause tells the compiler that the next loop is independent (all the iterations of the loop could be potentially executed simultaneously or in any order, and the result will always be the same). The compiler distributes the task in the loop among threads. If no other clause is present, the compiler decides how to distribute iterations among threads in the hierarchy of gangs-workers-vectors.

We can check the result from the output of the compiler or using the profiler tracing tool.

In this example, the compiler uses a default strategy that consists of dividing the iterations of the outermost loop among independent gangs (one iteration per gang, i.e., 4094 gangs), and dividing the iterations of the innermost loop among vectors (in this case vectors of 128 threads, so that the 4094 iterations are divided into 32 vectors of 128 threads, and each thread sequentially processes 32 elements that are at distance 128 in the loop).

The performance problem is that the inner loop indexes the columns of the matrix, and then consecutive iterations of the inner loop do not access consecutive memory positions. This is very inefficient (as in a classical CPU).

We can perform a loop interchange modification, or we can interchange the way the parallelism is distributed among gangs and vectors …



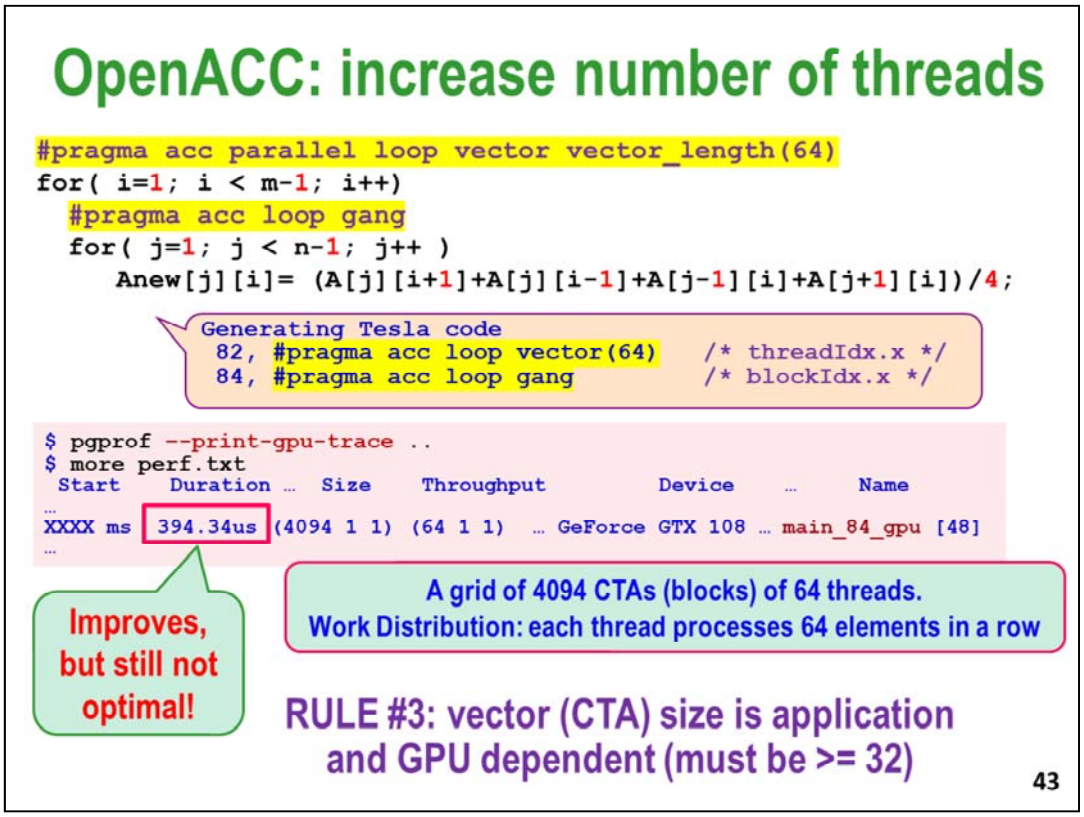
In the example shown in the slide, the way the parallelism is distributed among gangs and vectors is reversed. The outermost loop is now assigned both gang and vector parallelism, defining the vector length as 32, and the innermost loop is assigned only gang parallelism and not vector parallelism.

The result from the compiler tell us that the inner loop is not parallelized and only the iterations of the outer loop are distributed among threads. Only 128 gangs (CTAs) are created, each one processing 32 consecutive columns using vectors of 32 threads.

Execution time is improved significantly (more than 3x) because now the threads in a vector (or warp) access consecutive elements in the same row.

The problem with this work distribution is that there are too few threads to occupy the computation resources of the 1080 Ti GPU (4096 threads are not enough for a GPU that can execute 28*2048 = 57344 threads).

We will next propose an alternative to create a higher number of threads.



