A Dynamic Protection Mechanism for GPU Memory Overflow

Yaning Yang, Xiaoqi Wang, Shaoliang Peng

A Dynamic Protection Mechanism for GPU Memory Overflow

Yaning Yang[1], Xiaoqi Wang[1], and Shaoliang Peng[1,2] *

[1] College of Computer Science and Electronic Engineering, Hunan University, Changsha, 410082, China.
[2] National Supercomputing Centre in Changsha, Changsha, 410082, China.
{yangyn,xqw,slpeng}@hnu.edu.cn

Abstract. Graphics Processing Units (GPU) are widely used to accelerate computation in many applications such as autonomous vehicles, artificial intelligence and healthcare. However, most existing researches just focus on the performance but ignore the security issues of GPUs. In this paper, we design an efficient mechanism to dynamically monitor GPU heap buffer overflow by using the CPU. Concretely, we first analyze the specific requirements of GPU memory allocation. Second, in order to realize the monitoring from the CPU, we map the allocated device memory to the host-side. Third, the dynamic monitoring of buffer overflow is implemented based on the mapped memory. Our results show that it is feasible to protect the GPU memory from the CPU side. Our work can improve the efficiency of GPU memory allocation and increase the security at the same time. By offloading the detection of buffer overflow to the CPU, the performance of GPU kernels will not be affected significantly.

Keywords: GPGPU · Memory Allocation · Security · Memory Overflow.

1 Introduction

With the development of parallel computing, more and more programmers are willing to choose GPUs from vendors such as NVIDIA for high performance computing. For example, GPUs are often used to accelerate scientific compute-intensive tasks and finance operations [1]. In the field of real-time embedded systems, GPUs also show good performance advantages [2]. Further, most encryption algorithms are implemented based on GPU [3, 4, 5]. In addition, GPUs can be virtualized [6], so many cloud computing providers provide users with GPU support, enabling users to share GPUs.

Indeed, GPUs offer tremendous advantages in parallel computing, especially in some complex computing areas such as artificial intelligence [7]. However, there are certain security vulnerabilities on the GPU [8, 9], such as buffer overflow. Buffer overflow is a software error caused by accessing data outside the

*

buffer. It is a long-standing and common software vulnerability. Once the buffer overflows, it may cause the program to fail, system down, restart, and so on. More seriously, it can be used to execute unauthorized instructions, and even to obtain system privileges, and thus carry out various illegal operations. Infamous security attacks such as Code Red, Slammer and Morris Worm are all based on buffer overflow. If these problems occur on the GPU, it may cause the leakage of sensitive data, especially when performing computational financial operations and encryption algorithms. At present, most of the research on the vulnerability of buffer overflow targets the CPU. NVIDIA provides a tool CUDA-MEMCHECK [10] as a part of the CUDA toolkit, which can be used to detect out of bounds and misaligned memory access errors in CUDA applications, but it is clearly stated that applications would run much slower if using CUDA-MEMCHECK to detect memory errors and it may lead to failures of kernel launches such as timeout.

In response to the existence of GPU buffer overflows and the high overhead of existing tools, we design an efficient mechanism that leverages the CPU to dynamically monitor GPU heap buffer overflows. First, because of the efficiency issue of *cudaMalloc* provided by CUDA [11, 12, 13], we use *ScatterAlloc* [14] to allocate memory on the GPU. Second, we leverage the unified memory mechanism to implement our system, which has the advantage of allowing the CPU to access the memory allocated on the GPU. Third, we insert different canaries to the head and tail of dynamically allocated buffers and create a dedicated monitoring thread on the CPU to continuously detect buffer overflow. Once the canary is modified, it indicates an event of buffer overflow. The reason we use the CPU to detect buffer overflows is that if both the monitoring thread and the user threads are performed on the GPU, resource contention will inevitably occur in highly concurrent applications, which will result in high performance overhead.

The remainder of the paper is organized as follows. In Section 2, we introduce the background about CUDA programming model. We describe the overall design of our system and the structure of the buffer in Section 3 and specific implementation details in Section 4. Experimental evaluation is given in Section 5 and conclude this paper in Section 6.

