Received: 13 February 2020 | Revised: 1 June 2020 | Accepted: 13 August 2020
DOI: 10.1002/cpe.6018

SPECIAL ISSUE PAPER

Comparing unified, pinned, and host/device memory allocations for memory-intensive workloads on Tegra SoC

Jake Choi[1] | Hojun You[2] | Chongam Kim[2] | Heon Young Yeom[1] | Yoonhee Kim[3]

[1]Department of Computer Science, Seoul National University, Seoul, South Korea

[2]Department of Mechanical and Aerospace Engineering, Seoul National University, Seoul, South Korea

[3]Department of Computer Science, Sookmyung Woman's University, Seoul, South Korea

Correspondence
Yoonhee Kim, Department of Computer Science, Sookmyung Woman's University, Seoul, South Korea.
Email: yulan@sookmyung.ac.kr

Funding information
Ministry of Science and ICT, Korea, Grant/Award Numbers: 2015M3C4A7065646, 2016M3C4A7952587, 2020R1H1A2011685

Summary

Edge computing focuses on processing near the source of the data. Edge computing devices using the Tegra SoC architecture provide a physically distinct GPU memory architecture. In order to take advantage of this architecture, different modes of memory allocation need to be considered. Different GPU memory allocation techniques yield different results in memory usage and execution times of identical applications on Tegra devices. In this article, we implement several GPU application benchmarks, including our custom CFD code with unified, pinned, and normal host/device memory allocation modes. We evaluate and compare the memory usage and execution time of such workloads on edge computing Tegra system-on-chips (SoC) equipped with integrated GPUs using a shared memory architecture, and non-SoC machines with discrete GPUs equipped with distinct VRAM. We discover that utilizing normal memory allocation methods on SoCs actually use double the required memory because of unnecessary device memory copies, despite being physically shared with host memory. We show that GPU application memory usage can be reduced up to 50%, and that even performance improvements can occur just by replacing normal memory allocation and memory copy methods with managed unified memory or pinned memory allocation.

KEYWORDS
benchmark, CFD, CUDA, GPU, memory, pinned, Rodinia, unified

1 | INTRODUCTION

Processing is being pushed toward the edge with the offloading of computations and storage nearer to the source of data in order to improve latency and save bandwidth.[1] The Tegra[2] system-on-chip (SoC) is a lightweight, low power, small form factor embedded module equipped with a GPU, CPU, and memory making it an effective platform choice for edge computing.[3] Unlike typical discrete GPUs, the integrated GPU used in the Tegra SoC shares the same physical memory with the CPU ARM cores, visualized in Figure 1. This architectural difference from normal hardware setups entails changes in the way memory is managed in CUDA programs.

Three principal components are usually required in high performance computing (HPC) or cloud cluster configurations: host machines, discrete GPU devices, and the interconnect.[5] A typical discrete GPU is enclosed in a board with separate physical memory, or VRAM. Such GPUs generally cannot function independently without a host machine to perform control operations, for example, memory management. Host machine memory is physically distinct from GPU device memory. As a rule of thumb, host memory should generally exceed GPU device memory capacity in order to properly utilize the GPU device because of the necessity to perform data copies to and from GPU device memory. Therefore, in most HPC or cloud hardware setups, the programmer is responsible for maintaining consistency between host and GPU memory. This task may prove to be cumbersome, and in order to not deal with such explicit memory management nuisances, NVIDIA unified memory (UM),[6] explained in Section 2, can be used to unify the logical view of GPU device and host memory. Pinned memory (a.k.a zero-copy memory) is also another CUDA memory management technique used to avoid explicit memory copies from host memory to GPU device memory.

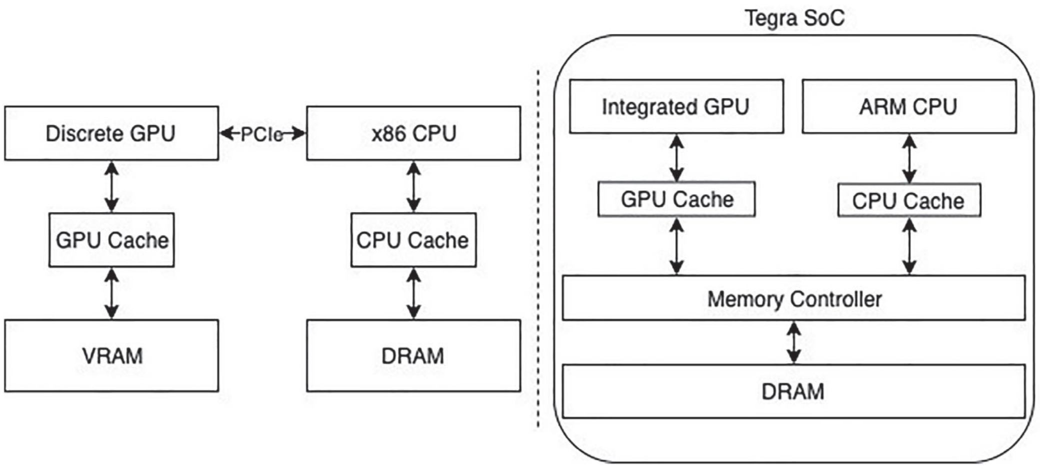

FIGURE 1 Simplified discrete GPU memory model (left) vs integrated GPU memory model (right) based on information from Reference 4

Because Tegra-based devices share the same physical GPU and CPU memory, the behavior of applications using the various types of memory management techniques on these devices will differ. Our goal is to investigate, compare, and evaluate the differences in performance and memory usage of using different types of CUDA memory allocation techniques on both Tegra-based SoC integrated GPUs and typical host-connected discrete GPUs.

