An Evaluation of CUDA Unified Memory Access on NVIDIA Tegra K1

John Joseph, Boston University Kurt Keville, MIT

Abstract

The UMA programming model was developed with the intent of simplifying memory management when developing applications with CUDA. We are interested in the change in performance one might observe when employing UMA on a Tegra K1 device, where program memory is physically unified, as opposed to a more traditional workstation where host and device memory reside in physically separate locations. We investigate the relationship between changes in performance and percentage of runtime spent in the kernel (as opposed to transferring data). Applications that spend more time copying data than they do in user-defined kernel operations will have the most to gain from using unified memory buffers, and we develop criteria by which one can determine if using UMA can benefit their application. We will also discuss the potential reduction in code complexity that accompanies code developed with unified buffers.

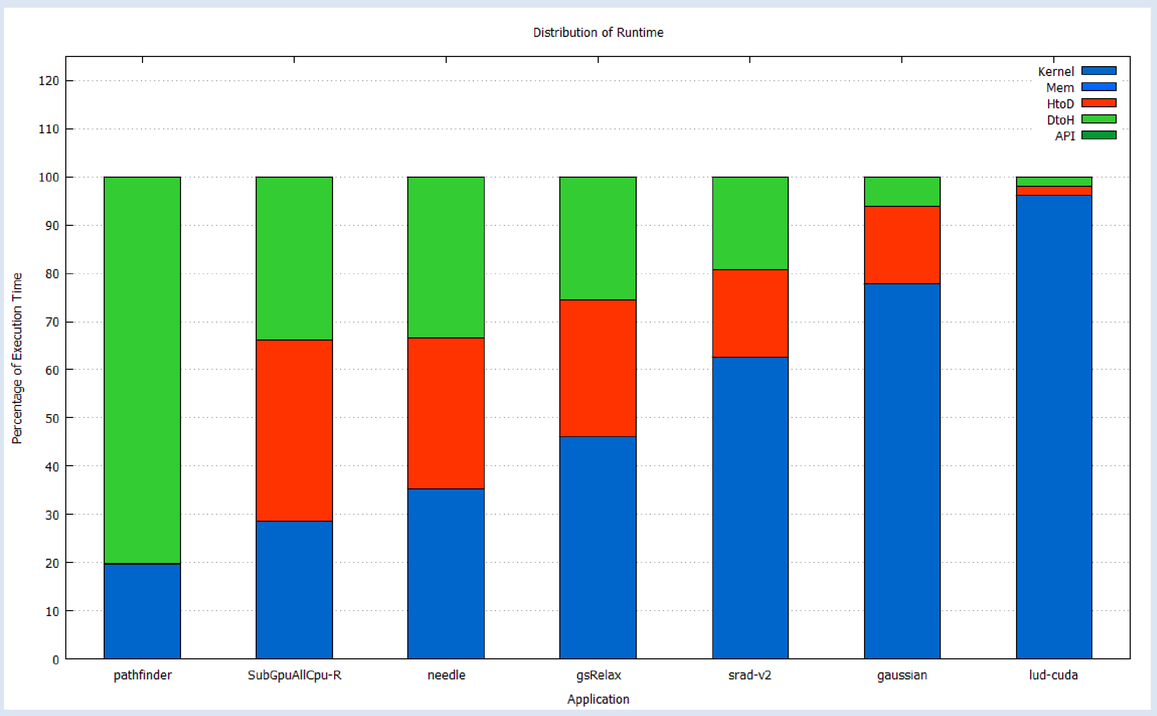
Materials and Methods

We run all tests on an Nvidia Jetson, a CUDA capable (SM 3.2) embedded computer with 192 cores at its disposal. Many of our benchmarks come from the Rodinia benchmark suite made available by the University of Virginia, and two tests were handwritten. The first handwritten test is a Gauss-Seidel relaxation to solve the 2-D Laplacian, the iterations of which set elements to the averages of their neighbors on the GPU before summing the square of all elements on the CPU. The second handwritten test, SubsetGPUAllCPU_Rand, selectively increments a random subset of the data buffer on the GPU before incrementing the entire buffer on the CPU. All Rodinia programs (save for mummergpu, not tested here) were profiled, and the ones we ported to use UMA were **pathfinder, needle, srad_v2, gaussian,** and **lud**.