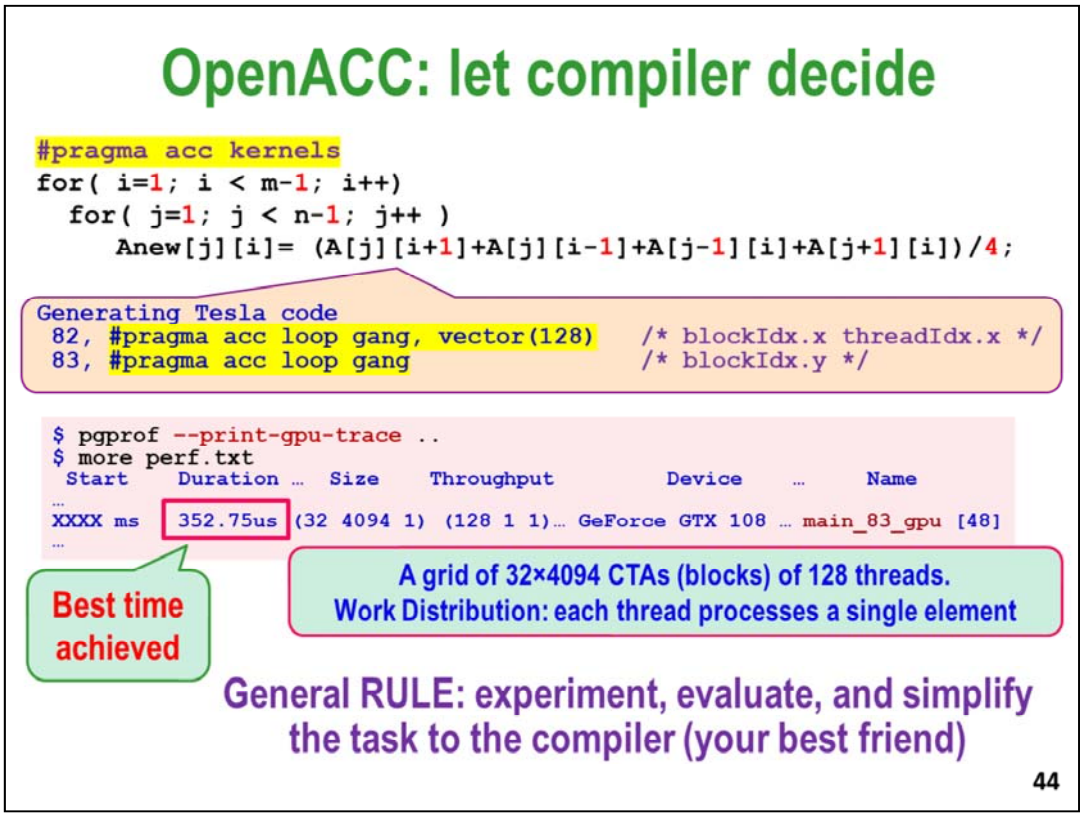
Now we only assign vector parallelism to the outermost loop

In the example shown in the slide, the outermost loop is now assigned only vector parallelism, with an arbitrarily chosen vector length of 32, and the innermost loop is assigned only gang parallelism.

The result from the compiler tell us that the inner loop is fully parallelized with gang parallelism and the iterations of the outer loop are distributed among vectors of 64 threads. Then 4094 gangs (CTAs) are created, one per row, and each gang sequentially processes the row as vectors of 64 consecutive elements.

Execution time is again improved significantly (more than 3x) because now the total number of threads is large, and also vectors (warps) access consecutive elements in the same row.

The question now is which is the optimal vector size or CTA size. This is very application dependent, and depends on the data access pattern of the loop, if data can be reused in the internal cache memory hierarchy, if there is cooperation among the threads in the vector (or CTA), …

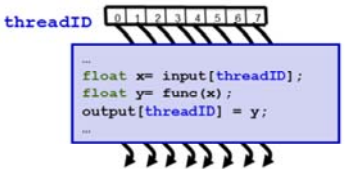


We analyze again the decision taken by the compiler when using the kernels directive and check that performance is improved around 1.17x with respect to the previous configuration. You can freely experiment with different values of the vector size, or by declaring worker parallelism. The variations in performance will not be high, since one can check using the profiler that the performance of this code fragment is bounded by the device memory bandwidth.

As a conclusion we can see that the simplest OpenACC design gives the better performance (for the GPU used and without doing an exhaustive analysis of all the possibilities). The program loop is simple, the addressing of the matrix is simple, and then the compiler can do its work very efficiently.

A good starting point is always using simple OpenACC annotations with simple code to make the compiler task easier. Then, one must assess the performance of the resulting code and identify potential problems and alternative solutions. If the potential improvement is small, there is no sense on developing a complex S/W solution. Also, when possible it is preferable to make incremental changes and assess the performance improvements on each step.

Parallel Programming



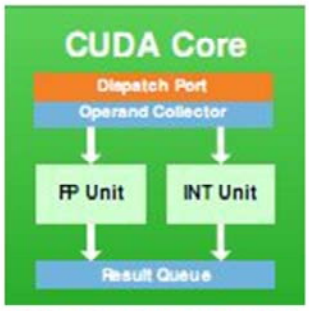
GPUs and accelerators

Throughput-oriented processors versus Latency-oriented processors

Next we will explain the main architectural issues that explain why GPUs (accelerators) can provide substantially higher performance than classical CPUs.

