UNIVERSITY OF CALIFORNIA

SANTA CRUZ

• • • •

# A Heterogeneous CPU/GPU Implementation Of PageRank and Single Source Shortest Path On An Embedded Device

By

JJ Bowen

August 2022

# 1 Introduction

The graph algorithms PageRank (PR) and Single Source Shortest Path (SSSP) are widely used in the world of modern computing. Running them on a Graphics Processing Unit (GPU) can increase their speed significantly, but certain segments of the graph are often better suited to CPU calculations. By splitting the graph into segments, one can run each segment on the processor and implementation that best suits its structure. However, this creates significant overhead, as the memory must be copied back and forth between the two processors, which slows the function down significantly. By running the SSSP algorithm on an embedded device, or a device where the CPU and GPU can access shared memory, one can increase the speed by a factor of 1.13x, however, this approach comes with its own challenges. The CPU and GPU can access the same memory, but not at the same time. This means that quite a bit of parallelization is lost in the process, which is a significant slowdown to the potential that could be achieved without this. This slowdown is particularly apparent on PR, which sees a speed decrease of over 3x. If a device existed where the CPU and GPU could both access the same memory at the same time, then I believe there would be a much more significant speed increase.

# 2 Different Device Schedules

In the GitHub repository [1] I used, which was designed by Christopher Liu, Sanya Srivastava, Professor Tyler Sorensen and myself, there is an existing framework to get benchmarking results from running the two algorithms on different processors and approaches and to use these results to create a heterogeneous

schedule. Depending on which device is used, however, these schedules can look radically different. The first device I used had a much larger GPU in comparison to the size of its CPU then the second one. The schedule of this device is shown below for running PR with 24 segments on a Kronecker graph [3] of scale 20.

| Segment | Device 0<br>Intel i7-9700K<br>[1.68 ms] | Device 1<br>NVIDIA Quadro RTX 4000<br>[2.09 ms] |
|---|---|---|
| 0 | | PR GPU block-red 256 1024<br>[0.09 ms] |
| 1 | | PR GPU block-red 1024 256<br>[0.07 ms] |
| 2 | | PR GPU block-red 1024 256<br>[0.09 ms] |
| 3 | | PR GPU block-red 4096 64<br>[0.07 ms] |
| 4 | | PR GPU block-red 4096 64<br>[0.07 ms] |
| 5 | | PR GPU block-red 4096 64<br>[0.09 ms] |
| 6 | | PR GPU warp-red 256 1024<br>[0.08 ms] |
| 7 | | PR GPU warp-red 256 1024<br>[0.07 ms] |
| 8 | | PR GPU warp-red 256 1024<br>[0.07 ms] |
| 9 | | PR GPU warp-red 256 1024<br>[0.07 ms] |
| 10 | | PR GPU warp-red 256 1024<br>[0.10 ms] |
| 11 | | PR GPU warp-red 256 1024<br>[0.10 ms] |
| 12 | | PR GPU warp-red 256 1024<br>[0.10 ms] |
| 13 | | PR GPU warp-red 256 1024<br>[0.10 ms] |
| 14 | | PR GPU warp-red 256 1024<br>[0.10 ms] |
| 15 | | PR GPU block-red 4096 64<br>[0.11 ms] |
| 16 | PR CPU one-to-one<br>[0.56 ms] | |
| 17 | PR CPU one-to-one<br>[0.56 ms] | |
| 18 | PR CPU one-to-one<br>[0.56 ms] | |
| 19 | | PR GPU warp-red 256 1024<br>[0.12 ms] |
| 20 | | PR GPU warp-red 256 1024<br>[0.12 ms] |
| 21 | | PR GPU warp-red 256 1024<br>[0.13 ms] |
| 22 | | PR GPU warp-red 256 1024<br>[0.17 ms] |
| 23 | | PR GPU one-to-one 256 1024<br>[0.17 ms] |

The schedule for the second device, also running PR with 24 segments on the same graph is shown below.

