Griffin Kuchar - GPU Project 1, Part 2 Written Discussion

The transition of switching from CUDA distributed memory to CUDA unified memory simplified and reduced the quantity of code, but overall increased the memory management time overhead. To accomplish this transition, I replaced all standard *malloc()* calls for the host, *cudaMalloc()* calls for the device, and all *cudaMemcpy()* from host to device calls with three trivial *cudaMallocManaged()* calls for the host. Additionally, prefetching the data onto the device greatly increased the performance of the kernel execution average time by allowing the device to have quick access to all the data it needed to compute with, without the overhead of making data page requests. This pre-calculation step was done using *cudaGetDevice()* and *cudaMemPrefetchAsync(),* and required additional synchronization with *cudaDeviceSynchronize()*. The only other changes needed were trivial, and consistent of synchronizing after the *vectorAdd<<<>>>()* call, removing the preceding *cudaMemcpy()* calls, and swapping out the standard *cudaFree()* and *free()* device and host memory deallocation with *cudaFree()* calls on the host.

With distributed memory, data movement costs are explicit and occur exactly when *cudaMemcpy()* is called, making them directly visible in timing measurements. However, with Unified Memory, the runtime handles data migration implicitly through demand paging, which shifts these costs to whenever the GPU actually accesses the data during kernel execution. Without prefetching, page faults occur during the kernel's computation phase, effectively hiding transfer overhead inside what appears to be kernel execution time. Prefetching with *cudaMemPrefetchAsync()* reintroduces explicit control over when migration happens, but the

cost now appears in a separate prefetch step rather than bundled with memory copies. Here is a calculated average result I observed from one execution of my vecAddUnifed.cu program:

The average CPU read-back / verification access time: 731.591553 milliseconds

The average kernel execution time: 0.904149 milliseconds

The number of runs used to compute the averages: 24