Our article makes the following contributions:

- We allocate host and GPU memory on standard and custom application benchmarks using Unified memory, pinned memory, and explicit or *normal* memory.
- We evaluate the execution time, and memory usage of such applications including our custom memory-intensive CFD code on integrated GPUs (iGPUs) when compared to discrete GPUs (dGPUs).
- Throughout our evaluation, we discuss the difference in behavior of the CUDA runtime on the different memory allocation APIs when iGPUs are used instead of dGPUs.

The rest of this article is structured as follows. We first provide a brief background on UM and pinned memory in Section 2 and describe related work in Section 3. Subsequently, we describe the implementation of our techniques on each respective application in Section 4. Then, we describe our experimental setup and show evaluation results for each application in Section 5, and finally conclude with directions for future work in Section 6.

2 | BACKGROUND

NVIDIA unified memory is a unified address space that is accessible from either GPU or CPU[7] using a single shared pointer. It was introduced in CUDA 6 and it builds upon Unified Virtual Addressing (UVM) introduced in CUDA 4,[8] which allows a single virtual address space accessible from any device. Using UM eliminates the need for explicit memory copies from host memory to GPU device memory. When UM is accessed, the CUDA runtime will automatically perform page (usually 4KB) migration on-demand to the memory of the accessing processor[9] depending on if the page is dirty. Pinned or **zero-copy** memory is a older technology introduced in CUDA 2 which also allows GPU access without memory copies. Memory is allocated in host memory using `cudaHostAlloc` and is accessed from the GPU by obtaining a pointer through `cudaHostGetDevicePointer`. After UVA was introduced, the additional step of obtaining pointers by calling `cudaHostGetDevicePointer` becomes redundant, and allocating the data in host memory is sufficient in order to utilize pinned memory. The intricate details of the exact operations that the CUDA API performs is not available to the public. Ultimately, the goal of this article is to show that `cudaMalloc` should not be the default go-to method for GPU memory allocation when GPUs are used, and also to investigate the differences in memory usage, execution time, and behavior of each of the different types of memory allocation operations on both Tegra and non-Tegra devices.

3 | RELATED WORK

There are some existing evaluations on the performance of UM on Tegra devices. Most of the literature provide details on speed performance instead of the actual memory usage of applications, which is what our work focuses on.

Li et al[10] published work that evaluates UM performance on NVIDIA Kepler K40 and Jetson TK1 GPUs, focusing on the performance loss that results after testing modified applications from benchmarks like Parboil Benchmark Suite,[11] and Diffusion3D Benchmark.[10] The authors profiled the results using NVIDIA Visual Profiler[12] in order to generate a timeline. They state that application performance loss on the Tegra TK1 results because of additional redundant memory transfers that occur within the shared DRAM *despite* UM being used. They also do not mention any details about the difference in memory usage between discrete and integrated GPUs, which is the primary focus of this article.

Jarząbek and Czarnul[13] evaluate the performance of UM and dynamic parallelism (DP) on NVIDIA GTX 970 and Tesla K20m boards. Dynamic parallelism is a feature introduced in CUDA 5[8] which allows recursive launches of kernels within kernels. Contrary to the purpose of our work, the authors do not use Tegra-based architecture for their evaluation, and also focus primarily on the speed performance effects of using both UM and DP for specific parallel applications like verification of Goldbach's conjecture, adaptive integration using the trapezoidal rule, and heat transfer simulation in 2D space. They state that using UM in their tested applications results in negligible performance loss which compensates for the ease of programming using managed memory.

Cavicchioli et al[14] highlight the memory interference issues on two Tegra SoCs (K1 and X1) caused by performing different types of iGPU memory operations using manual GPU copy kernels on device buffers with or without involving UVM or invoking DMA copy engines using pinned memory with `cudaMemset`or `cudaMemcpy` while the host CPU is actively accessing memory. Their work focuses on the worst case execution times that result due to latency spikes caused by the memory interference from either CPU or iGPU. This work is orthogonal to ours because we focus primarily on memory usage instead of latency issues.

Finally, Ukidave et al[15] evaluate UM performance on Jetson TK1 boards by co-executing pairs of applications from the Rodinia benchmark on the Tegra iGPU and externally connected Tesla K40 dGPU. Experiments are performed to see how much execution time along with energy and power consumption is improved, when compared to executing these pairs of applications on the dGPU alone. Again this work does not focus on memory usage, and does not expose much of the details of the CUDA runtime behavior.

In addition to these works, there are also several research papers published involving GPU and ARM processors as well as the GPU-CUDA environment, like Montella et al,[16,17] D'Amore et al,[18] and Laccetti et al,[19] which cover other facets of GPU and ARM core processing.

4 | RODINIA BENCHMARK IMPLEMENTATIONS

We target several Rodinia[20] benchmarks and our custom CFD code because they use relatively significant amounts of GPU memory. We replace normal `cudaMalloc` and `cudaMemcpy` calls in the source code of these applications with `cudaMallocManaged` or `cudaHostAlloc` in order to utilize Unified or pinned memory. When using pinned memory, we give the flag `cudaHostAllocDefault` to ensure that the data is page-locked on the CPU side. Table 1 shows the minor API differences that have to be implemented in the source code to utilize different memory allocation mechanisms. Most of the available GPU application source codes do not utilize managed memory. We plan to automatize the process of converting from normal `cudaMemcpy` operations to use managed memory without having to manually change source code either by parsing or changing the behavior of the compiler or CUDA runtime as future work. We measure the memory usage by using `cudaMemGetInfo` calls to get total and free memory on the GPU. On the Jetson TX2 SoC, this command retrieves the system memory because the iGPU does not have its own VRAM. We measure CPU memory usage on the host attached to a dGPU by measuring from `free` and `nvidia-smi` as the applications run. Unlike the CUDA API memory allocation functions, simply calling `malloc` does not report used memory until the memory space is actually filled with data.