| Segment | Device 0<br>Intel i7-9700K<br>[94.20 ms] | Device 1<br>NVIDIA Quadro RTX 4000<br>[102.24 ms] |
|---|---|---|
| 0 | | PR GPU block-red 2048 128<br>[3.22 ms] |
| 1 | | PR GPU warp-red 256 1024<br>[2.74 ms] |
| 2 | | PR GPU warp-red 256 1024<br>[3.48 ms] |
| 3 | | PR GPU warp-red 256 1024<br>[3.41 ms] |
| 4 | | PR GPU warp-red 256 1024<br>[3.38 ms] |
| 5 | | PR GPU warp-red 256 1024<br>[4.17 ms] |
| 6 | | PR GPU warp-red 256 1024<br>[4.53 ms] |
| 7 | | PR GPU warp-red 256 1024<br>[4.48 ms] |
| 8 | | PR GPU warp-red 256 1024<br>[5.18 ms] |
| 9 | | PR GPU warp-red 256 1024<br>[4.63 ms] |
| 10 | | PR GPU warp-red 256 1024<br>[5.55 ms] |
| 11 | | PR GPU warp-red 256 1024<br>[5.88 ms] |
| 12 | | PR GPU warp-red 256 1024<br>[5.91 ms] |
| 13 | | PR GPU warp-red 256 1024<br>[5.86 ms] |
| 14 | | PR GPU warp-red 256 1024<br>[5.96 ms] |
| 15 | | PR GPU warp-red 256 1024<br>[6.81 ms] |
| 16 | PR CPU one-to-one<br>[18.08 ms] | |
| 17 | PR CPU one-to-one<br>[17.92 ms] | |
| 18 | PR CPU one-to-one<br>[17.99 ms] | |
| 19 | | PR GPU warp-red 256 1024<br>[7.85 ms] |
| 20 | PR CPU one-to-one<br>[19.88 ms] | |
| 21 | PR CPU one-to-one<br>[20.33 ms] | |
| 22 | | PR GPU warp-red 256 1024<br>[9.75 ms] |
| 23 | | PR GPU one-to-one 256 1024<br>[9.45 ms] |

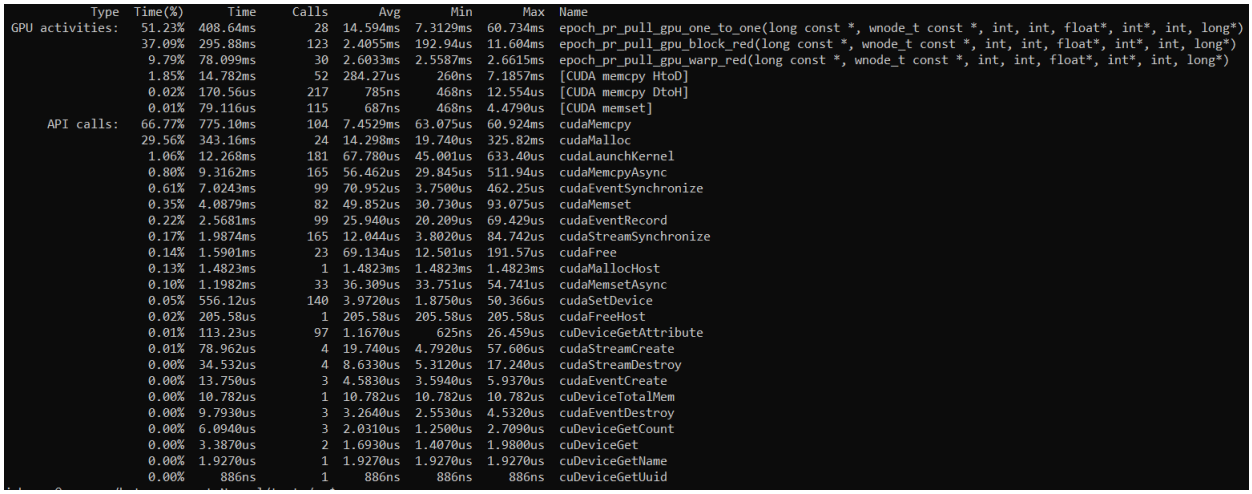
Though these schedules are somewhat similar, they do differ a bit in certain places. The first schedule ends up using the CPU for 3 out of 24 segments, which

are clustered together in one continuous block, while the second uses it for 5, which are almost continuous, with one small break. This outcome is to be expected, because the second machine has a much stronger CPU compared to its GPU, so it makes sense it would utilize that CPU for more segments.