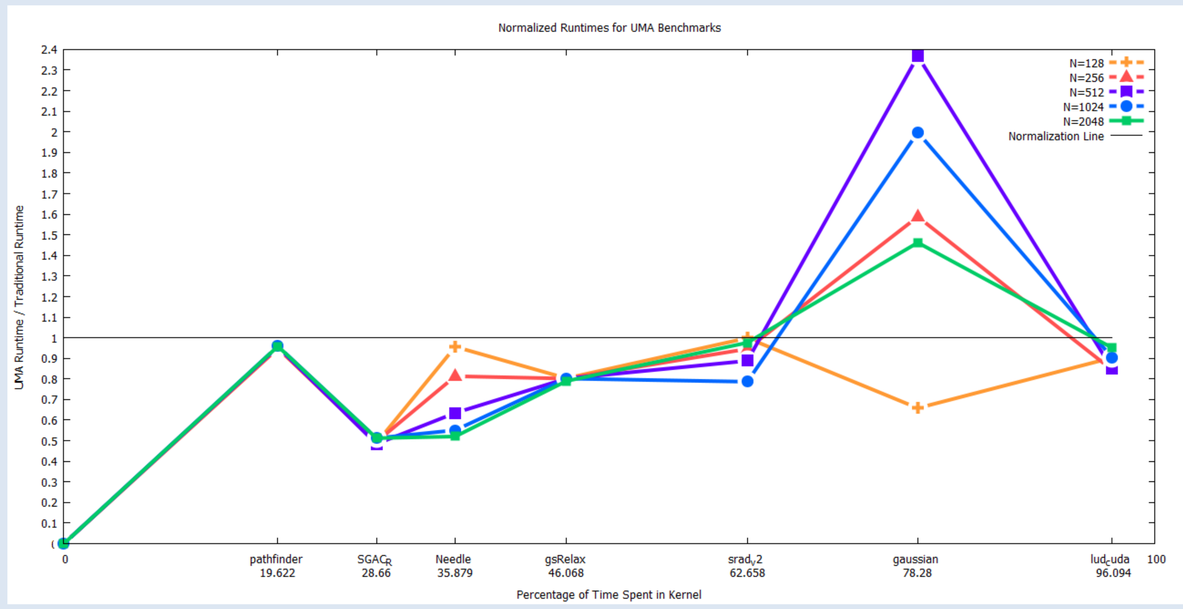
Henceforth, we will refer to programs that use separate host/device buffers as "**Traditional**", and programs that use unified memory as "**UMA**". Programs were first profiled in their traditional form using the nvprof tool in order to determine what percentage of their runtime was spent copying data and what percentage was spent in the kernel; we denote this percentage as **Pk.** We then ported the traditional programs with interesting **Pk** values to use unified memory buffers, noting the runtimes before and after our changes at several different problem sizes (denoted by the variable **N.**) The UMA runtime was divided by the traditional runtime in order to get a "**Normalized Runtime**", which represents the increase or decrease in runtime using UMA code. By plotting the normalized runtime against the time spent in the kernel, we hope to see what kind of program can benefit from using UMA. It should be noted that the word "problem size" generally refers to the number of elements in the problem; however, in every example but the AllGPUSubsetCPU_Rand example we deal with a matrix, and so the number of elements is actually N^2.

