Performance Evaluation of Data Migration Methods Between the Host and the Device in CUDA-Based Programming

Rafael Silva Santos, Danilo Medeiros Eler and Rogério Eduardo Garcia

Abstract CUDA-based programming model is heterogeneous – composed of two components: host (CPU) and device (GPU). Both components have separated memory spaces and processing units. A great challenge to increase GPU-based application performance is the data migration between these memory spaces. Currently, the CUDA platform supports the following data migration methods: UMA, zero-copy, pageable and pinned memory. In this paper, we compare the zero-copy performance method with the other methods by considering the overall application runtime. Additionally, we investigated the aspects of data migration process to enunciate causes of the performance variations. The obtained results demonstrated in some cases the zero-copy memory can provide an average performance on 19 % higher than the pinned memory transfer. In the studied situation, this method was the second most efficient. Finally, we present limitations of zero-copy memory as a resource for improving performance of CUDA applications.

1 Introduction

By considering the involved architectures, CUDA-based programming model is heterogeneous – composed of two components: host (CPU) and device (GPU) [5]. Both components have separated memory spaces and processing units. A great challenge for performance increasing in CUDA-based programming is the data migration between host and device memory spaces [7]. It is not a simple task to manage this data transfer, even without focusing in performance, since in some methods it is necessary to perform explicit requests for memory copy and the control of concurrent data access [8]. At present, the latest CUDA version provides four main methods for data migration between host and device memory spaces: zero-copy memory, UMA (Unified Memory Access) model, pageable and pinned memory [11].

R.S. Santos · D.M. Eler(✉) · R.E. Garcia
Faculdade de Ciências e Tecnologia, UNESP - Univ Estadual Paulista,
Presidente Prudente, SP, Brazil
e-mail: rafael.silva.sts@gmail.com, {daniloeler,rogerio}@fct.unesp.br

S. Latifi (ed.), *Information Technology New Generations*,
Advances in Intelligent Systems and Computing 448,
DOI: 10.1007/978-3-319-32467-8_60

In this paper, we investigate the zero-copy migration method, by comparing the performance with the other three memory migration methods provided by CUDA API and evaluating all runtime tests. We conducted a case study for the migration process. First, we developed an application to perform a scalar multiplication and the sum from the elements of a vector. Then, we adapted this application to execute and to evaluate two situations. In the first, less memory access is performed by the GPU during threads execution; in the second, a greater number of access transactions is performed. Finally, we present the limitations of using zero-copy memory method as an approach to increase a CUDA application performance.

Besides the introduction, this document consists of another five sections. Related works are introduced in Sect. 2. In Sect. 3, we describe the main data migration methods between host and device on the CUDA architecture. Then, in Sect. 4, we present the results. In this section, we evaluate and discuss the performance of the methods described in Sect. 3, comparing them with zero-copy memory method. In Sect. 5, we present some limitations in the use of zero-copy memory. Finally, Sect. 6 concludes this paper.

2 Related Works

In the last years, various studies on the transfer of data between the different architectures of the GPU-based programming model were produced. Kaldewey et al. [4] conducted a study on the efficiency of use of the bandwidth of the PCI-E bus in communication between the host and the device in the UVA model. By zero-copy memory they demonstrated the performance is close to the theoretical maximum bandwith memory. However, in that study, a comparison of the impact of the use of this memory in the global application performance compared to other transfer methods was not performed. Bai et al. [1] used the zero-copy memory to optimize algorithms performance for lattice-based cryptographic systems. In the tests, they analyzed both single GPU and multiple GPUs communicating with the CPU. Landaverde et al. [7] conducted a research of UMA model performance from the pageable memory transfer. In the methodology, they established a benchmark model to the methods similar to the model used in this work. Based on the results, they demonstrated that the use of UMA model causes a performance loss regarding to the pageable and pinned memory.

3 Data Migration Between Host and Device

3.1 CUDA Main Memory Access Model

Compute Unified Device Architecture (CUDA) is a platform developed by NVIDIA to allow use of GPUs in general-purpose computing. CUDA-based programming

Fig. 1 Heterogeneous CUDA-programming model

model is heterogeneous and each component, host and device, owns processing unit and memory space [11].