2 Background

2.1 CUDA Programming Model

CUDA is a general-purpose parallel computing architecture introduced by NVIDIA that enables GPUs to solve complex computing problems. It includes the CUDA Instruction Set Architecture (ISA) and the parallel computing engine inside the GPU. The CUDA code is divided into two parts, one running on the host (CPU), the normal C code, and the other part running on the device (GPU), which is called kernel. In CUDA, the host and device have different memory spaces, so when executing the kernel on the device, the programmer needs to explicitly transfer the data on the host memory to the device memory. After the kernel

is executed, the result needs to be transferred from the device memory back to the host memory and the device memory should be released. The CUDA runtime system provides APIs for programmers to perform these operations. A kernel is declared with the keyword *__global__*. A running kernel is composed of a large number of GPU threads, Threads are grouped into *blocks*, and blocks are grouped into *grids*. When the host calls a kernel, the programmer must set the dimensions of the grid and thread block with certain parameters.

2.2 Unified Memory

In the above programming model, it is necessary to separately define pointers for the host and the device and allocate memory separately on the host side and the device side, and to make an explicit copy between the CPU and GPU memory before and after the kernel is called. This procedure is tedious and error-prone. CUDA 6.0 introduces a feature called unified memory that greatly simplifies the implementation of CUDA applications [15]. The user only needs to define a pointer that can be used on both the host and the device, and the explicit memory copy is not need any more.

2.3 GPU Allocator Optimization

In terms of dynamically managing GPU memory, the current CUDA programming model supports the dynamic allocation and deallocation of device memory using APIs such as $cudaMalloc$ and $cudaFree$, but the GPU needs to interrupt the CPU execution during the process of memory allocation, which has a significant impact on the performance of data-intensive applications in high-concurrency environments. In order to reduce the blocking phenomenon between threads, several approaches can be used, such as optimistic concurrency control and multi-version concurrency control. For example, we can allocate a new memory space before modifying the data and copy the new data to this space for modification, then use it to replace the old version data. The invalid data is processed by the garbage collection mechanism. This mechanism greatly increases the frequency of memory allocation requests. Most experiments show that using $cudaMalloc$ and $cudaFree$ provided by CUDA for memory allocation is very inefficient [11, 12, 13, 14]. Therefore, many researchers have done a lot of work on GPU memory management and developed new allocators, such as XMalloc [11] and ScatterAlloc [14], which are optimized for dynamic memory allocation. Experiments show that ScatterAlloc is approximately 100 times faster than the CUDA toolkit allocator and up to 10 times faster than XMalloc. Therefore, we choose ScatterAlloc to manage the GPU memory in our system.

There are multiple global heaps in ScatterAlloc. The memory allocation request is directed to a different global heap by a hash operation, which reduces the probability of collisions accessing the global heap. ScatterAlloc divides memory into fixed-size pages and the pages are split into equally-sized chunks. In order to find free memory space in a page, ScatterAlloc employs a *pageusagetable* and every entry consists of three values: the chunk size, the number of allocated

chunks, and a bitfield. Each bit in the bitfield represents a single chunk of memory. These fixed-size pages are gathered in super blocks, which form the largest memory unit.

3 Design Overview

3.1 Buffer Structure

For buffer overflow detection, the most common way is to add a *canary* to the buffer structure, such as StackGuard [16]. In Cruiser [17], it adds different canary words to the head and tail of the buffer. Thus, we also add canary words to the buffer in a conventional way. The specific method is to add different canaries to the head and tail of the buffer, that is, the *headcanary* and the *tailcanary* as shown in Fig. 1. In addition to canaries, we also add encryption information about the buffer size to the buffer structure. If the head canary is changed, it means that the buffer is underflowed. Similarly, the buffer is overflowed if the tail canary is corrupted. The purpose of adding the buffer size information is to locate the tail canary given a buffer address. Because we encrypt the buffer size information, it will not be leaked to attackers. The value of head canary is the result of encryption of the head canary key, the buffer size, and the buffer address. The tail canary is calculated in the same way, but with another tail canary key. In this way, the head canary and the tail canary of different buffers are different. Even if the head canary and the tail canary of one memory block are leaked, other memory blocks are safe.

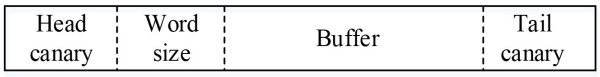


Fig. 1. Buffer structure.