# 3 Approaches

The different CPU and GPU approaches I used are outlined in a previous paper [4], so I will only go over them briefly. There is one CPU approach, and three GPU ones, one to one, warp reduce, and block reduce. These four implementations each have strengths and weaknesses depending on the structure of the graph they are operating upon. The scheduler, previously discussed in section 2, can split the graph into segments, and find on which segments to use which approaches. With this information, it then generates a new heterogeneous function that runs these segments.

In this heterogeneous version, the segments using the GPU approaches are run in sequence, each receiving the same input array, which they read from and write to. The CPU segments operate the same way and are run in parallel with the GPU ones. Once these are both complete, the function checks if any values were updated in any of the segments. If any have changed, the values from the GPU segments are copied into the CPU array and vice versa. This process is repeated until there are no updates. A breakdown of how the function utilizes its time using the command *nvprof* is shown below.

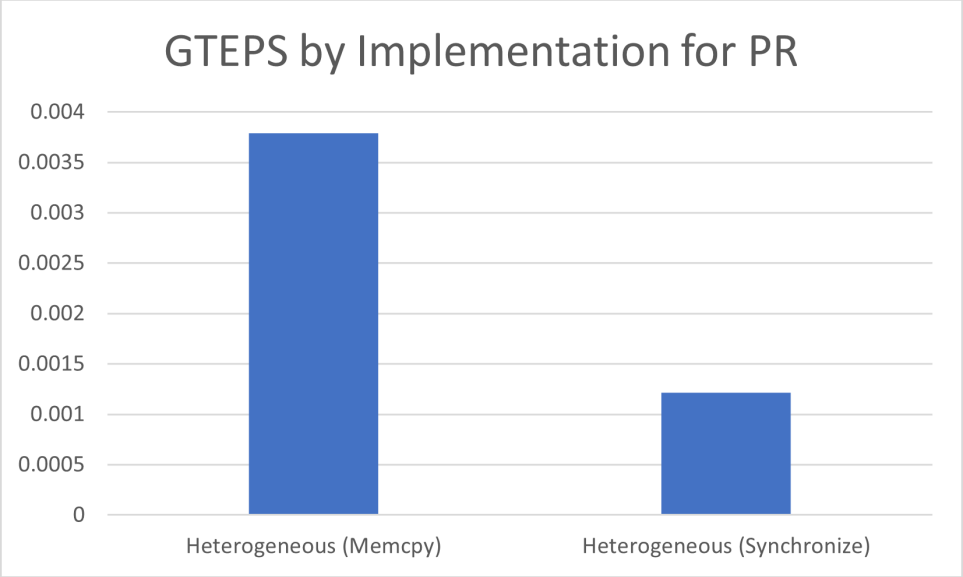


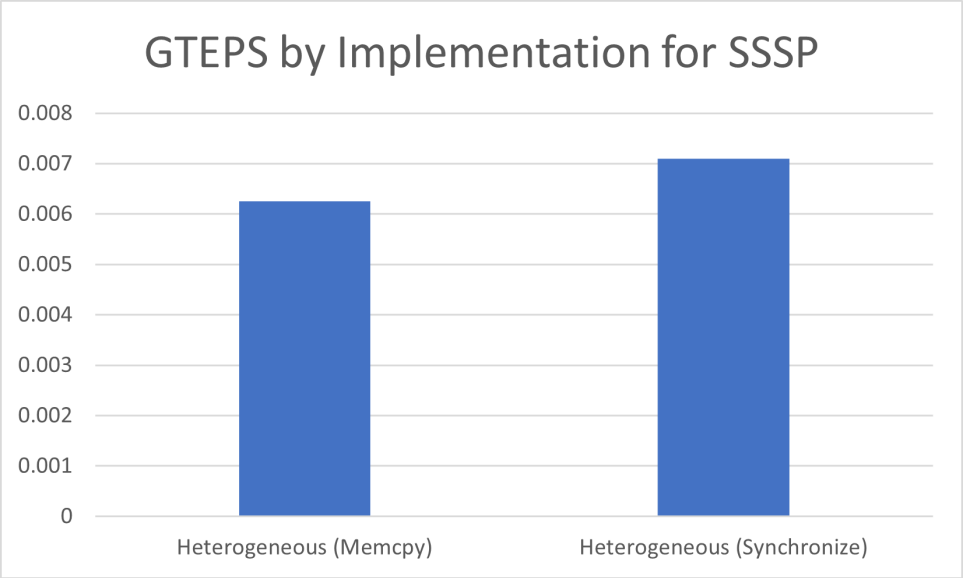
| Type | Time(%) | Time | Calls | Avg | Min | Max | Name |
|---|---|---|---|---|---|---|---|
| GPU activities: | 51.23% | 408.64ms | 28 | 14.594ms | 7.3129ms | 60.734ms | epoch_pr_pull_gpu_one_to_one(long const *, wnode_t const *, int, int, float*, int*, int, long*) |
| | 37.09% | 295.88ms | 123 | 2.4055ms | 192.94us | 11.604ms | epoch_pr_pull_gpu_block_red(long const *, wnode_t const *, int, int, float*, int*, int, long*) |
| | 9.79% | 78.099ms | 30 | 2.6033ms | 2.5587ms | 2.6615ms | epoch_pr_pull_gpu_warp_red(long const *, wnode_t const *, int, int, float*, int*, int, long*) |
| | 1.85% | 14.782ms | 52 | 284.27us | 260ns | 7.1857ms | [CUDA memcpy HtoD] |
