GPU computing latency analysis

Ing. Miloš Očkay, PhD[1]

[1] Department of Informatics
Armed Forces Academy of gen. M. R. Štefánik
Liptovský Mikuláš, Slovakia
milos.ockay@aos.sk

Abstract: Graphics units are no longer just the special purpose computing devices, but they successfully have conquered the world of parallel acceleration for the general purpose problems. Application interface has become simple enough to be accepted and adopted by many programmers. Graphics Processing Unit (GPU) can provide massive parallel power in small affordable package with reasonable power consumption characteristics. New parallel co-processor included even in a desktop size computer or laptop can provide clustered parallel power and speed up many originally Central Processing Unit (CPU) based problems. However, CPU problem has to be adopted carefully with many things in mind. Successful GPU acceleration relies on problem's data structure, the amount of communication and many other aspects. This goal usually requires great deal of optimization. This paper presents a transfer latency analysis in relation with the complexity of the problem. It also clearly explains how even embarrassingly parallel problem can fail the GPU acceleration. This paper also describes modern GPU solutions to the latency communication bottleneck.

Keywords: CUDA, GPU, Compute unified device architecture, Graphics processing units, Latency,

1 Introduction

During the last few years GPU has gained tremendous massively parallel computational power and also very high internal memory bandwidth. However, GPU is not the standalone system and the bottleneck of this parallel architecture resides in memory transfers between host and graphics adapter (device) memory. There are new concepts which try to overcome this problem by hiding the latency of transfer behind the computation and the concurrency, but all of these are more likely problem specific, not the general purpose solutions. Problem which can be successfully accelerated with GPU has to exhibit specific data dependencies and relevant level of computational intensity. Many conditions and dependencies hidden inside the algorithm and data structures can affect the adaptation of the problem in the positive or negative way.

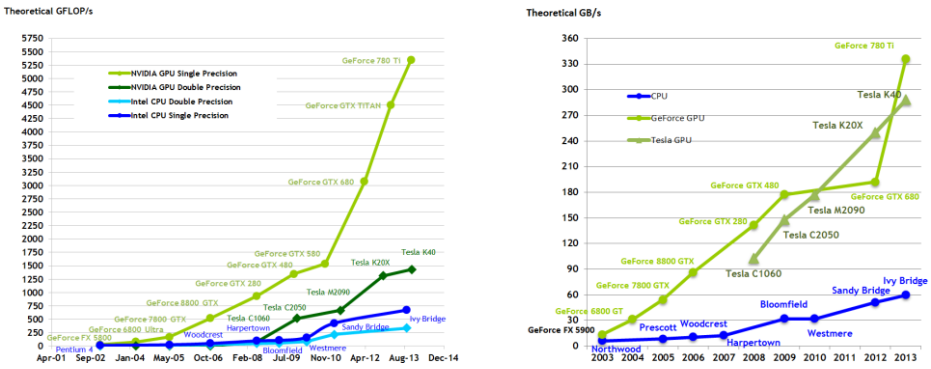

Fig. 1. Computational power and bandwidth of GPU devices [3]

1.1 GPU Transfers

All Compute Unified Device Architecture (CUDA) programs share the same structural pattern. Memory allocation and memory transfers are essential for the GPU computation. Basic procedure usually includes host and device memory allocation and transfers between these two memory spaces. Host to device memory transfer delivers data for GPU processing. Device to host memory transfer provides host with partial or final results computed by the GPU kernel. There are basically three variations of CUDA transfer directives.

```
// Host memory allocation
   float* host_A = (float*)malloc(size);
   float* host_B = (float*)malloc(size);

// Device memory allocation
   float* device_A;
   cudaMalloc(&device_A, size);
   float* device_B;
   cudaMalloc(&device_B, size);

// Copy vectors from host memory to device memory
   cudaMemcpy(device_A, host_A, size, cudaMemcpyHostToDevice);

// GPU Computation...

// Copy vectors inside the device memory
   cudaMemcpy(device_A, device_B, size, cudaMemcpyDeviceToDevice);

// GPU Computation...

// Copy vectors from device memory to host memory
   cudaMemcpy(host_B, device_B, size, cudaMemcpyDeviceToHost);
```

Code 1. Memory allocation and transfers [3]

"cudaMemcpy" directive performs transfer between host and device, but also internal high-speed transfers. Last parameter clearly specifies the direction and the nature of the transfer. MemcpyHostToDevice and MemcpyDeviceToHost are basic forward and backward transfer and they are used usually at the beginning of the GPU program and at the end, the results are computed. Last one, MemcpyDeviceToDevice is used for relocation in the GPU main memory. First two transfers are limited by the speed of the PCI-Express Bus. [Tab 1.]