4.1 | Vector addition

We implement simple CUDA vector addition using Unified, pinned, and normal memory allocation on both iGPUs and dGPUs. We allocate two arrays of identical types (float, int, and double) with 2^{28}elements each (except for type double which uses 2^{27} elements because of lack of sufficient memory

TABLE 1 Syntactical differences of unified, pinned, and normal memory

Unified memory example	Pinned memory example	Normal memory example
`float *x;`	`float *x;`	`float *x, *d_x;`
`cudaMallocManaged(&x, SIZE);`	`cudaHostAlloc(&x, SIZE, FLAGS);`	`x = malloc(SIZE);`
assign values to `x`	assign values to `x`	`cudaMalloc(&x, SIZE);`
call kernel with `x`	call kernel with `x`	assign values to `x`
		`cudaMemcpy(d_x, x, SIZE)`
		call kernel with `d_x`

capacity) in order to use a noticeable amount of memory. We initialize the arrays with constant values on the host, and run kernel addition on the GPU. Then we check for errors after the kernel is finished to make sure the values are the real sum. We check free memory on the Jetson TX2 using `cudaMemGetInfo` before and after memory allocations are made.

4.2 | B+ tree

The B+ tree application from the Rodinia benchmark receives an input of integers from a text file and generates a B+ tree capable of being stored on GPU memory. It then receives a set of command inputs to execute on the tree nodes. These include range and point searches. The entire tree is allocated in a single, contiguous block of memory using `malloc` to allocate the maximum number of nodes initially, followed by an assignment of the record and node pointers to point to two different regions of this single block of memory. We modify this application by replacing the `malloc` call with CUDA managed memory API calls, where we allocate separate regions of memory for the record and node pointer by calling an allocation wrapper function in the CUDA wrapper code. We then remove the `cudaMemcpy`calls in that file for that particular data structure, and directly use the data structures in both GPU kernels for the forward and backward phase. Therefore, a major but not entire portion of the application data is allocated using either unified or pinned memory. Other minor data structures remain untouched and use ordinary host `malloc` in all versions of the code.

4.3 | Backpropagation

Backpropagation is a two-phase application that trains the weights of connected nodes on a layered neural network. In the forward phase, the activations are propagated from the input to the output layer. In the backward phase, the error between the observed and requested values in the output layer is propagated backwards to adjust the weights and bias values.[20] We replace all of the internal allocation of the neural network data structures to use either pinned or unified memory, by calling our own allocation function in the CUDA source code. Then we remove CUDA memory copy operations and replace the kernel device variables with the host variables.

In addition to this, like the example given in the left column of Table 2, the host portion of the code allocates several of the weight data structures to use two-dimensional C-style arrays of fixed length (non-jagged). Although technically possible, CUDA is not very supportive of allocating such multi-dimensional arrays when using managed memory, and we must manually transform these data structures to one-dimensional arrays. Because the code was written when there was no support of managed memory, there is a redundant portion in the code which copies these multi-dimensional arrays to a one-dimensional array for use in the GPU kernels. As a result, we modify the application code to instead use one-dimensional arrays in the host code from the beginning, and omit these redundant memory transformations. This shows that simply replacing multi-dimensional `malloc` calls with managed memory allocation will be inefficient, not to mention cumbersome because of the syntactical differences of the allocation API dealing with pointer addresses (like the right column in Table 2). Because GPU allocation may not be performed lazily, explained more in Section 5.2, repetitive calls would cause too much unnecessary overhead, and it was better performance-wise to simply allocate contiguous one-dimensional arrays using managed memory from the beginning.

4.4 | BFS

BFS traverses all the connected components in a graph that is read from a file, and outputs the cost into a result file. The graph file can be generated by an input generator, and we used it to create two random graphs with 8 and 32 million points. The application was relatively straightforward to convert, and we simply replaced `malloc` calls with managed memory API, and removed all subsequent `cudaMalloc` and `cudaMemcpy` calls.

4.5 | Custom CFD code

Our custom CFD code focuses on high-order methods that break through the limitations of conventional second-order finite volume methods (FVM), due to the rising demand for analyzing more complex aerodynamic applications in the fields of mechanical and

TABLE 2 Multi-dimensional array allocation with managed memory

Using normal memory	Using managed memory
`x = malloc(N*sizeof(TYPE *));`	`cudaMallocManaged(&x, N*sizeof(TYPE *));`
`for( i to N )`	`for( i to N )`
`x[i] = malloc(M*sizeof(TYPE));`	`cudaMallocManaged(&x[i], N*sizeof(TYPE));`

aerospace engineering. The high-order methods possess many attractive features: the capability to achieve arbitrary high accuracy with compact stencils, high spectral resolvability fitted to turbulence simulation, and high scalability under large parallel computing systems.