3.2 Overview of the System

Our design goal is to increase the efficiency of GPU parallel computing as much as possible without compromising security due to GPU buffer overflow. Therefore, we propose to leverage CPU to monitor the GPU memory by separating the monitoring thread from the user threads. The advantage of this method is that the monitoring thread does not compete with the user threads for resources, allowing the user threads to perform the corresponding calculation with maximum efficiency and the data used by the user threads is also protected. This approach ensures that applications are both efficient and secure. As shown in Fig. 2, after a buffer is allocated, we encapsulate the address of the allocated buffer into an address collection. And the user threads running on the GPU can operate the

buffer normally and the monitoring thread running on the CPU will continuously monitor the corresponding buffer wrapped in the address collection. The whole process is mainly divided into three steps. First, the system allocates a buffer that is 3 words larger than what user thread requests, and the reason for adding extra 3 words is that we add two canaries and the buffer size information in the buffer structure as described in section 3.1. In the second step, we encapsulate the allocated buffer and then add the buffer information to an address collection. The monitoring thread traverses the entire address collection to determine if there are buffer overflows. In the third step, when a buffer is freed, we mark the first word of the buffer as released. The monitoring thread then removes this buffer from the address collection when it checks the buffer and finds that the buffer is marked released.

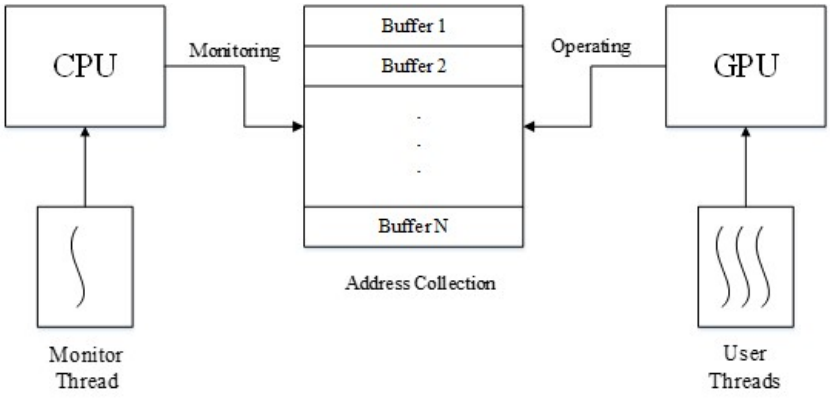


Fig. 2. System architecture.

There is a shortcoming in our system, which is that we check the canaries after the kernel is finished, because real-time detection may lead to high overhead due to frequent transmission of memory pages between the CPU and GPU. Concurrent accesses to the same memory page may lead to page swapping between the two ends. For example, a dirty page on the GPU would cause a swapping to the CPU when the monitoring thread reads the same page. In our future work, we plan to optimize this procedure by analyzing memory access patterns and make our system an online solution.

4 Implementation

4.1 Memory Allocation

In this paper, we allocate memory space by using the ScatterAlloc allocator. First, we allocate a large memory pool which is unified memory by calling the

cudaMallocManaged function and return a void pointer, see Code 1 (from line 1 to 5). Second, we declare a pointer by adding the *__managed__* keyword, which points to a block of unified memory (line 6). The actual allocation is done in a kernel (from line 7 to 9) which is called in a host function (line 13). In the original ScatterAlloc, a large memory pool was allocated by calling *cudaMalloc*, which can only be read/written on device side. To allocate memory blocks on this memory pool, it needs to read and write this memory pool. So the ScatterAlloc defined its allocation function on the device side and it was called by the kernel. We just copied this manner of allocation. Because we encapsulate a buffer by adding two canaries and buffer size information, the actual allocation size is increased by 3 extra words.

Code 1: Memory Allocation

```
1  static void* setMemPool(size_t memsize){
2   void* pool = NULL;
3   cudaMallocManaged(&pool, memsize);
4   return pool;
5  }
6  __device__ __managed__ unsigned long *buffer;
7  __global__ void alloc(size_t size, AllocHandle mMC){
8   buffer = (unsigned long*) mMC.malloc(size);
9  }
10 void run(){
11  ...
12  ScatterAllocator mMC(1U*1024U*1024U*1024U);
13  alloc<<<blocks,threads>>>(size + 3, mMC);
14  cudaDeviceSynchronize();
15  ...
16 }
```

4.2 Buffer Structure Construction

We have already described the structure of the buffer in Section 3.2, see Code 2 for the specific implementation. First, we declare a pointer p that is used to encapsulate the buffer *addr* (line 3), then insert the head canary, tail canary, and buffer size information, which are encrypted results with encryption algorithms (from line 4 to 6). Finally, the encapsulated buffer structure is added to the thread record list by calling the *produce* function.

