

How many threads can run on a GPU?

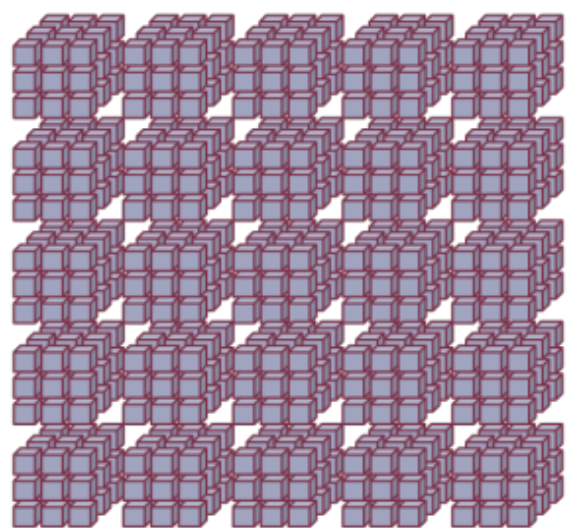
POSTED BY VINCENT HINDRIKSEN ON 24 JANUARY 2017 WITH 4 COMMENTS

Q: Say a GPU has 1000 cores, how many threads can efficiently run on a GPU?

A: at a minimum around 4 billion can be scheduled, 10's of thousands can run simultaneously.

If you are used to work with CPUs, you might have expected 1000. Or 2000 with hyper-threading. Handling so many more threads than the number of available cores might sound inefficient. There are a few reasons why a GPU has been designed to handle so many threads. Read further...

NOTE: The below description is a (very) *simplified model* with the purpose to explain the basics. It is far from complete, as it would take a full book-chapter to explain it all.



Blocks of Threads

First: what does “running” mean?

On a CPU there can be more software-threads than hardware-threads running, using continuous context-switching – only when one compute-intensive program needs to take over the whole computer, manual optimisation is needed to perfectly fit the work to the processor. This is often done by scheduling N to $2*N$ threads on N cores, depending on the effect of hyper-threading. So on the CPU *all* threads are in a running state, if not actively put into sleeping state.

On a GPU this is slightly different. If an OpenCL programming is running, only a subset of the program's enqueued threads are actually running. The non-running threads just wait their turn and don't interfere with the running threads. We can explain this by starting from with something familiar: hyper-threading.

Tomorrow on our blog, the answer to the GPGPU beginner's question: How many threads can run on a GPU?



The consensus on Twitter is that “running” is equal to the number of active threads, not the number of enqueued threads.

Hyper-threading on the CPU vs the GPU

Context-switching is used to hide memory-latency on both the CPU and the GPU. On the CPU there are 2 threads per core and on the GPU there are 4 to 10. Why doesn't have a CPU have more threads per core? Because the type of tasks are very different.

Threads for task-parallelism or data-parallelism

Parallelism is doing operations concurrently, and can be roughly split into data-parallelism and task-parallelism. Data-parallelism is applying the same operation to multiple data-items (SIMD), and task-parallelism is doing different operations (MIMD/MISD). A GPU is designed for data-parallelism, while a CPU is designed for task-parallelism. While both processors get better in the focus-area of the other, it isn't enough to remove the differentiation.

The following is a simplified model. On the GPU a kernel is executed over and over again using different parameters. A thread is no more than a function-pointer, with some unique constants – a scheduler handles multiple threads at once. This is in contrast with a CPU, where each core has its own scheduler.

CPU design goal: increase processor-usage

Intel CPUs have two threads per physical core for one main reason: optimising usage of the full core. How does this work? A CPU-core consists of several modules, which can run in independently – if two threads use different modules, the speed can be (almost) doubled. If the amount of threads is larger than the hardware can handle (2 times the number of cores), the hardware is shared by all threads by the OS – this can slow down all the processes.

GPU design goal: hide memory latency



Memory latency is the time that it takes to load data from main memory. A CPU solves the problem of memory-latency by having much bigger caches and very large schedulers. A GPU has so many more cores, that this approach does not work.

The execution model of GPUs is different: more than two simultaneous threads can be active and for very different reasons. While a CPU tries to maximise the use of the processor by using two threads per core, a GPU tries to hide memory latency by using more threads per core. The number of active threads per core on AMD hardware is 4 to up to 10, depending on the kernel code (key word: occupancy). This means that with our example of 1000 cores, there are up to 10000 active threads. Due to the architecture of the GPU (SIMD), the threads are not per work-item (core) but per work-group (compute unit).