PCIe Architecture	Raw Bit Rate	Bandwidth	Bandwidth for x16
PCIe 1.1	2.5 GT/s	2 GB/s	8 GB/s
PCIe 2.0	5 GT/s	4 GB/s	16 GB/s
PCIe 1.1	8 GT/s	8 GB/s	32 GB/s

Tab 1. PCI-Express speed specification

However, internal GPU bus can handle device to device memory transfer at the speed up to 317 GB/s. [5]

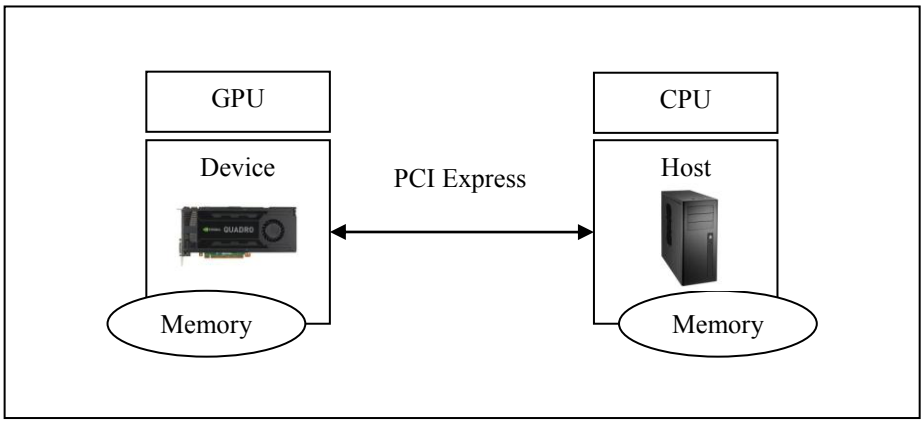

Fig. 2. Architecture scheme

2 Measuring computation time and time of transfers

CPU timers are quite useless according to the measuring GPU kernel times, because it is not possible to measure time of GPU program partitions. They can only provide summary run-time information.

CUDA has its own GPU timers system which allows much precise timing analysis. Application Interface (API) provides calls that can create and destroy time stamped events. It is possible to covert time stamp difference to the floating point value in milliseconds. Resolution is approximately half a microsecond. The timings

are measured on the GPU clock, so the timing resolution is operating system independent.

```
cudaEvent_t start, stop;
float time;

//Start and stop events
cudaEventCreate(&start);
cudaEventCreate(&stop);

//Record the time stamp of start
cudaEventRecord(start,0);
kernel<<<grid,threads>>> ();

//Record the time stamp of stop
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&time,start,stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Code 2. Kernel time measuring with CUDA events

```
cudaEvent_t start, stop;
float time;

//Start and stop events
cudaEventCreate(&start);
cudaEventCreate(&stop);

//Record the time stamp of start
cudaEventRecord(start,0);

// Host to Device transfer
   cudaMemcpy(device_A, host_A, size, cudaMemcpyHostToDevice);

kernel<<<grid,threads>>> ();

// Device to Host transfer
   cudaMemcpy(host_B, device_B, size, cudaMemcpyDeviceToHost);


//Record the time stamp of stop
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&time,start,stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Code 3. Kernel and transfers time measuring with CUDA events

According to this time measuring system, benchmarking serial CPU and parallel GPU program may process misleading results if the transfers are not included. CPU reads and writes are performed in global memory of CPU. GPU processing is not done, until the results are written in global memory of host. It is particularly important also measure the time of "Device to Host" and "Host to Device" transfers. The following section is providing correct CPU and GPU comparison process.