Code 2: Constructing Buffer Structure

```
1  inline void afterMalloc(void* addr, size_t word_size){
2   ...
3   unsigned long *p = (unsigned long *)addr;
4   p[0] = head_canary ^ new_word_size;
5   p[1] = new_word_size;
```

```
6  p[2 + word_size] = tail_canary ^ new_word_size;
7  Node node;
8  node.userAddr = p + 2;
9  t_threadRecord=g_threadrecordlist->getThreadRecord();
10 t_threadRecord->produce(node);
11 }
```

In some highly concurrent applications, memory allocation will occur frequently, so the $afterMalloc$ function will be called multiple times. In order to eliminate the overheads of function calls, we define it as an inline function by adding the keyword *inline*.

4.3 Buffer Overflow Detection

Once a buffer is released, the encapsulation of the buffer address information in the address collection is expired and should be removed. If a buffer is about to be released, we set the value of its first element (head canary) to zero. A buffer overflow depends on the value of the canaries we inserted before. The method $beforeFree$ in Code 3 is exactly the opposite of $afterMalloc$ as described in Section 4.2. First, we define a pointer variable p that is used to decapsulate the buffer address. The next step is to check the first two elements (the head canary and buffer size information) and the last element (tail canary) of the buffer, respectively. Because we mark the first element of the buffer that has been released as zero, it indicates a double free if the value of $p[0]$ is detected as 0 (from line 4 to 7). When the monitoring thread checks a buffer and finds the value of $head\_canary\_free$ stored in the address collection is not the same as the precomputed value by decrypting the $head\_canary$, the buffer is underflowed. Similarly, the buffer is overflowed if the $tail\_canary$ is corrupted. Once the buffer is overflowed, the monitoring thread will call the function $attackDetected$ to abort the application (from line 8 to 16). In addition, if the buffer size information ($p[1]$) is changed, the tail canary would not be located correctly. As a result, reading the tail canary may incur segmentation fault, which essentially exposes buffer overflows.

Code 3: Buffer Overflow Detection

```
1  inline static void beforeFree(void* addr){
2   ...
3   unsigned long *p = (unsigned long*)addr - 2;
4   if(!p[0]){
5    fprintf(stderr, "Duplicate frees are detected\n");
6    return;
7   }
8   size_t word_size = p[1];
9   unsigned long head_canary_free = p[0];
10  unsigned long tail_canary_free = p[2 + word_size];
11  if(head_canary_free != head_canary ^ word_size){
```

```
12    attackDetected(addr, 2);
13   }
14   if(tail_canary_free != tail_canary ^ word_size){
15    attackDetected(addr, 1);
16   }
17   ...
18   p[0] = 0;
19  }
```

5 Evaluation

This section reports the performance of the system, including memory allocation overheads. Our experimental setup is described in Section 5.1. In Section **??**, we show the feasibility of the experiment leveraging CPU to dynamically detect GPU buffer overflows. We evaluate the performance overheads in Section 5.2.

5.1 Experiment Setup

Our experiment was performed on Ubuntu 16.04.1. The CPU is Intel(R) Xeon(R) E5-2630 v3 clocked at 2.40 GHz and the host memory size is 32G. The GPU is the NVIDIA GeForce GTX 1070 (Pascal architecture) that has computing capability 6.1, 1920 CUDA Cores, 8 GB of GDDR5 memory. The CUDA runtime version is 9.0. We use *nvcc* to compile CUDA code.

5.2 Performance Analysis

Additional impact on system performance is mainly caused by memory allocation and buffer overflow detection. For memory allocation, we use *ScatterAlloc* allocator to allocate memory. As the number of threads continues to increase, the performance of the allocator almost remains essentially at a constant level. In the case of full use of the GPU, the memory allocation using ScatterAlloc is 100 times faster than the CUDA toolkit allocator, 10 times faster than using SIMD optimized XMalloc [11]. On the other hand, related studies have shown that unnecessary data copies take 16% to 537% longer to execute than the actual data movement [18], which is not acceptable in high concurrency and data-intensive applications that employ GPUs. This inefficiency is solved in our system by declaring unified memory which can be accessed by the CPU and GPU at the same time. It is not necessary to use *cudaMemcpy* to copy data between CPU and GPU.