Single GPU unit (CUDA core)

One INTeger / Floating-Point operation per cycle



GPU unit = Execution Unit
Can execute one operation per cycle

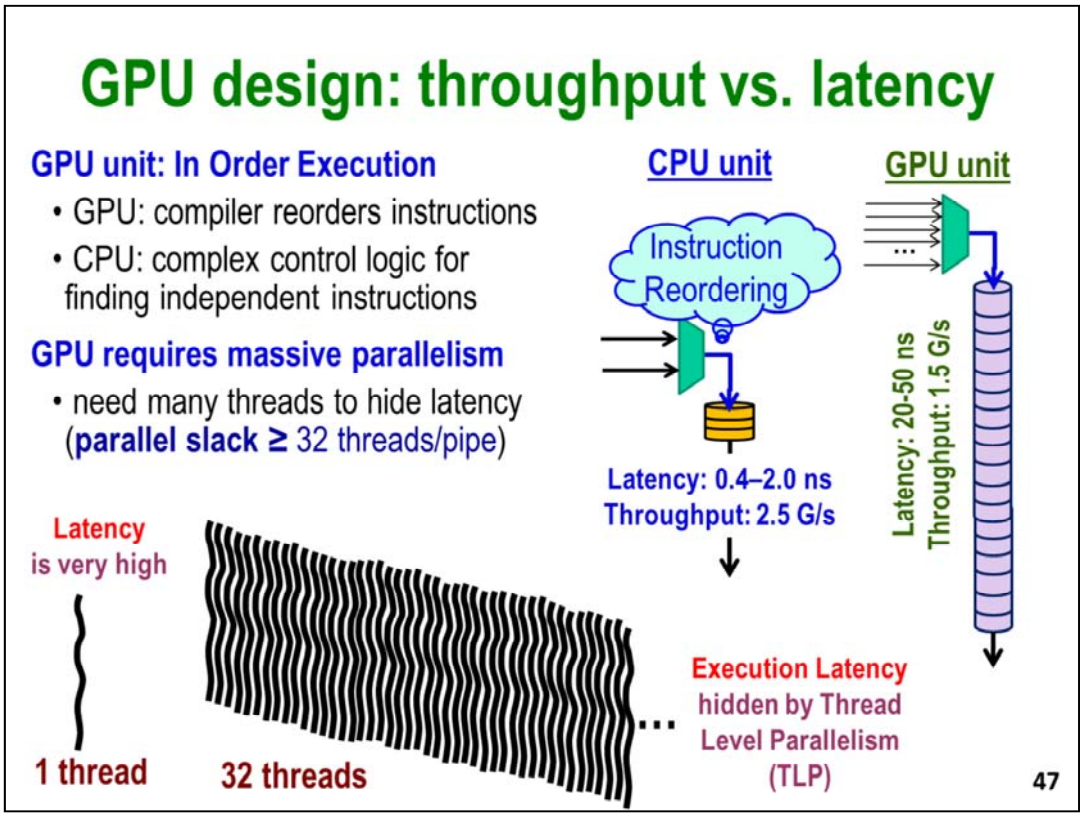
Operation Type	Latency
INT	24 clocks
FP single precision	28 clocks
FP double precision	48 clocks

Very High Latencies !!

New Paradigm: Throughput Computing

The hardware units that perform compute operations can be called GPU pipelines (or pipes): it is an abstraction that helps understanding GPU performance in a massively parallel model of execution. NVIDIA calls these units "CUDA cores": it is a term coined for marketing purposes and may create confusion, since a CUDA core (or GPU pipe) is not comparable to a CPU core. A CUDA core or GPU pipe can execute one 32-bit integer or floating-point operation every clock cycle. A CPU core can execute up to 4 compute operations per clock cycle, and some of these operations can be SIMD operations working simultaneously with vectors of 4 up to 8 values of 32 bits. As a result, a CPU core can execute between 10x and 20x the number of operations per clock cycle that can execute a GPU pipe. In addition, CPU cores often work at clock frequencies that are 2x to 4x the frequency of operation of GPU pipes. Therefore, the peak performance (throughput, measured as number of integer of simple-precision floating-point operations executed per unit of time) of a CPU core can be two orders of magnitude (100x) higher than the peak performance of a GPU pipe: clearly, both units are not equivalent.

The slide shows the operation latencies for the Fermi GPU architecture (CUDA Capability 2.0, or CC 2.0). Recent GPU architectures, like KEPLER, MAXWELL and PASCAL (CC 3.0, CC 5.0 and CC 6.0), exhibit similar execution latencies. GPU operation latencies are very high compared to CPU operation latencies. For example, integer (INT) operations often take a single cycle in a CPU core, while GPU pipes need between 20 to 30 clock cycles. Next we will explain the reasons for this strong differences.