We used embarrassingly parallel problem to measure processing time for GPU and CPU and direct performance comparison. Algorithm is processing long sequences of high resolution images. It combines two images by color altering operations (per pixel) and generates the result, which is new image. There is no dependence among the pixels and all the images are independent. Image pairs are clearly distinguishable by the name. The dependence of the pair is tied before the processing and does not require further processing. This dataset structure generates embarrassingly parallel problem and GPU parallel processing should be very efficient in this case. Algorithm structure follows generic GPU program structure:

1. Memory allocation
2. CPU run
3. "Host to Device" transfer
4. GPU kernel run
5. "Device to Host" transfer
6. Free memory

We applied two measuring schemes:

I.	II.
1. Memory allocation	1. Memory allocation
-> CPU timer start	*-> CPU timer start*
2. CPU run	2. CPU run
-> CPU timer stop	*-> CPU timer stop*
	-> GPU timer start
3. "Host to Device" transfer	3. "Host to Device" transfer
-> GPU timer start	
4. GPU kernel run	4. GPU kernel run
-> GPU timer stop	
5. "Device to Host" transfer	5. "Device to Host" transfer
	-> GPU timer stop
6. Free memory	6. Free memory

Kernel runs as many times as many pairs need to be processed. This structure generates multiple nested loops with CPU processing and it requires intensive sequential computation. GPU performs this task in parallel fashion and should be much more efficient.

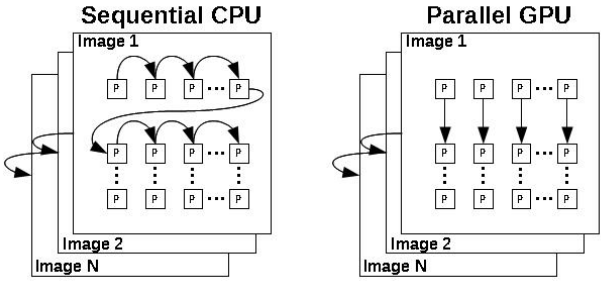

Fig. 3. Sequential CPU vs. Parallel GPU (P-Pixel)

2.1 Comparing results

Acquired results display parallel GPU advantage concerning only time of computation. Sequential CPU processing starts to fall behind as the number of images rises, almost linearly. GPU performs efficiently even for the extensive amount of processed images. The bigger the size of dataset is, the more pronounced the difference is between CPU and GPU processing.

Unfortunately, declaring this solution as a successful GPU acceleration is actually not correct. Considering the time of the "Host to Device" and "Device to Host" transfers change the outcome dramatically. CPU results will not change because transfers for the CPU remain the same. Time of GPU transfers will be added to the processing time and the advantage of GPU processing vanishes.

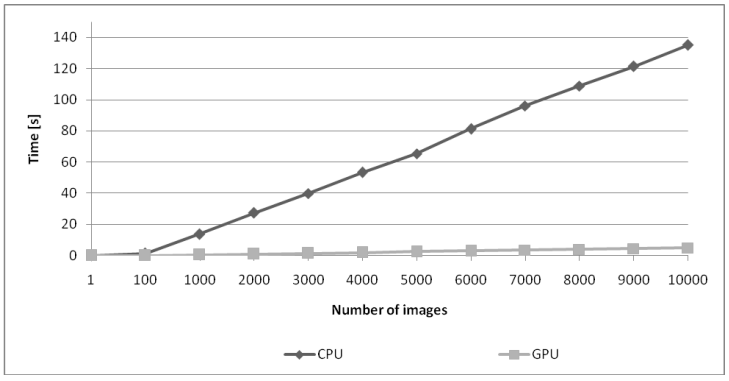

Fig. 4. CPU and GPU without transfer comparison

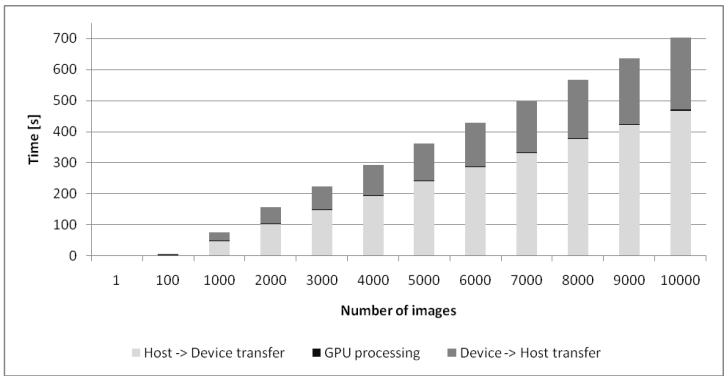

Fig. 5. CPU and GPU transfers included [4]