The three-dimensional compressible Navier-Stokes equations given by

$$\frac{\partial \mathbf{Q}}{\partial t} + \nabla \cdot \mathbf{F}_c(\mathbf{Q}) = \nabla \cdot \mathbf{F}_v(\mathbf{Q}, \nabla \mathbf{Q}), \tag{1}$$

where $\mathbf{Q}$ are conservative variables and $\mathbf{F}_c(\mathbf{Q}), \mathbf{F}_v(\mathbf{Q}, \nabla \mathbf{Q})$ are convective and viscous fluxes, respectively, are considered in our custom CFD application. Among the various high-order methods, we focus on the discontinuous Galerkin method[21] that is the most widely used in CFD society because of its intuitive form and rigorous mathematical backgrounds. Applying the discontinuous Galerkin method into Equation (1), we finally get the following weak formulation on each element Ω:

$$\begin{aligned}&\int_{\Omega} \frac{\partial \mathbf{Q}}{\partial t} \varphi dV + \int_{\partial\Omega} (\widehat{\mathbf{F}_c \cdot \mathbf{n}} - \widehat{\mathbf{F}_v \cdot \mathbf{n}}) \varphi dA = \\ &\int_{\Omega} \nabla \varphi \cdot (\mathbf{F}_c - \mathbf{F}_v) dV,\end{aligned} \tag{2}$$

where φ is the orthogonal basis computed from the modified Gram-Schmidt process.[22] Here, $\widehat{\mathbf{F}_c \cdot \mathbf{n}}$ is a monotone numerical convective flux used in FVM, and $\widehat{\mathbf{F}_v \cdot \mathbf{n}}$ is a numerical viscous flux computed via BR2 method.[23]

Our GPU implementation of this custom CFD code has three versions, one using NVIDIA UM, the other pinned memory, and the last using manual CUDA memory allocation and copy operations. Workload size is limited to GPU device capacity when ordinary `cudaMalloc` is used, but using UM allows the workload size to reach host memory limits, which is more important for dGPUs. Table 3 shows the size of major data arrays along with their access type in our GPU kernels. Total memory usage of the host code is not limited to only these data structures, with the existence of many host-allocated structures that remain in pure CPU code. However, the structures shown in Table 3 are the ones used heavily in GPU calculation, and so we allocate these data structures using either Unified, pinned, or normal memory.

In our previous work,[24] we utilized ZFP compression[25] to compress data in the pre-processing stages of the CFD code to save dGPU memory space by up to 50% without loss of accuracy. However, on iGPU-based SoCs, it is ineffective to use compression solely on the GPU side as an alternative solution to Unified memory because host and GPU memory is shared. Therefore we have to perform compression on data allocated with Unified memory in unison as a solution to memory capacity limitations. Doing this requires modification on the host CPU code to utilize the compressed data at all times. We leave this to future work.

TABLE 3 List of major data arrays referenced in GPU kernels and major kernels for the custom CFD code

		% of data	Kernel		% of
Array name	RW	Usage	Name	Avg. time	time
cell_coefficients	R	50.81%	fourth_loop_3	9.74ms	38.18%
cell_basis_value	R	16.94%	fourth_loop_1	5.93ms	23.24%
face_owner_basis_value	R	5.77%	third_loop_1	3.95ms	15.47%
face_neighbor_basis_value	R	5.77%	third_loop_3	2.81ms	11.01%
face_owner_coefficients	R	5.77%	first_loop_1	1.22ms	4.79%
face_neighbor_coefficients	R	5.77%	first_loop_3	583.88μs	2.29%
peribdry_owner_basis_value	R	1.15%	fourth_loop_2	490.32μs	1.92%
peribdry_neighbor_basis_value	R	1.15%	third_loop_2	334.34μs	1.31%
peribdry_owner_coefficients	R	1.15%	memcpy HtoD	7.38μs	1.10%
Flux	RW	3.02%	first_loop_2	125.88μs	0.49%
solution	RW	0.17%	memcpy DtoD	15.39μs	0.05%
rhs	W	0.17%			

5 | BENCHMARK EVALUATIONS AND EXPERIMENTAL SETUP

We evaluate the Rodinia and custom CFD benchmarks on two machines. One is a GeForce GTX 1050 Ti dGPU using the Pascal architecture with 4GB of VRAM. The host CPU is an Intel i7-7700 @ 3.6GHz with 4 physical cores. We use CUDA version 10 in this setup with Linux kernel version 4.15.0. The second setup is a Jetson TX2[26] equipped with 6 cores and a Pascal architecture iGPU with 8G of VRAM. The Linux kernel version is 4.4.38-tegra, and we use CUDA version 9. In our evaluation, we refer to the Jetson TX2 with the iGPU as SoC, and the private machine equipped with the dGPU as non-SoC.

One caveat that we encountered in our SoC when conducting memory usage experiments is that even though the tested application is completed, there is a constant amount of pre-allocated memory (about ~900 MB) that lingers in the system as "used" memory which is reserved for any subsequent usages of all of the CUDA memory allocation functions. Therefore, in order to completely free all of the pre-allocated memory for accurate measurement purposes, we artificially triggered a segmentation fault error using normal host `malloc` calls followed by initialization of arbitrary data after every run because that allowed us to free up the GPU memory pre-allocations on the SoC device. In the case of host memory allocations unrelated to the GPU (i.e., triggering memory usage from `malloc`), there was no such issue.

5.1 | Vector addition

The experimental results for vector addition are shown in Figure 2. In the non-SoC case, allocations with pinned memory use almost no dGPU memory (except for a minor amount resulting to about ~50 MB allocated on the dGPU device memory, which is not included in the graph results). We need to drop the caches in order to not allocate existing buffer cache memory when measuring pinned memory. Execution time of vector addition for the double type is fastest for both SoC and non-SoC, which is natural because the total number of elements calculated is half the other data types, hence having the same memory usage. The results are more interesting for SoC devices, because we can see that performing normal CUDA memory allocations with memory copies actually uses double the space needed, and has longer execution times. Another interesting note is that the application executed with pinned memory performed the fastest among the three. This is because the data access patterns exhibited spatial locality in the absence of temporal locality, which does not offset the initial costs of setting up the synchronization and caching behavior of unified memory.