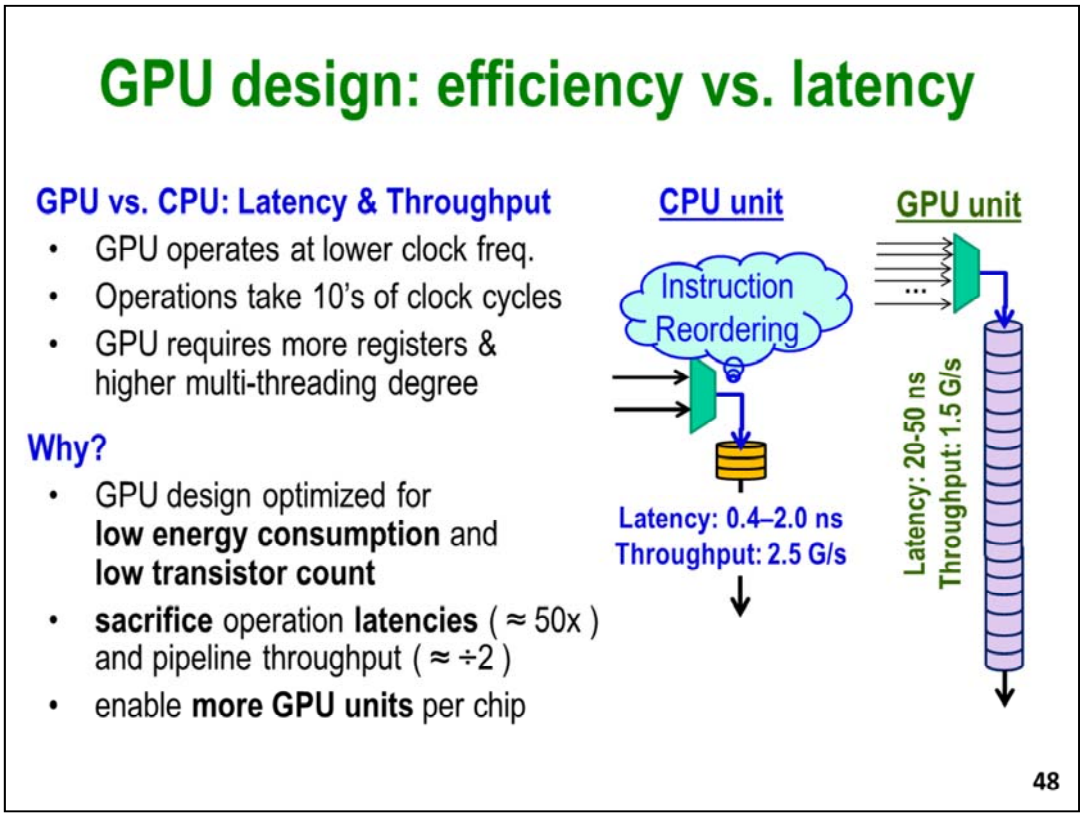


GPUs execute a single instruction in much more time (around 20x-30x) than CPUs, but they can execute much more instructions in a given time period: *throughput versus latency*. This higher performance potential is based on two issues:

1. Designing a compute unit operating with very low latency (providing the result very fast) requires much more transistors than a higher latency (slow) design. Therefore, with the same silicon area and the same (or even less) energy consumption, processor engineers can implement more GPU compute units (throughput-oriented) than CPU compute units (latency-oriented).
2. CPU designs couple 4 to 10 compute units in a single CPU core, which is then feed by a single thread or a very small number of threads (a strategy that Intel calls *hyper-threading*). The CPU core uses complex techniques to find independent instructions (Instruction-Level Parallelism or ILP) in these few threads, consuming lots of extra silicon area and providing effective utilization of the CPU cores ranging from 10% to 70%. GPU pipes couple just two compute units (integer and floating-point) but, most importantly, there is a proportion from 10 to 32 threads for every compute unit that helps achieving resource utilizations often reaching more than 90% of the overall resource capabilities.

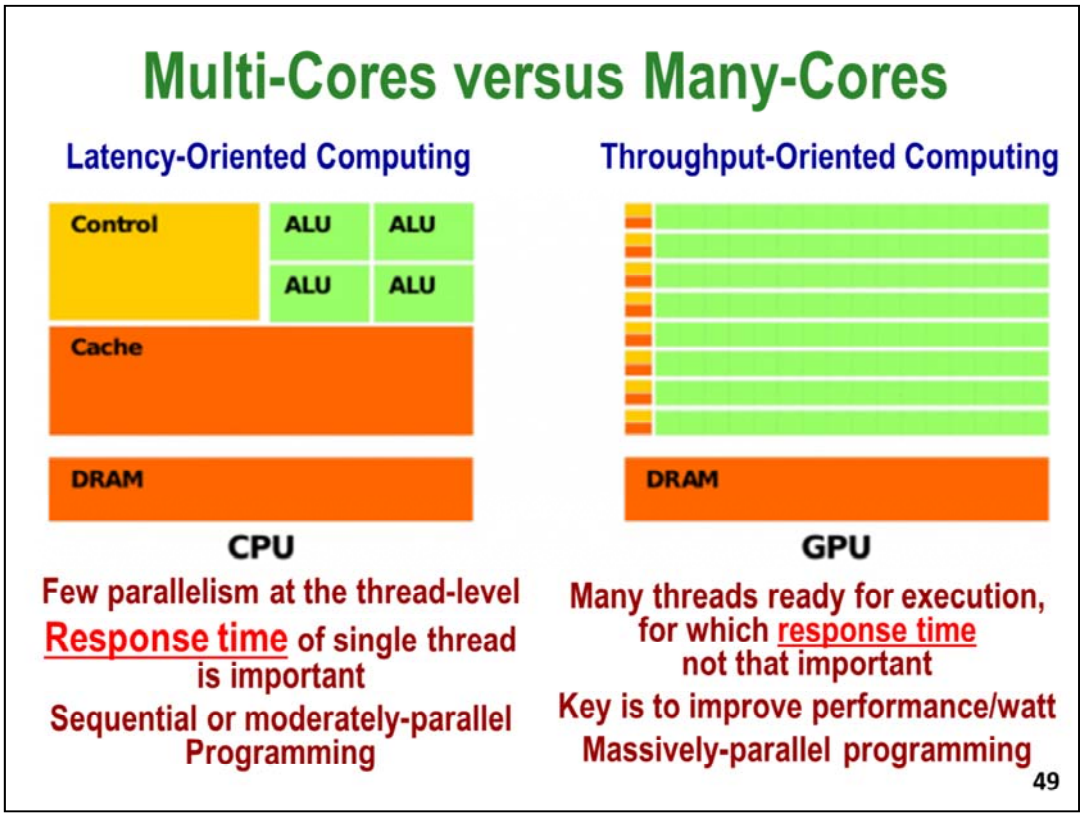
The net result is a higher number of computation resources in a GPU with respect to a CPU, and most of the time these resources are used more efficiently. The proportion of threads per GPU unit is called the parallel slack and represents the amount of Thread-Level Parallelism (or TLP) of the program.