Results

Given that Tegra K1 devices physically unify host and device memory, our hypothesis was that programs that spend a large portion of their runtime copying data would stand to gain the most from using unified buffers. In the Runtime Distribution figure below we see the output of the nvprof tool arranged to show what percentage of runtime is spent by each program doing certain tasks. The tasks we were interested in are **kernel** (time spent in code written by the programmer), **DtoH** (Device to Host memcpy), **HtoD** (Host to Device memcpy), **Memset** (cudaMemset), and **API** (any other CUDA API calls). In general, runtime is dominated by kernel and DtoH or HtoD; it is the kernel percentage we are interested in.



From this information, we chose several programs in order to get a fair sample of kernel percentages ranging from 19.8% (pathfinder) to 96% (lud). We could not find anything lower than 19.8%, but decided to tie off the graph below at 0 for appearance's sake. Again, the list of programs chosen using this data were **gsRelax, SubsetGPUAllCPU_Rand, pathfinder, needle, srad_v2, gaussian,** and **lud**.



As you can see in the figure above, applications whose time spent in the kernel was roughly less than 60% demonstrated an improved runtime when using unified buffers. This improvement was seemingly irrespective of problem size, at least in the dimensions that we tried. It should be noted that for lower problem sizes (N < 128), UMA virtually always outperformed the Traditional code.

UMA Miscellany

- The code reduction that accompanies UMA is trivial. It does make things a bit cleaner, but our intuition is that the intended target market was new programmers. It is confusing to work with at first, and newer programs should learn firsthand where their data is going (and when) rather than abstract it away with UMA.
- UMA is not a silver bullet. When allocating a unified buffer, one invokes the **cudaMallocManaged** function, rather than **malloc** or **cudaMalloc**. However, for optimal performance we noticed that it was best to use **cudaMallocManaged** as sparingly as possible. In other words, if you know a buffer will only be used on the host, use **malloc**, and if you know a buffer will only live on the device, use **cudaMalloc**. Unified buffers are freed with **cudaFree**.
- Another "gotcha" is the need to call a synchronization routine, for instance **cudaDeviceSynchronize()** or **cudaThreadSynchronize()** before accessing unified data on the host after touching it on the device. It is necessary to block execution with these calls until CUDA is done with our buffers, or you will likely segfault.
- UMA works well with libraries like cuBLAS and Thrust, but when using Thrust be sure to set the **execution policy** to run on the device and/or use device pointers to unified buffers.

Conclusions

While the results do show an improvement for any program that uses roughly 60% or lower (srad_v2's kernel percentage was 62%), that improvement was difficult to quantify. Only in our two handwritten examples, which were rather contrived in their memory accesses, showed the most improvement. The mark of a good GPGPU program is to avoid data transfer as much as possible even in a shared memory system. However, moving data around is an unavoidable part of any problem, and in situations where data transfer consumes the majority of an application's execution time, using a unified buffer seems to be a fairly easy way of boosting performance. However, any programmer should be aware that there's more going on than just memory transfers when using UMA, and proper profiling and analysis should precede the coding effort. The paper that inspired this work focused on the fact that UMA operates using a paging mechanism [1] that favors accessing specific regions of a buffer rather than touching every element. This conclusion should be taken into consideration as well; when using unified buffers, it pays to be aware of which regions of data you'll need access to, and where and when you'll need them within the program's operation. A criticism that Landaverde et. al. brought up with UMA is that, by abstracting away the actual data transfer, you lose out on potential optimizations that could be made by intelligently transferring your data. We validate this statement, and furthermore using the unified buffers does not yield a huge improvement to workflow or productivity. Given the nature of the Tegra K1's hardware, using unified buffers to avoid superfluous memcpys can be a trivial change that will both simplify and optimize your code so it does represent some utility for a small coding effort penalty.

Code examples hosted at http://meegs.mit.edu

[1] An Investigation of Unified Memory Access Performance in CUDA, Landaverde et al. http://www.ieee-hpec.org/2014/CD/index_htm_files/FinalPapers/103.pdf

978-1-4673-9286-0/15/$31.00 ©2015 IEEE