5.2 | B+ tree

We run the experiments using 5,000,000 points from input. Results are shown in Figure 3. Initial tree generation is performed at the host and constitutes a significant portion of the application execution time (not included in the graph), and can take many minutes especially on the SoC. We notice different behaviors of the CUDA runtime when allocating managed memory on the SoC and non-SoC. The reason why the SoC allocates more memory even when using managed or pinned memory is because the entire memory is mapped as soon as `cudaMallocManaged` or `cudaHostAlloc` is called on the SoC leading to lots of unused space. However, for unified memory on the non-SoC, memory is mapped in a lazy manner; therefore memory usage shows only the actual number of used nodes. Memory usage for pinned memory on the non-SoC is comparable to normal and Unified even though it should be much less, only because it also maps the entire maximum memory space from the initial `cudaHostAlloc` call. For normal

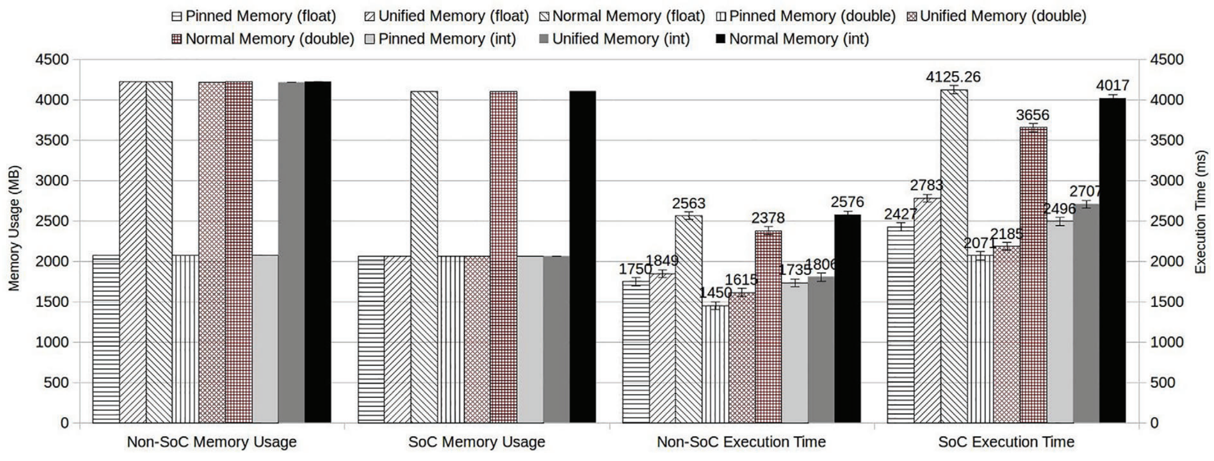

FIGURE 2 Vector addition memory usage and execution time

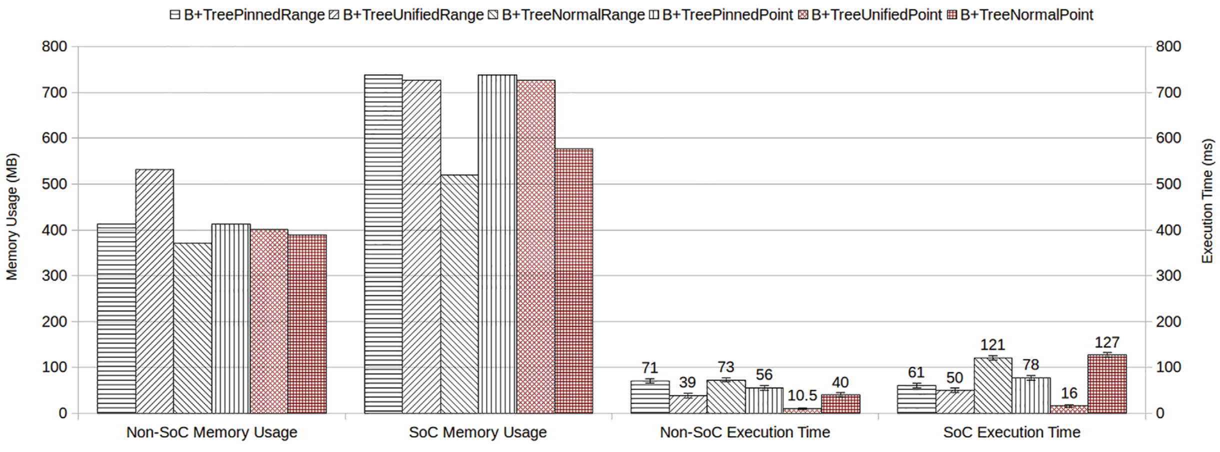

FIGURE 3 Rodinia B+ tree memory usage and execution time (range and point search))