A CPU host requires each thread to execute hundreds of thousands of instructions in order to achieve high compute efficiency. Conversely, GPU devices implement very fast H/W mechanisms to create and manage threads, which allows the efficient execution of a thread that executes as few as 20 machine instructions.



GPU designers aim for a high energetic efficiency, sacrificing the execution latencies of operations. For example, adding 32-bit integer numbers takes around 20 to 30 cycles. This is the time required from the moment when an instruction starts its execution until it generates a result that can be used by a subsequent instruction: it is the minimum time between the execution of dependent operations (the second instruction uses the result produced by the first instruction). Taking into account that clock frequency is also faster in a CPU, then the latency of a GPU is tens of times higher than on a CPU.

Basically, GPUs are able to execute more instructions per second because they contain thousands of GPU pipes –in comparison to 4 to 16 CPU cores in a CPU processor–, and they execute hundreds of thousands of independent threads –in comparison to 4 to 32 threads in a CPU processor– that can effectively hide the effects of the longer latency of the GPU execution.



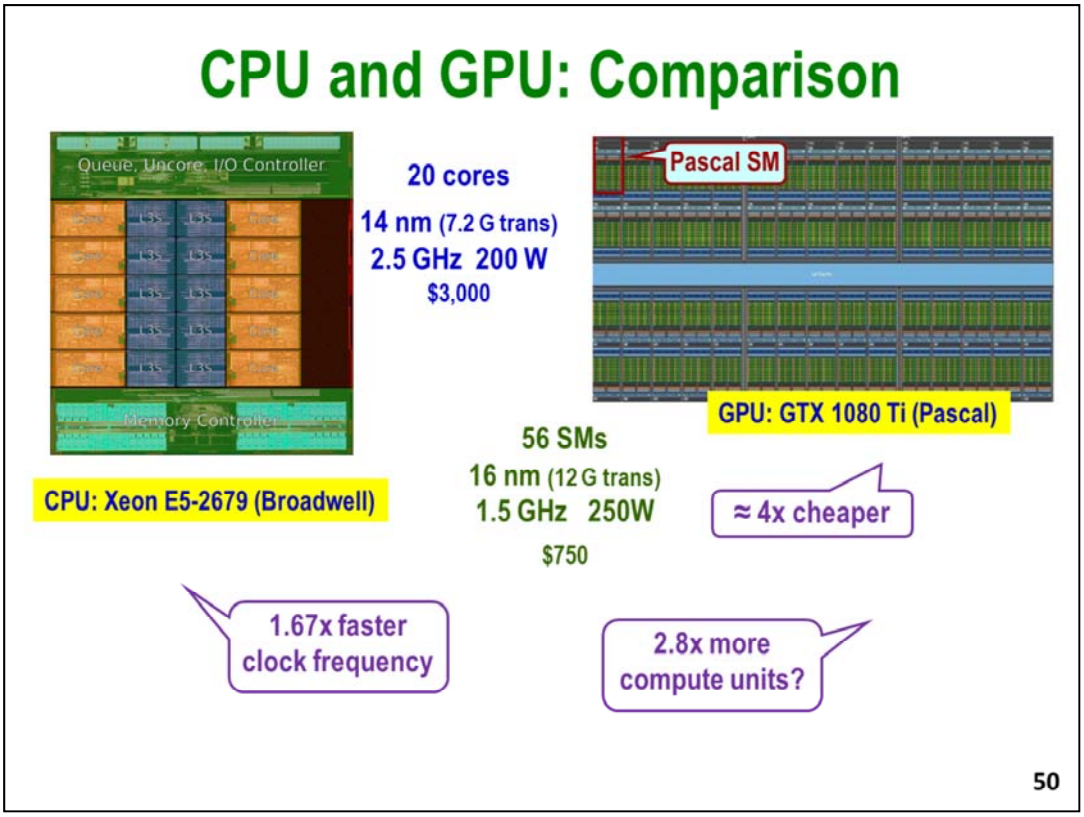
CPUs are specialized in reducing the latency time for executing a single thread, while GPUs are focused on offering high compute throughput for executing thousands to millions of threads. As a result of the different goals, GPUs are able to offer an order of magnitude better ratio among performance and energy consumption.