On the GPU it takes more than 600 times longer to read from main memory than to sum two numbers. This means that if two data-items are being summed, the GPU-cores would be doing mostly nothing when there was only one active thread.

A beginner's mistake: manual scheduling

A typical beginner's mistake is to handle scheduling of the GPU-threads as if they were CPU-threads. It is logical with a CPU-mindset to put the number of threads equal to the number of GPU-cores times i.e. four.

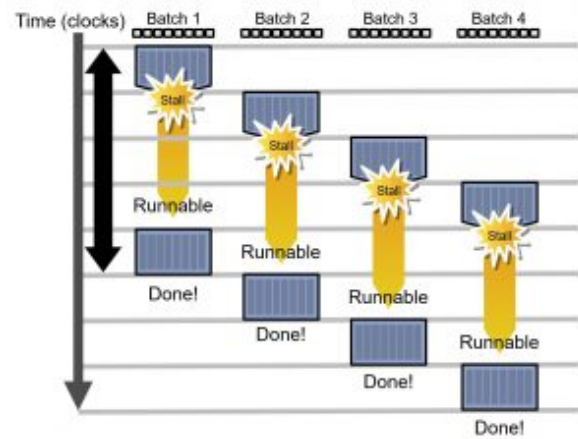
This does not give a speedup for two reasons. First there is no opportunity for the scheduler to schedule new threads when possible. Second there is overhead to start a new kernel on the GPU.

When the number of threads is relatively low, having the number of threads equal to an exact multiple of the number of cores does have a slight advantage (a few percent). As handling "the rest" takes more time, it is not an optimisation technique that's often practical.

So, how many threads can a GPU handle?

The maximum number of active/running threads is equal to the number of cores on a GPU times 10. As we've learnt, we should not try to manually optimise for that. But how about the number of enqueued threads?

We're talking scheduled threads here. The maximum amount of (enqueued) GPU-threads is *at least* close to what is representable in 32 bits integers (4 billion) for each dimension. Given the three dimensions, this is more than enough. Let's put it this way: we never got an `CL_INVALID_GLOBAL_WORK_SIZE` error for a too large dimension.



Latency hiding with 4 threads. Image ©AMD

`CL_INVALID_GLOBAL_WORK_SIZE` if `global_work_size` is NULL, or if any of the values specified in `global_work_size [0], ... global_work_size`



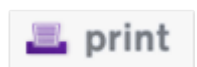
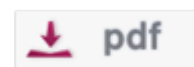
[work_dim - 1] are 0 or exceed the range given by the sizeof(size_t) for the device on which the kernel execution will be enqueued.

This means that when you want to do pixel-wise operations on a 100 Megapixel image (i.e. 10.000 x 10.000 pixels) you loaded to the GPU, then you can simply launch such kernel without thinking about those 100 million threads.

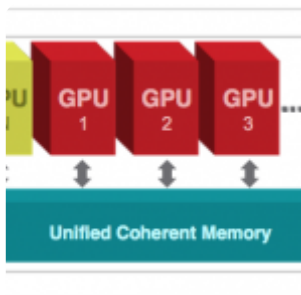
Questions?

If you want to know more about latency hiding on the GPU, you can find more information on this to look for OpenCL 2.0 sub-groups, NVidia's warp, AMD Wavefronts and Intel/AMD CPU's hyper-threading.

Feel free to ask questions in the comments. You can also attend one of our trainings.



Related Posts



Heterogeneous Systems Architecture – memory sharing and task dispatching

... Queuing). These two techniques had undergone so many updates, that new terminologies were used. In this blog post, ...



MPI in terms of OpenCL

... the ideas behind them are different. Explained below. Threads - Kernels: and the kernels work in threads too, so nobody beats you ...



The OpenCL power: offloading to the CPU (AVX+SSE)

... it is an operation using an M-wide vector or is it using M threads. The answer is both - vector-computations are a subset of ...



Aparapi: OpenCL in Java

... Aparapi has been open sourced and many issues have already been fixed and improved. If you have an AMD ... and ...

General



WRITTEN BY **VINCENT HINDRIKSEN**

Mr. Vincent Hindriksen M.Sc is the founder of StreamComputing, a collective of performance engineering experts.