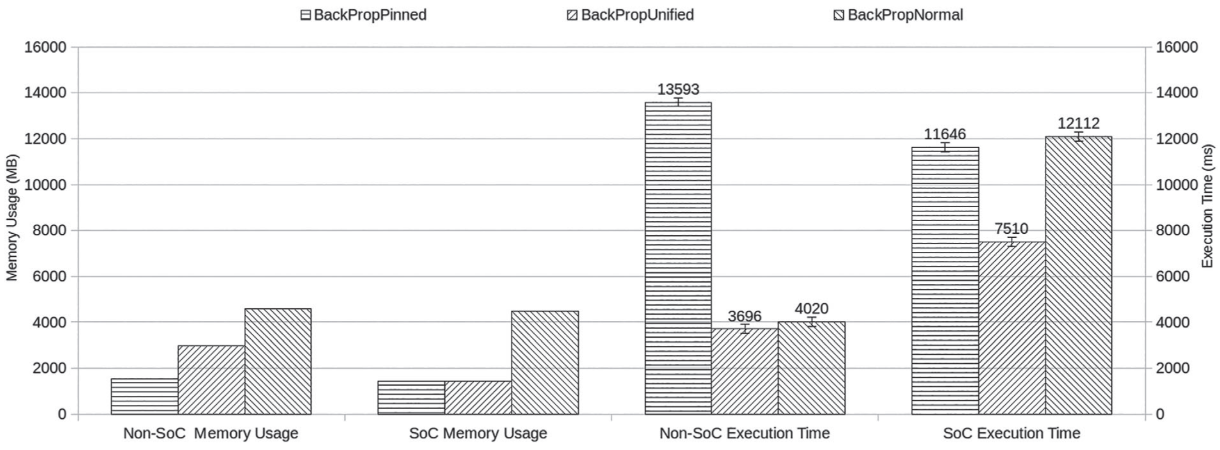

FIGURE 4 Rodinia backpropagation memory usage and execution time (n=10,000,000)

`malloc` calls, memory is always mapped in a lazy manner in the operating system level; therefore the results actually show slightly less memory usage compared to unified or pinned memory.

Execution time of the compute kernel is reduced when managed memory is used on the SoC because there is less memory copy overhead. Unified memory takes the least amount of execution time, especially more so on the point searches because GPU pages are fetched on-demand. It is interesting to note that pinned memory performs either worse or near to normal memory allocations with execution times of 71 and 56 ms for range and point searches on the non-SoC, but actually performs better than normal memory allocation on the SoC, with execution times of 61 and 78 ms. This shows that even though memory is host-allocated on the SoC, because of its proximity to the iGPU and lack of need for memory copy operations, it is actually better in terms of execution times to allocate memory using either pinned or Unified memory.

5.3 | Backpropagation

We test backpropagation using 10,000,000 random points as input. In the original version of the benchmark, the maximum input size is limited to 1,048,576 points because the points are divided in batches of 16 among the GPU blocks, which is limited 65,536 in one dimension of the GPU grid. Strangely, the y-axis of the grid was being utilized, and we changed it so that the x-axis would be used to allow for larger dimensions so that we could run the application with a greater number of points.

Results are shown in Figure 4. Unified memory uses less memory than normal allocations even in the non-SoC case because we had to flatten the 2-dimensional arrays into one-dimension in the host level. Using pinned or Unified memory for the SoC case utilizes less than half of the system memory usage of the original code. Performance-wise, unified memory performs the fastest on both environments because of

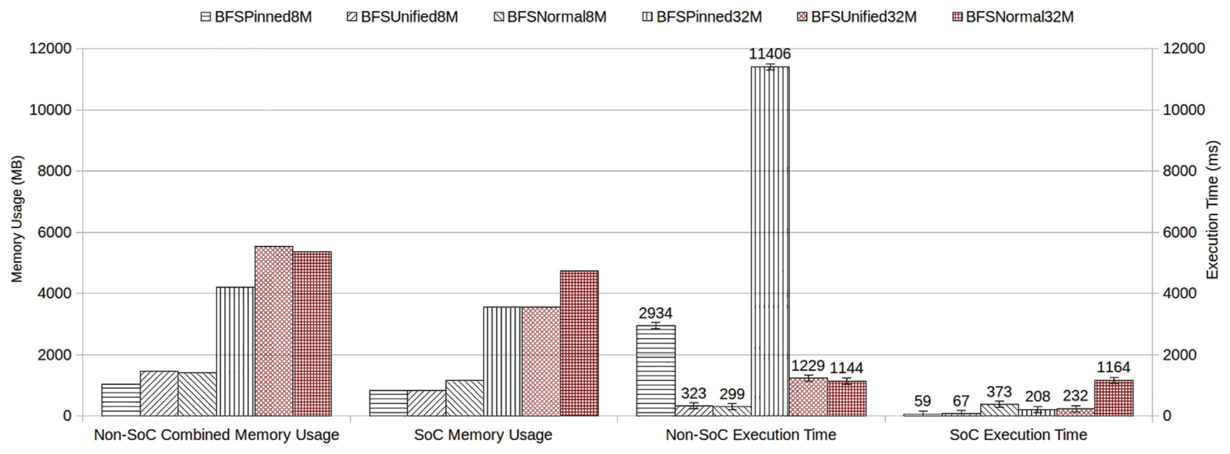

FIGURE 5 Rodinia BFS memory usage and execution time (8M and 32M points)

the lack of memory copy, and also because of temporal locality of the workload, with a lot of repetitive accesses allowing for the caching effect to take place. Pinned memory performs the worst in the non-SoC execution, but actually performs better than normal memory in the SoC because of the lack of memory copy overhead, despite no cache support. Unified memory seems to be the best choice for this particular application.

5.4 | BFS

In order to get the proper memory usage results for this workload on the non-SoC, we had to drop the buffer caches before each run, because all host memory allocations used memory from the already pre-allocated buffer cache pool, which could have led to incorrect measurements. Experimental results are shown in Figure 5. Unified memory uses slightly more memory than normal memory allocations in the non-SoC, but the difference is negligible enough to be considered as slight measurement errors due to factors like the buffer cache and other application interference. Pinned memory uses the least amount of memory as usual, but took the longest execution time in the non-SoC. Surprisingly, pinned memory exhibited the best execution time for the SoC. This is most likely attributed to the spatial locality of the workload, and the fact that each thread tries to not visit the previously visited point in BFS. In addition to this, the data are also located much closer to the GPU compared to in the non-SoC case. For the BFS application, pinned memory is optimal on SoC devices, with normal host/device allocations being the choice for the non-SoC.