| | 0.02% | 170.56us | 217 | 785ns | 468ns | 12.554us | [CUDA memcpy DtoH] |
| | 0.01% | 79.116us | 115 | 687ns | 468ns | 4.4790us | [CUDA memset] |
| API calls: | 66.77% | 775.10ms | 104 | 7.4529ms | 63.075us | 60.924ms | cudaMemcpy |
| | 29.56% | 343.16ms | 24 | 14.298ms | 19.740us | 325.82ms | cudaMalloc |
| | 1.06% | 12.268ms | 181 | 67.780us | 45.001us | 633.40us | cudaLaunchKernel |
| | 0.80% | 9.3162ms | 165 | 56.462us | 29.845us | 511.94us | cudaMemcpyAsync |
| | 0.61% | 7.0243ms | 99 | 70.952us | 3.7500us | 462.25us | cudaEventSynchronize |
| | 0.35% | 4.0879ms | 82 | 49.852us | 30.730us | 93.075us | cudaMemset |
| | 0.22% | 2.5681ms | 99 | 25.940us | 20.209us | 69.429us | cudaEventRecord |
| | 0.17% | 1.9874ms | 165 | 12.044us | 3.8020us | 84.742us | cudaStreamSynchronize |
| | 0.14% | 1.5901ms | 23 | 69.134us | 12.501us | 191.57us | cudaFree |
| | 0.13% | 1.4823ms | 1 | 1.4823ms | 1.4823ms | 1.4823ms | cudaMallocHost |
| | 0.10% | 1.1982ms | 33 | 36.309us | 33.751us | 54.741us | cudaMemsetAsync |
| | 0.05% | 556.12us | 140 | 3.9720us | 1.8750us | 50.366us | cudaSetDevice |
| | 0.02% | 205.58us | 1 | 205.58us | 205.58us | 205.58us | cudaFreeHost |
| | 0.01% | 113.23us | 97 | 1.1670us | 625ns | 26.459us | cuDeviceGetAttribute |
| | 0.01% | 78.962us | 4 | 19.740us | 4.7920us | 57.606us | cudaStreamCreate |
| | 0.00% | 34.532us | 4 | 8.6330us | 5.3120us | 17.240us | cudaStreamDestroy |
| | 0.00% | 13.750us | 3 | 4.5830us | 3.5940us | 5.9370us | cudaEventCreate |
| | 0.00% | 10.782us | 1 | 10.782us | 10.782us | 10.782us | cuDeviceTotalMem |
| | 0.00% | 9.7930us | 3 | 3.2640us | 2.5530us | 4.5320us | cudaEventDestroy |
| | 0.00% | 6.0940us | 3 | 2.0310us | 1.2500us | 2.7090us | cuDeviceGetCount |
| | 0.00% | 3.3870us | 2 | 1.6930us | 1.4070us | 1.9800us | cuDeviceGet |
| | 0.00% | 1.9270us | 1 | 1.9270us | 1.9270us | 1.9270us | cuDeviceGetName |
| | 0.00% | 886ns | 1 | 886ns | 886ns | 886ns | cuDeviceGetUuid |