Do you need expertise in performance engineering? We have several experts available (HPC, GPGPU, OpenCL, HSA, CUDA, MPI, OpenMP) and solve any kind of performance problem. Contact me via phone or mail to discuss further: +31 854865760 or vincent@streamcomputing.eu

4 Comments

StreamComputing

 Login ▾

 Recommend

 Share

Sort by Best ▾



Join the discussion...



Nikolay Polyarniy • 3 months ago





What about time limit for kernel execution (~5 seconds)?

Does it setup upper bound for global worksize scheduled for execution?

^ | v • Reply • Share ›



streamcomputing Mod → Nikolay Polyarniy • 3 months ago

There are two main ways to prevent this. One is to remove the Windows GPU watchdog. Second is to indeed feed the GPU batches of about one second.

^ | v • Reply • Share ›



Nikolay Polyarniy → streamcomputing • 3 months ago

I am talking about common case when application used on home/office PC by final customer, so removing watchdog is not an option.

The feeding of batches about of one second leads to new problem: How to balance between full GPU utilization (there are many GPUs with 3K+ threads, so running small tasks leads to under-utilization) and absence of timeout on low-end mobile GPUs (timeout often leads to driver and application crash).

My best idea (that is portable enough) is to believe that $256 \times \text{ComputeUnits}$ leads to full GPU utilization without risk of crash due to timeout. (GTX Titan X has 3072 cores= $24\text{CU} \times 128$, GT 525M has 96 cores= $2\text{CU} \times 48$)

Am I missing something? Is there a better portable way to balance between timeout and under-utilization?

^ | v • Reply • Share ›



Timur Magomedov → Nikolay Polyarniy • 3 months ago

Number of compute units can be used as a rough performance guess together with device clock frequency but kernel time is a best performance metric for particular kernel on particular device.

I personally divide global size in parts and process them sequentially to balance between timeout and under-utilization. Say you should have single kernel duration be from 5 ms for a good GPU utilization up to 20 ms to avoid screen freezes if system have same GPU for graphics and calculations. With $10\text{K} \times 10\text{K}$ initial global size start with kernels of $10\text{K} \times 8$ size and increase it by two if kernel time is smaller than 5 ms. Divide it by two if time is bigger than 20 ms. Kernel can be written with `global_work_offset` in mind for this to work.

Such approach works usually if task calculated by a single work item is not huge enough to take 5+ ms.

^ | v • Reply • Share ›

ALSO ON STREAMCOMPUTING

We're a member of Khronos now!


2 comments • 2 years ago •

Meta Says: Wow, well done


AMD gets into Machine Intelligence with "MI" range of hardware and software

1 comment • 5 months ago •




 **Mate Soos** — wow, well done, congratulations! Well deserved, too :) Good job!

1 comment • 3 months ago •

 **tivadj** — I wonder how HIP impacts OpenCL support/evolution. We may end up with HIP be more suited for Desktop hardware and


Should SPIRV be supported in CUDA?

5 comments • 2 months ago •

 **Animanteum** — I assume what you're saying is that the other languages can be compiled to SPIRV, right? I supposed I was just a bit

Call for papers: SYCL workshop, 13-March-2016, Barcelona, Spain

2 comments • 2 years ago •

 **streamcomputing** — The hint to Nvidia (and all other vendors) is: support SPIRV 1.1 in

Latest posts

 [AMD ROCm 1.5 Linux driver-stack is out](#)

 [DHPCC++ Program known](#)

 [IWOCL – all the talks](#)

 [StreamComputing is 7 years!](#)

 [Looking for the company's GPU-pioneers](#)

Upcoming Events

No events

 Follow us    

ABOUT THE COMPANY

Our clients are R&D focused SMEs and corporations, research institutes and startups.

We solve HPC, low-latency, low-power, and programmability problems.

ADDRESS

StreamComputing BV
Naritaweg 12B
1043 BZ Amsterdam
Netherlands, Europe



phone: +31 854865760

e-mail: info@streamcomputing.eu

NEWSLETTER

Sign up for our Newsletter

Email Address*

First Name*

Last Name*

Most interested in:*

- ☐ Faster software in general
- ☐ Faster Artificial Intelligence
- ☐ Faster Big Data
- ☐ Faster Image Processing
- ☐ GPU programming with OpenCL and CUDA
- ☐ OpenCL on networked FPGAs
- ☐ Stay up-to-date with what we do

Background:*

- ☐ Management
- ☐ Professional GPU/FPGA developer
- ☐ Professional CPU developer
- ☐ Academic researcher
- ☐ Other

Want to say anything to us? (not required)

* = required field