The rate that a processing unit can access the main memory is limited by its self **memory bandwidth**. In CUDA-based programming model, in addition to memory bandwidth of GPU and CPU, there is a **bus** between the host and device, as shown in in Fig. 1. Typically, the bus that mediates this communication is the PCI (Peripheral Component Interconnect). The PCI memory bandwidth is generally less than CPU and GPU [10]. Thus, it is important to analyze the memory transfer impacts on the design of a CUDA application.

3.2 CUDA Data Transfer Methods

Along the versions, NVIDIA introduced different memory management methods on the CUDA platform. By the time this study was conducted, the latest version of CUDA SDK – CUDA 7 SDK – offers the following methods: pageable memory, pinned memory, zero-copy memory, and Unified Memory Access (UMA) model [11].

Standard Transfer Method: Standard transfer method can be executed with pageable memory or pinned memory. In a **pageable memory**, during the execution of an application, the physical address of the data may change, given the memory pages may undergo swap for the secondary memory. In this way, before the data is copied from the host to the device, architecture migrates the desired data portion for pinned memory buffer on the host [11]. Then, through the PCI resource Direct Memory Access (DMA) performs the data transfer from the buffer to the device [3]. In contrast to pageable memory, **pinned memory** does not undergo swap. Thus, in standard transfer method of pinned memory, it does not occur to data migration to the buffer. This feature allows the PCI use DMA to directly transfer the data from the current physical location in host memory to the device memory space [10]. In pageable memory, the additional transfer contributes to a decrease in performance (memory bandwidth) regarding to pinned memory [2].

Zero-copy Method: Zero-copy memory is a "kind" of pinned memory provide by Unified Virtual Addressing (UVA). The use of zero-copy method discards the need

to explicit requests (i.e., a function call) for data migration. UVA model provides a single shared virtual space of memory between the host and the device, providing the address of the data on the host is mapped to the device [9]. Data migration occurs implicitly whenever a portion that is not presented in context is referenced, either the host or the device [10]. Thus, the zero-copy memory is used when a set of data does not fit completely in the device memory; once the CUDA API maintains only a portion of data that is accessed at a specific time in the device [6]. The remaining data set is maintained in host memory [10]. By considering the use of multiple GPUs, UVA model also provides a shared memory space and eliminates the intermediate step to copy the host memory during data transfer from one GPU to another. In the state of the art, the aspects of data movement in zero-copy memory provide by CUDA API are not disclosed.

UMA Model: Unified Memory Access (UMA) model is similar to zero-copy memory regarding no need for an explicit request for data migration. On the other hand, the data is not allocated in a pinned memory. As in zero-copy memory, the API is responsible for managing the entire data migration process.

4 Case Study: Evaluation of Memory Management Methods

In this section, we evaluate the performance of data migration between the host and device memory spaces. In addition to the performance comparison, we investigated the aspects that justify the variation between migration methods.

In the tests, we setup two different situations for the scenario: in the first, described in Subsect. 4.1, we created a simulated application in which the GPU makes a small amount of memory access transactions for each kernel thread; in the second, we adapted this application to the GPU performs a greater amount of memory access transactions, as shown in Subsect. 4.2.

4.1 Situation 1: Less Amount of Memory Access Transactions During a Thread Execution

Application Model: For the tests, we have developed an application that performs scalar multiplication and sum from vector elements. The implemented algorithm can be divided into the following steps:

1. Initialization of vector elements
2. Data transfer to the device (HtoD)
3. Vector scalar multiplication (kernel)

4. Data transfer to the host (DtoH)
5. Sum of vector elements

The step that comprehends the vector scalar multiplication was parallelized and runs on the GPU. The code snippet that is executed in this step can be seen in Fig. 2.

```
1  _ _global_ _ void mult(int *vect, int num, int N){
2          //Thread index
3          int id = blockIdx.x * blockDim.x + threadIdx.x;
4          //Multiplication = vector element * constant value
5          if (id < N)
6                  *(vect + id) = *(vect + id) * num;
7  }
```

Fig. 2 Code snippet that runs on the device for vector multiplication by a constant.