As shown in the slide, CPUs devote a large amount of the transistor budget to implement control logic for the dynamic reordering of the execution of the instructions in a single thread (trying to exploit the so-called Instruction-Level Parallelism or ILP). Instead, GPUs devote a very small amount of transistors to implement simpler mechanisms to schedule the execution of the instructions of multiple threads (Thread-Level Parallelism or TLP).

Additionally, CPUs devote most of the silicon in the chip for caching data from DRAM and reducing the latency of most memory accesses. Instead, GPUs mostly rely on a very high-bandwidth external GDDR memory, and contain small cache memories and small scratchpad memories that are explicitly managed by the program (the shared memory or S/W cache, and a very large pool of registers) for accelerating specific cases.

Which is the disadvantage of GPUs? Programmability.

(1) Designing massively parallel applications with an explicit and fine-grain management of the memory hierarchy is inherently complex. (2) Current massively parallel programming languages (CUDA, OpenCL, OpenACC) offer a relatively simple model for regular parallel computational patterns like MAP and REDUCE, but not for irregular parallel patterns with complex tradeoffs between memory usage and work efficiency, or parallel patterns requiring thread cooperation.

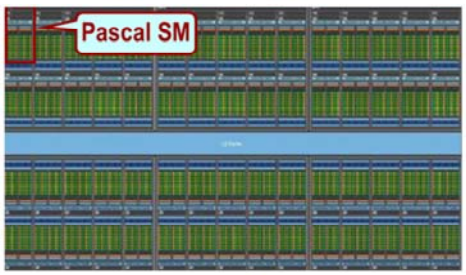


This slide presents a simplified comparison among two recent processors. The processor on the left is a typical CPU with 20 cores, targeted to server systems. The GPU on the right contains 56 multiprocessors (SMs). The GPU is approximately 4 times cheaper and contains 2.8 more compute units, but the CPU works at 1.67 more clock frequency.

These data are not enough to make a fair comparison

CPU and GPU: Latency Comparison

2.5 GHz



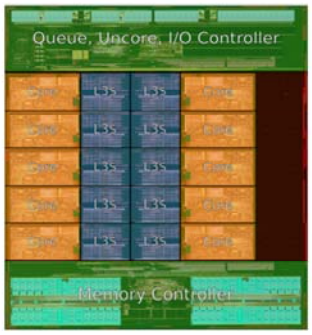
1.5 GHz

Operation Type	CPU		GPU
INTEGER 32bit	0.4 ns	50x worse	20 ns
Single Precision FLOAT	1.2 ns	20x worse	24 ns
Double Precision FLOAT	2.0 ns		40 ns
Per-Core Memory (cache) Read	1.2 ns	8x worse	20 ns
Off-chip Memory Read	50 ns		400 ns

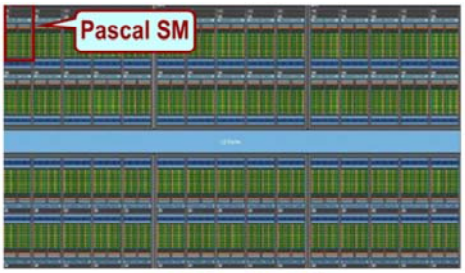
51

This slide presents the estimated latency of the 5 more frequent operations. The GPU is clearly much more slow, with differences ranging from 8x to 50x.

CPU and GPU: Throughput Comparison



20 cores
2.5 GHz



56 SMs 1.5 GHz

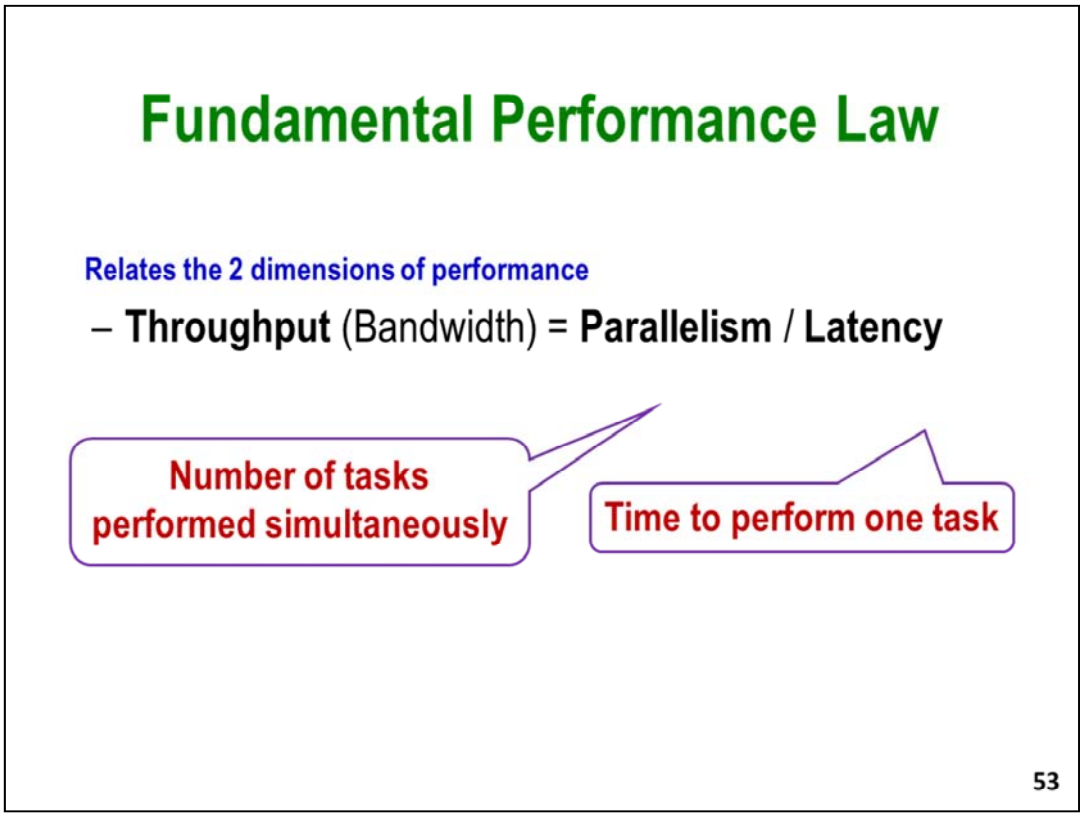
Operation Type	CPU	GPU
INTEGER 32bit	20×24×2.5 = 1.2 T/s	56×64×1.5 = 5.38 T/s
Single Precision FLOAT	20×16×2.5 = 0.8 T/s	
Double Precision FLOAT	20×8×2.5 = 0.4 T/s	56×32×1.5 = 2.69 T/s
Per-Core Memory (cache) Read	20×16×2.5 = 0.8 T/s	56×16×1.5 = 1.34 T/s
Off-chip Memory Read	71.5 GB/s	484 GB/s