We use *ScatterAlloc* to allocate a series of buffers on the unified memory, the buffer size is 4096B. And we use *cudaMalloc* to allocate buffers of the same size and number on the global memory and perform a simple *kernel* that calculates the sum of two matrices with 4096 threads. Experiments show that the ratio of the overheads of the two methods is about 60% on average. As we can see from

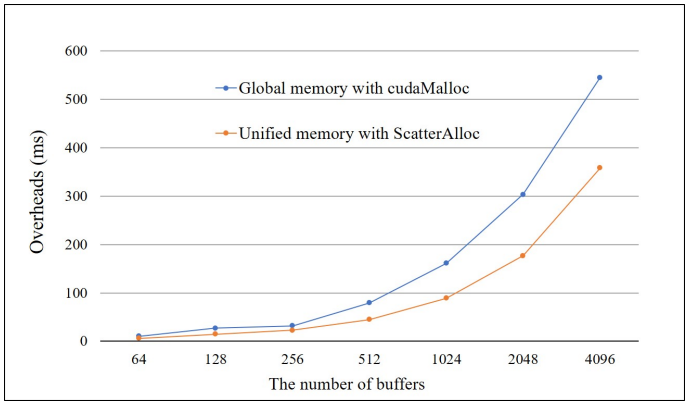
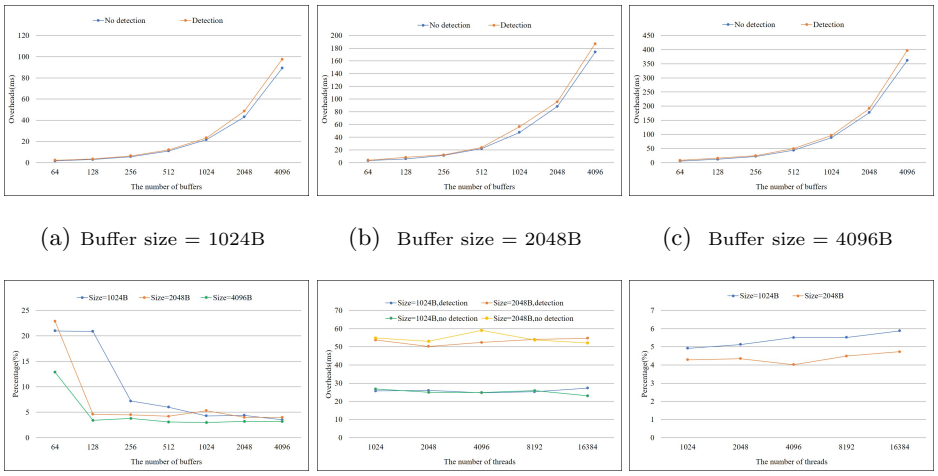


Fig. 3. Overheads about cudaMalloc and ScatterAlloc.



(a) Buffer size = 1024B (b) Buffer size = 2048B (c) Buffer size = 4096B

(d) Proportion of detection time (e) Overheads with different number of threads (f) Proportion of detection time with different number of threads

Fig. 4. Overheads of buffer overflow detection

Fig. 3, the overhead of the first method is significantly lower than that of the second one.

For the performance overheads of buffer overflow detection, we test the impact of the buffer overflow detection on the overall application in the case of different sizes and the number of allocated buffers with a fine-tuned benchmark called CUDA-quicksort [19] with 16384 threads. All results reported in this section are average values of 10 runs. A large number of experiments have proved that our method has little impact on system performance. As we can see from the Fig. 4(a), 4(b), 4(c), when the buffer size is less than 512B, the overhead

when enabling the overflow detection is almost negligible. Even when the buffer size is larger than 512B, the overhead incurred by buffer overflow detection is only slightly higher than that when the detection is disabled. Fig. 4(d) shows how the overhead of buffer overflow detection varies with different buffer sizes. When the buffer size is increased from 64B to 512B, the overhead caused by the detection is gradually decreasing. As the size is larger than 512B, the overhead remains basically unchanged at around 4%.

We also test the impact of the number of GPU threads on performance. In Fig. 4(e), we allocate 1024 buffers of 1024B and 2048B, respectively, and measure the performance overhead of overflow detection as compared to the case when detection is disabled. In this experiment, the number of threads is variable. The experimental results show that under the same conditions, when the number of threads increases from 1024 to 16384, the overheads remain basically unchanged. The detection overhead varies between $4\% \sim 6\%$ as shown in Fig. 4(f). This means that the number of user threads does not affect the performance of overflow detection significantly.