This approach is somewhat problematic, because copying the memory is a slow process that can often take longer than the calculation phase. This is where one can see the advantages of an embedded device. Embedded devices have shared memory that can be accesses by the GPU and the CPU, which allows PR and SSSP to be run without any memory copying. This is done by first running the GPU segments, with the same input array as the previous approach, and then, once these are complete, passing the input array into the CPU segments. The *nvprof* results of this approach are shown below.

| Type | Time(%) | Time | Calls | Avg | Min | Max | Name |
|---|---|---|---|---|---|---|---|
| GPU activities: | 66.63% | 389.85ms | 63 | 6.1881ms | 1.1483ms | 16.408ms | epoch_pr_pull_gpu_block_red(long const *, wnode_t const *, int, int, float*, int*, int, long*) |
| | 31.20% | 182.53ms | 66 | 2.7656ms | 609.40us | 23.242ms | epoch_pr_pull_gpu_warp_red(long const *, wnode_t const *, int, int, float*, int*, int, long*) |
| | 1.86% | 10.897ms | 72 | 151.35us | 470ns | 7.2349ms | [CUDA memcpy HtoH] |
| | 0.30% | 1.7343ms | 4 | 433.59us | 1.9280us | 1.2706ms | [CUDA memcpy HtoD] |
| | 0.01% | 71.145us | 95 | 748ns | 468ns | 8.1790us | [CUDA memset] |
| | 0.00% | 28.016us | 34 | 824ns | 728ns | 1.7700us | [CUDA memcpy DtoH] |
| API calls: | 40.59% | 551.18ms | 75 | 7.3491ms | 93.909us | 23.490ms | cudaMemcpy |
| | 32.63% | 443.15ms | 12 | 36.929ms | 114.54us | 431.40ms | cudaMallocManaged |
| | 23.67% | 321.40ms | 136 | 2.3632ms | 1.2145ms | 4.6773ms | cudaDeviceSynchronize |
| | 0.99% | 13.467ms | 129 | 104.39us | 60.001us | 206.26us | cudaLaunchKernel |
| | 0.65% | 8.7629ms | 61 | 143.65us | 86.565us | 975.76us | cudaMemset |
| | 0.33% | 4.5416ms | 35 | 129.76us | 108.91us | 301.16us | cudaMemcpyAsync |
| | 0.31% | 4.1518ms | 5 | 830.37us | 34.481us | 2.1689ms | cudaMalloc |
| | 0.23% | 3.1397ms | 68 | 46.171us | 23.073us | 465.22us | cudaEventRecord |
| | 0.20% | 2.7610ms | 17 | 162.41us | 29.636us | 463.82us | cudaFree |
| | 0.17% | 2.3723ms | 103 | 23.032us | 3.9060us | 451.58us | cudaStreamSynchronize |
| | 0.12% | 1.6591ms | 34 | 48.795us | 39.585us | 79.012us | cudaMemsetAsync |
| | 0.05% | 721.43us | 68 | 10.609us | 4.1670us | 20.730us | cudaEventSynchronize |
| | 0.03% | 456.11us | 144 | 3.1670us | 1.6140us | 27.084us | cudaSetDevice |
| | 0.01% | 109.33us | 97 | 1.1270us | 625ns | 25.105us | cuDeviceGetAttribute |
| | 0.01% | 78.648us | 3 | 26.216us | 5.1040us | 63.856us | cudaStreamCreate |
| | 0.00% | 25.885us | 3 | 8.6280us | 5.3120us | 14.323us | cudaStreamDestroy |
| | 0.00% | 10.208us | 1 | 10.208us | 10.208us | 10.208us | cuDeviceTotalMem |
| | 0.00% | 9.7910us | 2 | 4.8950us | 3.2810us | 6.5100us | cudaEventCreate |
| | 0.00% | 7.3970us | 2 | 3.6980us | 2.8130us | 4.5840us | cudaEventDestroy |
| | 0.00% | 5.7820us | 3 | 1.9270us | 1.0940us | 2.7610us | cuDeviceGetCount |
| | 0.00% | 2.9170us | 2 | 1.4580us | 1.2500us | 1.6670us | cuDeviceGet |
| | 0.00% | 1.9790us | 1 | 1.9790us | 1.9790us | 1.9790us | cuDeviceGetName |
| | 0.00% | 833ns | 1 | 833ns | 833ns | 833ns | cuDeviceGetUuid |