GPU computation is so efficient that it only takes fraction of time. Transfers, on the other hand are time consuming operations. One of the possibilities is to hide the transfers behind the computation. In this case there is simply not enough GPU processing to overlap the transfers. [2] There are two solutions. First, use the smaller dataset to reduce the transfers or another possibility is to perform more complicated algorithm which generates more intensive use of GPU computational resources. Downsizing the data set is not an option because all the images have to be processed and also CPU version will perform better with the smaller dataset. We modified algorithm to perform original color alteration multiple times. There is no practical use of the repetition with the same alteration in this case. We performed it just for the experimental purpose to see if the latency of transfers can be effectively hidden.

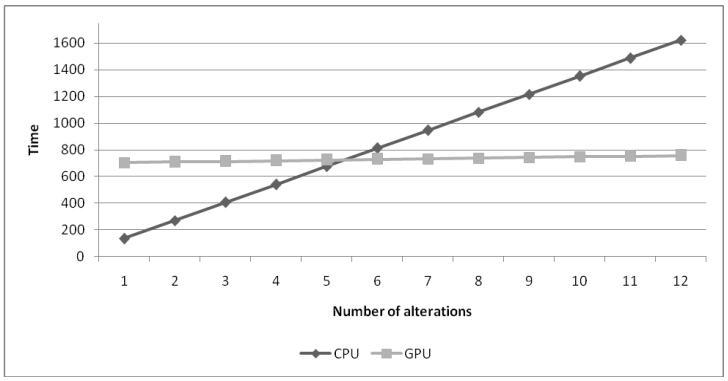

Fig. 6. CPU and GPU repeated alterations

Figure 6 shows that there is the point where the number of performed alterations actually raises the complexity of the computation to the level where it is possible to successfully hide the latency of transfers. If transfers remain constant, the complexity of performed algorithm has direct impact on computation. CPU has to resolve more complex nested loops which multiply result time significantly. On the

other hand, GPU performs very well. Complexity in this case has minimal impact on GPU processing time. Main idea is not to speed up GPU computation, but compute the problem which is complex enough to hide the latency of the transfers and slow down serial CPU solution to the level that GPU performs better even with included transfers. This approach requires the problem which is big enough concerning the dataset and complex enough to employ the GPU on the required level. Dataset does not have to be gigantic in the terms of amount, but it has to have dependencies which do not prevent successful parallelization, but raise the complexity of processing.

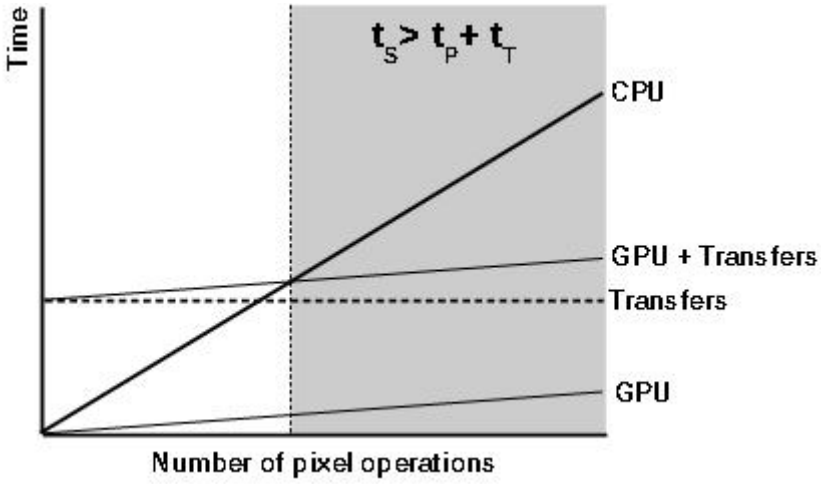

Fig. 7. Hiding the latency and over performing CPU
t_S-Sequential time, t_P-Parallel time, t_T-Time of transfers **[4]**

Complexity parameter can be expressed as a ratio between serial and parallel execution time. It shows advantage or disadvantage of the parallel solution compared to the serial one.