The host is responsible for the initialization steps and sum of vector elements. These steps are performed sequentially. Throughout the algorithm, all steps use the vector as the Input and Output Dataset. After the initialization of the vector, it is necessary to migrate this set of data from the host to device (HtoD). Then, after multiplication by a scalar value, the vector should be migrated back to the host (DtoH). The context of CUDA application comprehends a single stream. A pseudo-code containing all algorithm steps is presented in Algorithm 1. The code snippet that contains the parallelized instructions was previously shown in Fig. 2.

To compare and investigate the performance of each memory management method, we adapted the application with the necessary function calls to allocate and copy memory. It is worth mentioning that we focus exclusively on evaluating the performance differences and research aspects of the data migration between the host and the device. Thus, the implemented algorithm is not complex and does not require a large computational effort.

Algorithm 1. Scalar multiplication and sum of vector elements

Require: $n > 0$
Ensure: $vec \leftarrow vec * c$
  $vec \leftarrow Allocation$
  $c \leftarrow k_1$ {k_1 is a constant}
  for $i = 1, i \leq n, i++$ **do**
    $vec[i] = k_2$ {k_2 is a constant}
  end for
  $\Rightarrow$ *migration data* – HtoD
  mult<<<N, 1>>> (vec, c, n) {CUDA kernel}
  $\Rightarrow$ *migration data* – DtoH
  $sum = 0$
  for $i = 1, i \leq n, i++$ **do**
    $sum = sum + vec[i]$
  end for

Methodology: We developed an application which performs a scalar multiplication and sum from the vector elements (see the Subsect. 4.1) for our microbenchmarks. NVIDIA Visual Profiler and nvprof tools were used to measure processing time and to obtain details of the data transfer aspects throughout the runtime. We run the application with four different configurations of data migration methods provided by CUDA platform.

In the tests, we use five different vector sizes. The arrangement of threads and threads blocks on kernel function was performed based on these sizes. Number of elements in each vector size: 1048576 (1024 threads x 1024 blocks), 2097152 (1024 threads x 2048 blocks), 4194304 (1024 threads x 4096 blocks), 8388608 (1024 threads x 8192 blocks) and 16777216 (1024 threads x 16384 blocks) elements.

The timeline during runtime for each test was divided into five ranges as the steps that have been described in the implemented algorithm. To define each time range, we use NVTX (NVIDIA Tools Extension). We named these ranges as follows: initialization, HtoD transfer, multiplication, DtoH transfer and sum. It is easy to associate each range with the described algorithm steps, which are presented in Subsect. 4.1. Data transfer ranges were not considered in the UMA model and the zero-copy memory, once there is no explicit memory copy in these methods.

All results represent an average of five executions. All the tests were performed on computer with Windows 8.1 64 Bits Operating System, Intel I5-2320 CPU, 4 GB RAM, PCI Express x16 2.0 bus, NVIDIA Geforce GT 740 GPU, with 1 GB RAM DDR3 and 384 CUDA cores. The application was implemented using the CUDA 7 SDK.

Experimental Results: In this section, we present results of the tests. In order to provide a better understanding, all values were normalized.

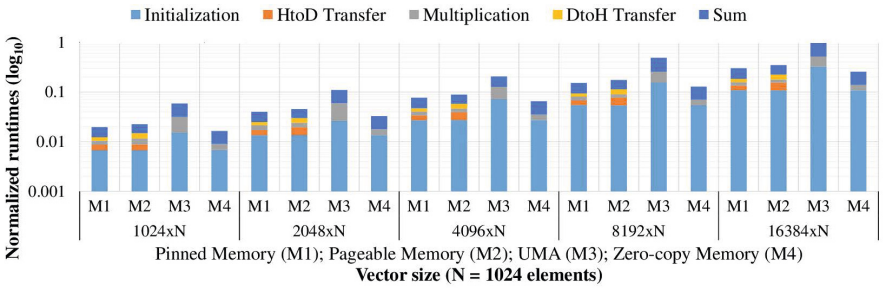

Fig. 3 Global normalized runtime for the application of scalar multiplication and addition of vector elements (Situation 1).

Figure 3 shows the normalized runtimes for application that performs the scalar multiplication and the sum from the vector elements. In particular, the results match the configured application in four memory management methods: pageable memory (represented by M1), pinned memory (represented by M2), UMA

(represented by M3) and zero-copy memory (represented by M4). Apart from the global runtime, it is also shown the runtime of each range that the application was divided: initialization, HtoD transfer, multiplication, DtoH transfer and sum.