From the results above, it is clear that Device to Host (DtoH) memory copying, the type that is being done at the end of each iteration, has been drastically reduced. The performance results of doing this vary between algorithms and are shown below in giga edges traversed per second (GTEPS).






From the results, one can see that running SSSP without memory copying is about 1.13x faster than with, while PR is only .33x the speed. This is unsurprising, because the calculation phase in PR is much more complicated than that of SSSP. Because of this, the time increase of removing memory copying is much less significant than that lost by the removal of parallelization.

# 4 Device Limitations

Looking at the previously described approach, one might wonder why the CPU and GPU segments are not run in parallel with one another. While this practice would most likely be significantly faster, it is unfortunately made impossible by the way the device defines a bus error. When the CPU and GPU are both given the same array, in shared memory, to read from and write to at the same time this triggers a bus error. Though unfortunate in this situation, it is understandable for the device to behave in such a way. What is much more problematic, however,

is how the device handles the situation where the CPU and GPU are accessing different shared memory at the same time. In this situation, the actions of one processor should have no effect on the other, however, this is not the case. If the GPU kernel is running, the CPU cannot write to any array in shared memory. Worse still, it can read from an array in shared memory only if it is not writing to any memory, even its own. This is true even if the GPU is given an empty function, with no parameters and no code in the function body, which implies that when the GPU kernel launches, it locks up all memory it has access to. This is an incredibly suboptimal practice that, if removed, could lead to significant performance improvements.

# 5 Conclusion and Future Work

The results show a small, but nonetheless significant, increase in the speed of the SSSP algorithm when run using the shared memory method over one that relies on memory copying. More importantly, they highlight a behavior in embedded devices that could be improved upon, although this is up to the devices designers to correct. If this flaw is eventually remedied, then future work on this subject could include an approach to both of these algorithms that utilizes both parallelization and shared memory, which could be significantly more effective than the implementations that are used today.

# Bibliography

[1] JJ Bowen, Chris Liu, Tyler Sorensen & Sanya Srivastava “Hetero-Compute.” GitHub, https://github.com/chrisliu/hetero-compute.

[2] Brin, S. & Page, L. (1998). The Anatomy of a Large-scale Hypertextual Web Search Engine. Comput. Netw. ISDN Syst., 30, 107–117. doi: 10.1016/S0169-7552(98)00110-X

[3] Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C. & Ghahramani, Z. (2010). Kronecker graphs: An approach to modeling networks. The Journal of Machine Learning Research, 11, 985–1042.

[4] JJ Bowen ”A Heterogeneous CPU/GPU Implementation Of PageRank” 2022