$$K = t_s / t_p => t_s / t_p + t_T$$

K - Problem complexity, t_S-Sequential time, t_P-Parallel time, t_T-Time of transfers

The greater the value of K is the more pronounced the advantage of GPU acceleration is. Values form interval <0,1> indicate that the problem is not suitable for parallel acceleration. The closer to the zero the value is, the less effective parallel solution is. All values outside interval <0,1> require further analysis. Parallel execution time has to be merged with the transfer time t_T. **[4]**

2.1 Other transfer optimization approaches

Concurrency is the key element of hiding the latency of transfers in the process of optimization GPU programs. Some recent devices can perform copies between page locked host and device memory spaces concurrently with the kernel execution. They can also perform a copy from page-locked host memory to device memory concurrently with a copy from device memory to page-locked host memory. Applications manage concurrency through streams.

```
//Memory copy, two streams
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);

//Kernel launch, two streams
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
    for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
```

Code 4. Two streams, memory transfer and kernel launch concurrency [3]

Multi device concurrency assumes that the host is equipped with more than one device. It is possible to select specific device to execute the kernel. Streams can be used for concurrent kernel execution.

```
cudaSetDevice(0);             // Set device 0 as current
cudaMalloc(&p0, size);        // Memory allocation on device 0
MyKernel<<<1000, 128>>>(p0); // Kernel launch on device 0
cudaSetDevice(1);             // Set device 1 as current
cudaMalloc(&p1, size);        // Memory allocation on device 1
MyKernel<<<1000, 128>>>(p1); // Kernel launch on device 1
```

Code 5. Multi device kernels execution [3]

Peer-to-Peer Memory Copy is model and architecture specific feature and allows dereferencing pointer to the memory of the other device. This feature is supported strictly between two devices. It can also be supported by the use of advanced network interconnection like Infiniband or HyperTransport. Also memory copy can be performed directly between two devices. [1]

```
cudaSetDevice(0);                   // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);              // Allocate memory on device 0
cudaSetDevice(1);                   // Set device 1 as current
float* p1;
```

```
cudaMalloc(&p1, size);              // Allocate memory on device 1
cudaSetDevice(0);                   // Set device 0 as current
MyKernel<<<1000, 128>>>(p0);        // Launch kernel on device 0
cudaSetDevice(1);                   // Set device 1 as current
cudaMemcpyPeer(p1, 1, p0, 0, size); // Copy p0 to p1
MyKernel<<<1000, 128>>>(p1);        // Launch kernel on device 1
```

Code 6. Peer-to-Peer memory copy [3]

3 Conclusions

Graphics adapters have brought the power of parallel computing to the wide range of general-purpose programmers. Quickly adoptable application interface also plays an important role and keeps learning curve fairly steep. CUDA programming directives can be easily integrated within many current high-level programming languages. However, there are many design and configuration challenges related to the problem transformation. Probably the baggiest optimization challenge is represented by the inter host-device and device-device communication and transfers during the kernel execution. Transfers' latency can degrade parallel solution, even for embarrassingly parallel problem, to the level with zero acceleration. It is important to minimize communication and utilize the complexity of the problem to successfully outperform CPU solution and achieve reasonable level of GPU acceleration. Parallel programming, using graphics adapter shows promising outcomes, and surely represents current trend of raising the computing power for general-purpose problems. New hardware designs and advanced communication technologies could bring significant improvements in the field of communication latency reduction.

References

[1] MELLANOX.: MELLANOX Technologies. MELLANOX, 2010, [cit. 2015]. Dostupné na webovskej stránke (world wide web): http://www.mellanox.com/

[2] HARRIS, M.: CUDA Occupancy Calculator, 2007, [cit. 2015]. (world wide web): http://forums.nvidia.com/index.php?showtopic=31279

[3] NVIDIA.: NVIDIA CUDA: Programming Guide. Nvidia, 2015, [cit. 2015]. (world wide web): http://docs.nvidia.com/cuda/cuda-c-programming-guide

[4] OCKAY, M.: Hardware HPC accelerators. The Technical University of Košice: Thesis, 2012.

[5] NVIDIA.: NVIDIA Quadro GPUs. Nvidia, 2015, [cit. 2015]. (world wide web): http://www.nvidia.com/object/quadro-desktop-gpus.html