6 Conclusion

In this paper, we discuss the issue of GPU memory management, including the optimization of memory allocation and the use of unified memory, and we study the GPU buffer overflow problem. On this basis, we propose a new mechanism for dynamically detecting GPU memory overflow and design a prototype system that uses the CPU to detect GPU buffer. Our tests show that in high-concurrency and data-intensive applications, the performance overhead is only about $4\% \sim 6\%$. With this mechanism, the overflow of GPU heap memory can be effectively prevented.

Acknowledgment

This work was supported by NSFC Grants U19A2067, 61772543, U1435222, 61625202, 61272056; National Key R&D Program of China 2017YFB0202602, 2018YFC0910405, 2017YFC1311003, 2016YFC1302500, 2016YFB0200400, 2017YFB0202104; The Funds of Peng Cheng Lab, State Key Laboratory of Chemo/Biosensing and Chemometrics; the Fundamental Research Funds for the Central Universities, and Guangdong Provincial Department of Science and Technology under grant No. 2016B090918122.

Bibliography

[1] Gaikwad, A., Toke, I.M.: Parallel iterative linear solvers on gpu: a financial engineering case. In: 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. pp. 607–614. IEEE (2010)

[2] Kim, J., Rajkumar, R., Kato, S.: Towards adaptive gpu resource management for embedded real-time systems. ACM SIGBED Review **10**(1), 14–17 (2013)

[3] Di Biagio, A., Barenghi, A., Agosta, G., Pelosi, G.: Design of a parallel aes for graphics hardware using the cuda framework. In: 2009 IEEE international symposium on parallel & distributed processing. pp. 1–8. IEEE (2009)

[4] Vasiliadis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: Pixelvault: Using gpus for securing cryptographic operations. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1131–1142 (2014)

[5] Nishikawa, N., Iwai, K., Kurokawa, T.: High-performance symmetric block ciphers on cuda. In: 2011 Second International Conference on Networking and Computing. pp. 221–227. IEEE (2011)

[6] Shi, L., Chen, H., Sun, J., Li, K.: vcuda: GPU-accelerated high-performance computing in virtual machines. IEEE Transactions on computers **61**(6), 804–816 (2011)

[7] Le, Y., Wang, Z.J., Quan, Z., He, J., Yao, B.: Acv-tree: A new method for sentence similarity modeling. In: IJCAI. pp. 4137–4143 (2018)

[8] Di, B., Sun, J., Chen, H.: A study of overflow vulnerabilities on gpus. In: IFIP International Conference on Network and Parallel Computing. pp. 103–115. Springer (2016)

[9] Di, B., Sun, J., Li, D., Chen, H., Quan, Z.: GMOD: a dynamic gpu memory overflow detector. In: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques. pp. 1–13 (2018)

[10] Nvidia: CUDA-MEMCHECK, https://developer.nvidia.com/cuda-memcheck. Last accessed 26, Aug, 2020

[11] Huang, X., Rodrigues, C.I., Jones, S., Buck, I., Hwu, W.m.: Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In: 2010 10th IEEE International Conference on Computer and Information Technology. pp. 1134–1139. IEEE (2010)

[12] Widmer, S., Wodniok, D., Weber, N., Goesele, M.: Fast dynamic memory allocator for massively parallel architectures. In: Proceedings of the 6th workshop on general purpose processor using graphics processing units. pp. 120–126 (2013)

[13] Huang, X., Rodrigues, C.I., Jones, S., Buck, I., Hwu, W.m.: Scalable simd-parallel memory allocation for many-core machines. The Journal of Supercomputing **64**(3), 1008–1020 (2013)

[14] Steinberger, M., Kenzel, M., Kainz, B., Schmalstieg, D.: Scatteralloc: Massively parallel dynamic memory allocation for the gpu. In: 2012 Innovative Parallel Computing (InPar). pp. 1–10. IEEE (2012)

[15] Unified Memory, http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf. Last accessed 26, Aug, 2020
[16] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX security symposium. vol. 98, pp. 63–78. San Antonio, TX (1998)
[17] Zeng, Q., Wu, D., Liu, P.: Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures. ACM SIGPLAN Notices **46**(6), 367–377 (2011)
[18] Zhang, J., Donofrio, D., Shalf, J., Kandemir, M.T., Jung, M.: Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures. In: 2015 International Conference on Parallel Architecture and Compilation (PACT). pp. 13–24. IEEE (2015)
[19] Manca, E., Manconi, A., Orro, A., Armano, G., Milanesi, L.: Cuda-quicksort: an improved gpu-based implementation of quicksort. Concurrency and computation: practice and experience **28**(1), 21–43 (2016)