6x higher

6x higher

52

This slide presents the estimated throughput of the 5 more frequent operations. The GPU is now clearly much more fast, with an advantage of around 6x



This is a classical equation to relate latency and throughput with parallelism.

Latency-Oriented vs. Throughput-Oriented Tasks

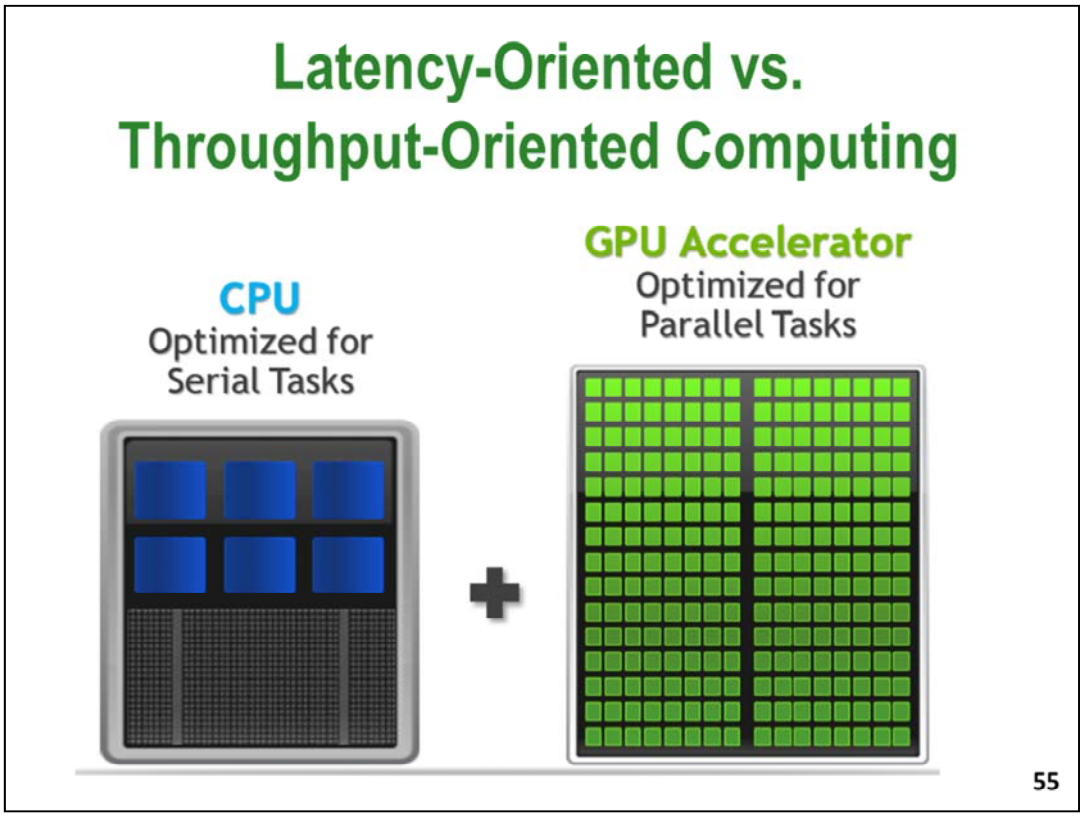
Speed

Throughput

Which is better depends on your needs...