5.5 | Custom CFD code

We use two workloads to test our custom CFD code, one which is relatively small and can easily fit into the memory of both GPUs, and a larger one that pushes the memory limits of a single physical GPU. Both workloads use the same flattened data arrays and kernels outlined in Table 3, but with differing sizes of the data arrays. Results are shown in Figure 6. Execution time is shown in seconds, instead of milliseconds. The large workload on the SoC overflowed system memory (about 8GB), and therefore was drawn with an error bar close that extends beyond the non-SoC result. Execution time was also derived from the smaller workload results.

Unified memory in the non-SoC case takes up less memory than normal memory allocations, most likely due to the migration of data pages as the calculation stages are being performed. As expected, both pinned and unified memory in the SoC case take up the same amount of total system memory, with normal memory allocations overflowing beyond system memory in the large workload. The results are not as drastic in the smaller workload because there is a lot of CPU-side prerequisite data which is not used by the GPU kernels at all.

Even though pinned memory seems to use the least amount of total memory on both systems, execution time is dismal as the GPU kernels must continually fetch data from host memory as calculation takes place. However, the relative difference between pinned and other allocation modes is not as distinct on the SoC because of the proximity of the data to the iGPU engine. Regardless, unified memory seems to be the best option for running the workload with minor differences in execution time compared to the normal memory allocation mode.

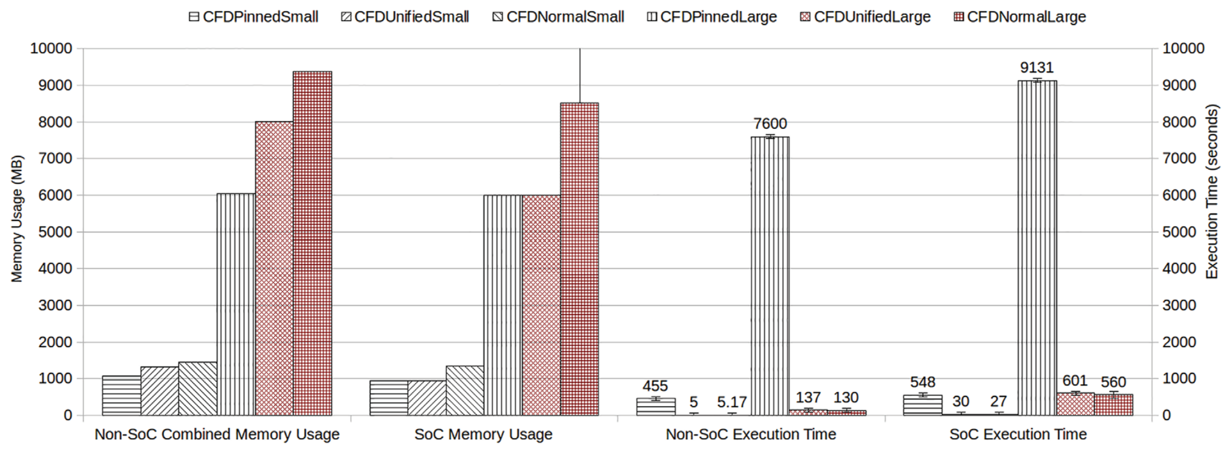

FIGURE 6 Custom CFD memory usage and execution time (5x5x5 and 10x10x10 workloads)

6 | CONCLUSION AND FUTURE WORK

Edge computing focuses on processing near the source of the data. Devices that are used with this concept, like the Tegra architecture, have physically distinct memory architectures from average server processing nodes, especially when GPU usage is factored in. In order to take advantage of these architectures, other modalities of memory allocation need to be considered. Different GPU memory allocation techniques yield different results in memory usage and execution times of identical applications on these types of edge computing related SoC devices. In this article, we focused on the differences in behavior on both common and custom GPU application benchmarks when SoC iGPUs are used in comparison to non-SoC dGPUs.

There is not a single cookie-cutter answer to determine the correct memory allocation method to be used in each application. Execution time, and even total memory usage may depend on the circumstances of the application, and how it is originally programmed. Some applications, like the B+ tree benchmark (Section 5.2), may actually use even more memory on both SoC and non-SoC devices when managed or pinned memory is used, due to the differences in how memory allocation is actually performed on the CUDA runtime from API calls. Other applications that have good spatial but poor temporal locality (Section 5.1) would benefit the most from using pinned memory because of the lack of need for setting up caching functionality. Nevertheless, for many GPU applications which do benefit from temporal locality, using unified memory provides the greatest efficiency in terms of memory usage compared to execution time.

In future work, we will explore methods in which we can effectively convert between different GPU memory allocation modes without source code modification. We will also try to explore how we can effectively reduce memory usage for our custom CFD code using compression without incurring too much performance overheads. We also wish to discover a more definitive way of classifying the different applications in order to determine which type of memory allocation method they should use.

ACKNOWLEDGEMENTS

This research was supported by National Research Foundation of Korea (NRF) grant funded by the Korean Government (MSIT) (No.2015M3C4A7065646, No.2016M3C4A7952587, and No.2020R1H1A2011685).