From Fig 3, we can see that there was a distinct variation in application performance among the data migration methods. Additionally, it is possible to observe the results were consistent, once the performance variation pattern in retained once the array size is increased. From the obtained results, we ordered the migration data methods regarding best described performance: zero-copy memory, UMA model, pageable memory and pinned memory. In all presented tests, this order of efficiency is the same. On average, the application configured with the standard transfer method with pinned memory spends 19.40% more time than zero-copy memory; with pageable memory, 37.24% more time was spent; and in UMA model, it was 256.57%.

In order to figure out the reasons for performance variations among the methods, we evaluate aspects of data migration during tests runtime. In the tests, the configuration with pageable memory spends on average 62.47% more time than the time spent by the pinned memory in HtoD transfer range and 77.40% in DtoH transfer range. Through the NVIDIA Visual Profiler, it is possible to measure the throughput of pinned memory – on average 6.68 GB/s (HtoD) and 6.697 GB/s (DtoH), whereas in the pageable memory was 3.69 GB/s (HtoD) and 3.87 GB/s (DtoH). In the other analyzed ranges, the time consumed by both methods is the same. On average, there is a difference of 0.59% in relation to the run time in the initialization, 1.52% in multiplication, and 2.43% in the sum of vector elements. In the standard transfer method, the data migration does not affect the performance of other ranges that the timeline of the tests was divided. Thus, we used this method to compare and to investigate the data migration aspects and performance in UMA model and zero-copy memory.

In UMA model, there is no explicit memory copy. However, from Fig. 3, of course notice a high discrepancy between the processing time of this method regarding the standard transfer method, when we analyze the ranges of initialization, multiplication and sum from vector elements. On average, the UMA model spent 152% more time than the average of the pageable and pinned memories during initialization, 670% more than multiplication and 251% more than sum range.

In all tests, the zero-copy memory is more efficient. As in UMA model, there is no explicit memory copy. However, the data migration takes place at different times. Figure 4 shows the time consumed during the execution of multiplication range for zero-copy memory and standard transfer methods for all tests. As we can see, once the vector has more than 2097152 (2048x1024) elements, the kernel runtime with zero-copy method becomes greater than the time taken by the standard transfer method with pageable memory and pinned. NVIDIA Visual Profiler tool does not support a graphical analysis of the data migration with zero-copy memory. Although, it is possible to collect information about reading and writing transactions in system memory, i.e., in the host, while running the kernel. In all executed tests, regardless of the vector size, there are on average 2097152 (2048x1024) access transactions to host memory (reading and writing) for the kernel running with zero-copy memory. Each Access transaction features 32-bit width and the transfer rate is on average 5.88 GB/s.

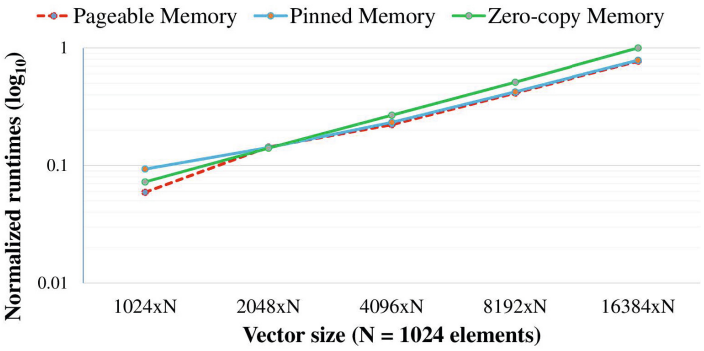

Fig. 4 Multiplication range runtime. Configured application with the memory management methods: zero-copy memory, pageable memory and pinned memory.

Discussion: The results demonstrate the data migration between host and device memory spaces effectively impacts the application performance. Additionally, different memory management methods provided by CUDA, exhibit high performance variation.

Zero-copy memory method was more efficient than other methods. However, we cannot assert that for any application of this method will be more efficient. The results demonstrate that the use of zero-copy memory affects the performance of the kernels, once that occur access transactions to host memory during the execution. Thus, to further investigate the behavior of this method, we modified the kernel function to run the tests again. The modified kernel and the results obtained are shown in Subsect. 4.2.