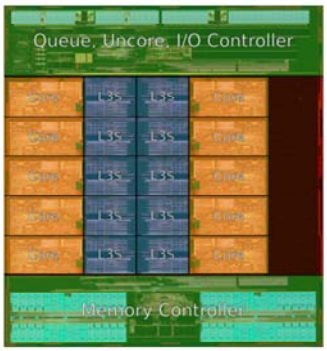
54

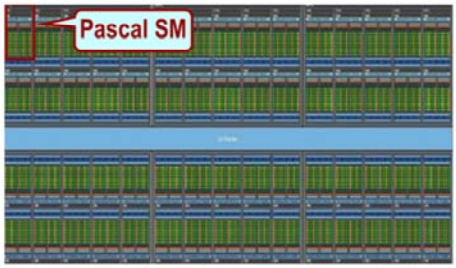
One person can be moved faster from one city to another city using a motorbike, but if your problem consists on moving hundreds of people and also you care about the cost, then a better solution requires the use of buses.



GPUs are specific for highly parallel tasks: in this scenario, GPUs often provide 10x more performance.

CPU and GPU: Operations Executed Concurrently

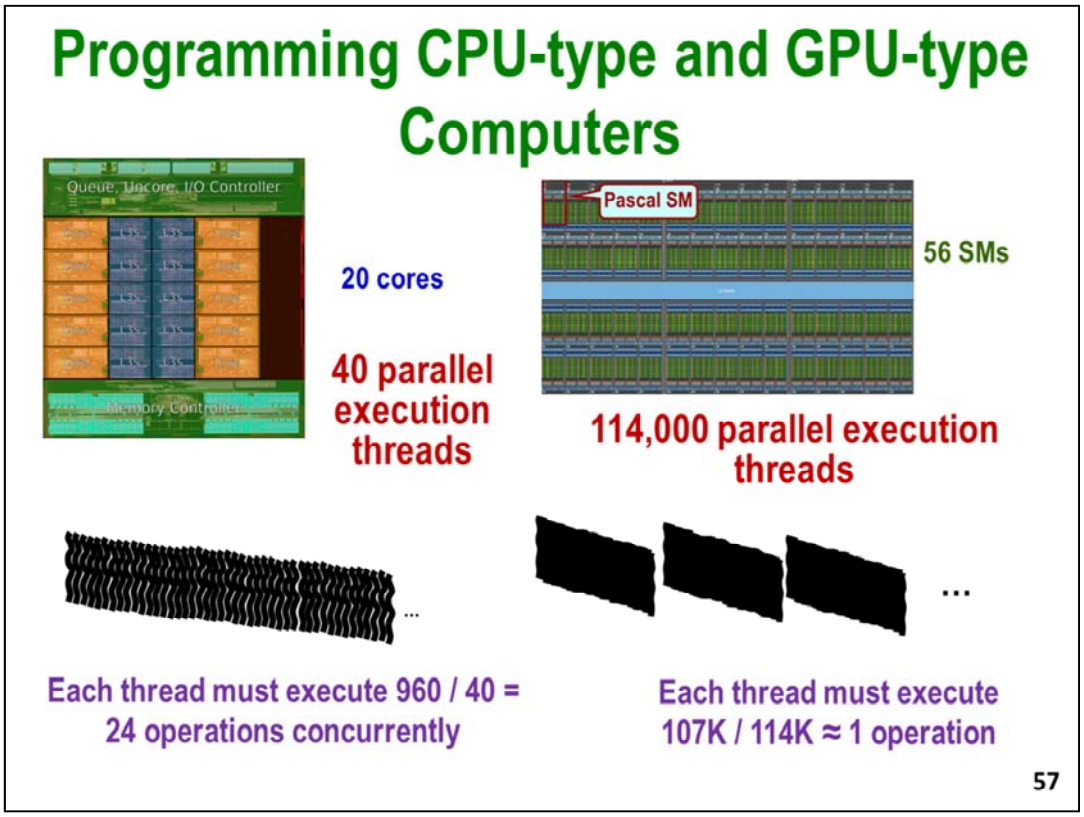




Concurrent Operations	CPU	GPU
INTEGER 32bit	1,200×0.4 = 480	5,380×20 = 107,600
Single Precision FLOAT	800×1.2 = 960	
Double Precision FLOAT	400×2.0 = 800	2,690×40 = 107,600
Per-Core Memory (cache) Read	800×1.2 = 960	1,340×20 = 26,800
Off-chip Memory Read	71.5×50 = 3,575 Bytes	484×400 = 193,600 B

56

Using the previous equation relating latency and throughput with parallelism, we calculate the amount of concurrent operations that must be performed in each system to achieve the peak throughput. The parallelism required by GPUs is hundreds of times higher. Next we will analyze how this parallelism is obtained from programs.



CPUs may seem easier to program, since you only need 20 parallel threads running on your processor to get reasonable performance. Even running a single thread seems to provide a 20th of the peak potential of your processor. The important question, though, is that each thread must be able to expose more than 20 independent operations to be able to approximate the potential performance peak of the processor: this requires a complex combination of ILP and SIMD parallelism (achieved by using the complex loop transformations that we have studied).

On the other hand, programming a GPU seems more complex, since you need hundreds of thousands of threads to reach the potential performance peak. Of course, a single thread executing on a GPU is nonsense. However, if a large loop can be parallelized to provide enough threads, then each thread is responsible to provide a single independent operation, which is trivially achieved. Therefore, there is often no need to make an effort to achieve ILP and SIMD from single threads, which may simplify programming.