ORCID

Yoonhee Kim https://orcid.org/0000-0003-4799-3209

REFERENCES

1. What is edge computing? - enterprise it definitions. Enterprise IT Definitions; HPE EUROPE. https://www.hpe.com/emea_europe/en/what-is/edge-computing.html. Accessed January 20, 2020.
2. NVIDIA Embedded Systems for Next-Gen Autonomous Machines. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/. Accessed January 20, 2020.
3. Mittal Sparsh A. Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform. *J Syst Architect*. 2019;97:428-442. https://doi.org/10.1016/j.sysarc.2019.01.011 http://www.sciencedirect.com/science/article/pii/S1383762118306404.
4. Amert T, Otterness N, Yang M, Anderson JH, Smith FD. GPU scheduling on the NVIDIA TX2: hidden details revealed. Paper presented at: 2017 IEEE Real-Time Systems Symposium (RTSS). Paris, France; 2017:104-115.
5. Kindratenko VV, Enos JJ, Shi G, et al. GPU clusters for high-performance computing. Paper presented at: 2009 IEEE International Conference on Cluster Computing and Workshops. New Orleans, Louisiana, USA; 2009:1-8.
6. Li W, Jin G, Cui X, See S. An evaluation of unified memory technology on NVIDIA GPUs. Paper presented at: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Shenzhen; 2015:1092-1098.

7. Harris M. Unified Memory for CUDA Beginners; NVIDIA Developer Blog. NVIDIA Developer Blog. https://devblogs.nvidia.com/unified-memory-cuda-beginners/. Accessed January 21, 2020.
8. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#unified-virtual-address-space. Accessed January 21, 2020.
9. Negrut D, Serban R, Li A, Seidell A. Unified Memory in CUDA 6: A Brief Overview. https://www.drdobbs.com/parallel/unified-memory-in-cuda-6-a-brief-overvie/240169095?pgno=2. Accessed January 21, 2020.
10. Li W, Jin G, Cui X, See S. An evaluation of unified memory technology on NVIDIA GPUs. Paper presented at: Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGRID'15; IEEE Press; Piscataway, NJ, USA. 2015:1092–1098. https://doi.org/10.1109/CCGrid.2015.105.
11. Stratton JA, Rodrigues C, Sung IJ, et al. Parboil: a revised benchmark suite for scientific and commercial throughput. *Comput Secur*. 2012;127.
12. Nvidia Corp: Profiler user's guide. An optional note; 2017. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview. Accessed January 22, 2020.
13. Jarząbek Ł, Czarnul P. Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. *J Supercomput*. 2017;73(12):5378-5401. https://doi.org/10.1007/s11227-017-2091-x.
14. Cavicchioli R, Capodieci N, Bertogna M. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. Paper presented at: 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). Limassol, Cyprus; 2017:1-10.
15. Ukidave Y, Kaeli D, Gupta U, Keville K. Performance of the NVIDIA Jetson TK1 in HPC. Paper presented at: 2015 IEEE International Conference on Cluster Computing. Chicago, Illinois; 2015:533-534.
16. Montella R, Kosta S, Oro D, et al. Accelerating Linux and Android applications on low-power devices through remote GPGPU offloading. *Concurr Comput Pract Exper*. 2017;29:e4286. https://doi.org/10.1002/cpe.4286.
17. Montella R, Giunta G, Laccetti G. Virtualizing high-end GPGPUs on ARM clusters for the next generation of high performance cloud computing. *Cluster Comput*. 2014;17:139-152. https://doi.org/10.1007/s10586-013-0341-0.
18. D'Amore L, Laccetti G, Romano D, Scotti G, Murli A. Towards a parallel component in a GPU-CUDA environment: a case study with the L-BFGS Harwell routine. *Int J Comput Math*. 2015;92:59-76. https://doi.org/10.1080/00207160.2014.899589.
19. Laccetti G, Montella R, Palmieri C, Pelliccia V. The high performance Internet of Things: using GVirtuS to share high-end GPUs with ARM based cluster computing nodes. Paper presented at: International Conference on Parallel Processing and Applied Mathematics. Warsaw, Poland; 2014:734-744.
20. Che S, Boyer M, Meng J, et al. Rodinia: a benchmark suite for heterogeneous computing. Paper presented at: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), IISWC'09; IEEE Computer Society; Washington, DC, USA. 2009:44-54.
21. Cockburn B, Shu CW. The Runge-Kutta discontinuous Galerkin method for conservation laws V. *J Comput Phys*. 1998;141:199-224.
22. You H, Kim C. High-order multi-dimensional limiting strategy with subcell resolution I. Two-dimensional mixed meshes. *J Comput Phys*. 2018;375:1005-1032.
23. Bassi F, Crivellini A, Rebay S, Savini M. Discontinuous Galerkin solution of the Reynolds-averaged Navier-Stokes and k-ω turbulence model equations. *Comput Fluids*. 2005;34:507-540.
24. Choi J, Kim Y, Yeom HY. Overcoming GPU memory capacity limitations in hybrid MPI implementations of CFD. In: Raffaele M, Angelo C, Giancarlo F, Antonio G, Antonio L, eds. *Internet and Distributed Computing Systems*. Cham: Springer International Publishing; 2019:100-111.
25. Lindstrom P. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Trans Visual Comp Graph*. 2014;20:2674-2683.
26. Harness AI at the Edge with the Jetson TX2 Developer Kit. 2019. https://developer.nvidia.com/embedded/jetson-tx2-developer-kit. Accessed January 22, 2020.

How to cite this article: Choi J, You H, Kim C, Young Yeom H, Kim Y. Comparing unified, pinned, and host/device memory allocations for memory-intensive workloads on Tegra SoC. *Concurrency Computat Pract Exper*. 2020;e6018. https://doi.org/10.1002/cpe.6018