The pinned memory obtained the second best performance. Regarding pageable memory, the performance difference is caused by the memory throughput. Pinned memory has a transfer rate about 77% higher than the pageable memory. The performance of UMA was lower than all other methods.

4.2 Situation 2: Greatest Amount of Memory Access Transactions During the Execution of a Thread

In order to better investigate the behavior of zero-copy memory and another memory management methods, we adapt the original kernel function shown in Fig. 2. The modified code snippet can be seen in Fig. 5. The kernel modification does not cause changes in the application results. The modification added 49 redundant instructions of each thread execution. By performing this kernel, we intend to simulate an increased amount of memory access transactions while running the kernel. Note that the remaining steps of the algorithm (Algorithm 1) have not been modified. Thus, the produced results by the application are the same.

```
1  _ _global_ _ void mult(int *vect, int num, int N){
2            //Thread index
3            int id = blockIdx.x * blockDim.x + threadIdx.x;
4            //Multiplication = vector element * constant value
5            if (id < N)
6                    for (int i = 0; i < 50; i++)
7                            *(vect + id) = *(vect + id) * num;
8  }
```

Fig. 5 Code snippet of modified kernel.

Methodology: To perform the tests, we use the same methodology from the original application (see Subsect. 4.1), i.e., with unmodified kernel.

Experimental Results: Fig. 6 shows the normalized runtime for the application in Situation 2. In comparison with Fig. 3, which represents the application execution time in Situation 1, we can observe that the relative time consumed by the multiplication range increased in all tests. Moreover, in all other observed ranges (initialization, HtoD transfer, DtoH transfer and sum), the spent runtime was the same. In all tests, zero-copy memory has the lowest performance. Whereas, the standard transfer method with pinned memory is the most efficient.

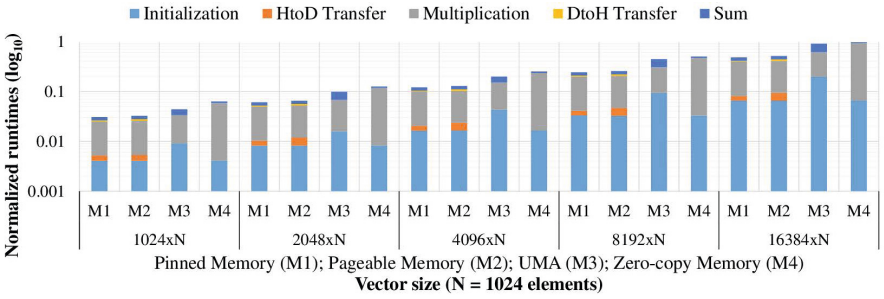

Fig. 6 Global normalized runtime for the application with kernel adapted (Situation 2).

The ratio of the multiplication range runtime with both modified kernel and unmodified kernel is shown in Figure 7. From this figure, we can see that the standard transfer method increases of about 20 times the runtime. Whereas the runtime increase in zero-copy memory was at least 40 times. Through NVIDIA Visual Profiler, we can observe in zero-copy memory, the kernel runtime increase is accompanied by an increase in the number of host memory access transactions. In all tests, the number of access transactions increased 50 times, which corresponds to the increased number of instructions in the modified kernel.

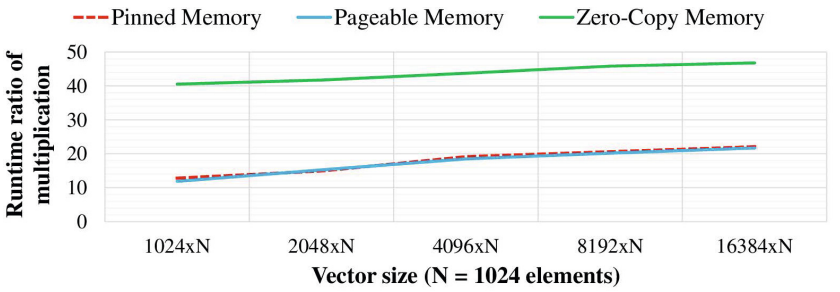

Fig. 7 Runtime ratio of multiplication range. Ratio between runtime multiplication in Situation 2 and Situation 1

Discussion: The tests demonstrated that an increase in access to data during kernel execution affects the migration of memory between the host and the device when using the zero-copy memory method. Additionally, we can observe that there is not a self optimization CUDA API when using zero-copy memory, once the kernel modification aims to increase the amount of memory access transactions and leads to the execution of redundant instructions.

5 Limitations of Zero-copy Memory Usage

The first limitation on the use of zero-copy memory resides in the fact that the data is allocated in a pinned memory. The allocation of large amounts of pinned memory can affect the operating system performance [10]. Unfortunately, it is not possible to measure a precise relationship between the amount of memory allocated and total memory installed in the system. Beyond the amount available memory, the operating system and other applications that are running in the environment influence the performance loss of the entire system. On this way, a good practice is not to use zero-copy memory when it is not known in advance, the maximum amount of data that will be allocated. As discussed in Subsect. 4.2, a problem in the use of *zero-copy* memory is the occurrence of performance loss when the amount of memory access transactions increases. Particularly, part of these transactions may include redundant copies performed on a kernel. In some cases, it can use a local variable within the kernel function that receives a copy of the data used for to avoid the redundant accesses. **Basically, the zero-copy memory must be used on data undergoing a lesser amount of access during transactions execution**. In case of more amount of access transactions, other transfer methods are recommendable.

6 Conclusion

Zero-copy memory presents implicit and transparent memory copies to the programmer, hiding the complexity of managing the data migration. Originally, this method was conceived as a feature that allows the use of data sets that may not be entirely stored in the memory device.

This study showed that in cases in which the kernel function performs a small amount of memory access transactions (in particular, a single access transaction), zero-copy memory can be used to provide performance increase. In certain cases, the use of zero-copy memory can provide a performance gain of more than 19% in the runtime application when compared to pinned memory. Based on the obtained results, we demonstrated that the total number of memory access transactions during execution of the kernel reduces the overall performance of the application and establishing a barrier in using zero-copy memory.

In the tested situations, we do not use multiple streams and memory copy process was not overlapped by the running kernel. Therefore, further works may include analysis of the performance of the zero-copy memory in a concurrent streams scenario. In a other future study, we will investigate the performance of zero-copy memory in other models of GPUs and also in the OpenCL API.

Acknowledgment This work was supported by Brazilian financial agency São Paulo Research Foundation (FAPESP) – grants 2015/00622-7 and 2013/03452-0.

References

1. Bai, T., Davis, S., Li, J., Jiang, H.: Analysis and acceleration of ntru lattice-based cryptographic system. In: 2014 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 1–6, June 2014
2. Fatica, M.: Accelerating linpack with cuda on heterogenous clusters. In: Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units,GPGPU-2, pp. 46–51. ACM, New York (2009)
3. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Fifth Edition: A Quantitative Approach, 5th edn. Morgan Kaufmann Publishers Inc., San Francisco (2011)
4. Kaldewey, T., Lohman, G., Mueller, R., Volk, P.: Gpu join processing revisited. In: Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, pp. 55–62. ACM, New York (2012)
5. Kim, Y., Shrivastava, A.: Memory performance estimation of cuda programs. ACM Trans. Embed. Comput. Syst. **13**(2), 21:1–21:22 (2013)
6. Kirk, D.B., Hwu, W.M.W.: Programming Massively Parallel Processors: A Hands-on Approach, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2010)
7. Landaverde, R., Zhang, T., Coskun, A., Herbordt, M.: An investigation of unified memory access performance in cuda. In: High Performance Extreme Computing Conference (HPEC), 2014 IEEE, pp. 1–6, September 2014

8. Li, W., Jin, G., Cui, X., See, S.: An evaluation of unified memory technology on nvidia gpus. In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 1092–1098, May 2015
9. Tang, K., Yu, Y., Wang, Y., Zhou, Y., Guo, H.: Ema: Turning multiple address spaces transparent to cuda programming. In: ChinaGrid Annual Conference (ChinaGrid), 2012 Seventh, pp. 170–175, September 2012
10. NVIDIA Corporation: CUDA C Best Practices Guide, March 2015
11. NVIDIA Corporation: CUDA C Programming Guide, March 2015