EPIC MEMO #003

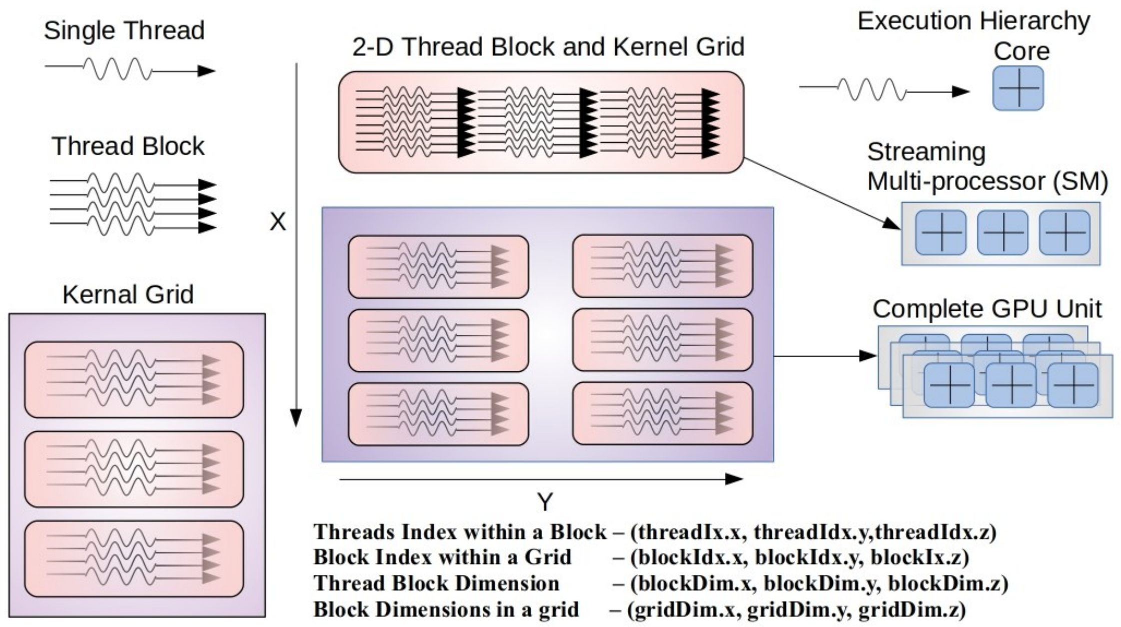
ARIZONA STATE UNIVERSITY
TEMPE, ARIZONA 85287

Optimization of Romein Kernel in EPIC

Hariharan Krishnan
20, January, 2020

GPU Primer

Single thread is a copy of a functionality or process to be implemented on the GPU. Figure 1 describes the basic distribution of the process / function kernel across threads, thread blocks and kernel grids, their execution hierarchy on the physical units within a GPU device. Each of the above components is indexed through



Threads Index within a Block – (threadIx.x, threadIdx.y,threadIdx.z)
Block Index within a Grid – (blockIdx.x, blockIdx.y, blockIx.z)
Thread Block Dimension – (blockDim.x, blockDim.y, blockDim.z)
Block Dimensions in a grid – (gridDim.x, gridDim.y, gridDim.z)

Figure 1:Basic GPU - terminologies ; Representation of process distribution

GPU Memory Distribution

Figure 2 describes the memory access distribution within the GPU device across the threads, thread blocks and kernel grid. The directions specified in Figure 2 represent read-only and read-write capability. Data stored in register memory is visible only to the thread that writes it and lasts for the duration of the thread. Local memory has the same scope as register memory but comparable in speed to the global memory. Constant and Texture memory are read-only. Constant memory is used for data that doesn't change over the course of kernel execution. Texture memory is useful when the reads are physically adjacent in memory and reduces memory traffic which in-turn increases performance.

Romein Kernel

The romein CUDA-kernel is an implementation of a GPU-accelerated gridding algorithm. It is an efficient work distribution strategy for GPUs to efficiently convolve the raw voltages (in the context of EPIC) from individual antennas of a compact radio telescope array to create real-time radio sky images.

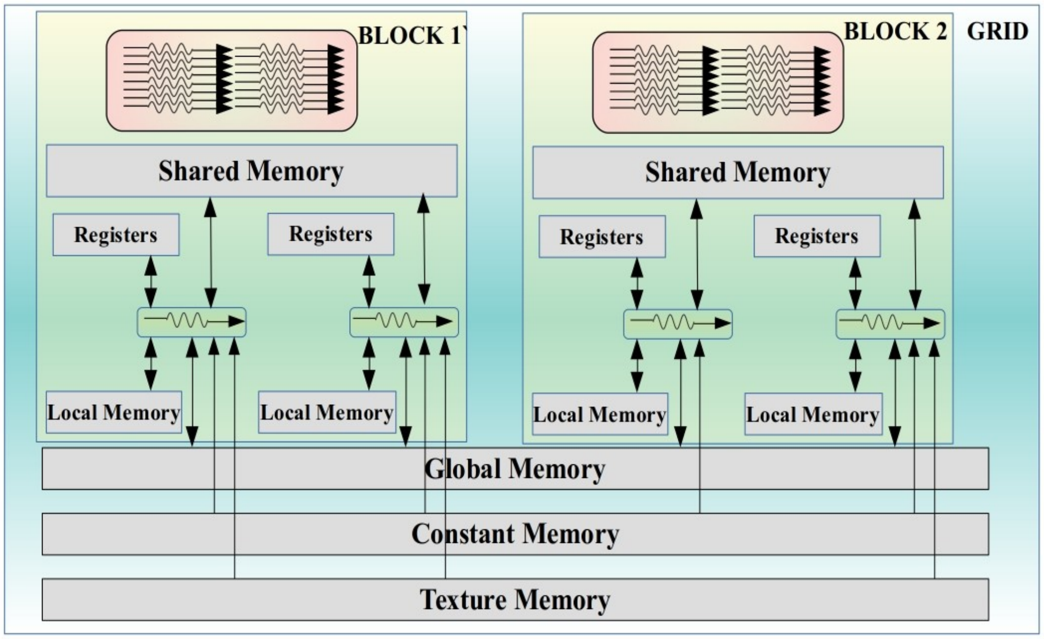


Figure 2: Different GPU Memory sections and Access Distribution across threads, thread blocks and kernel grid

The latest implementation of the kernel, however, had several performance bottlenecks as mentioned below :

1. Number of threads per block was insufficient causing poor thread occupancy on the GPU device for the kernel

2. “For” loops that introduced severe intra-warp divergences causing performance degradation.

3. Single thread does the job of gridding a single time sample for all 256 antennas in a for loop which was highly time-consuming.

4. There are two versions of the romein kernel on which makes use of shared memory with function name romein_kernel_sloc() and the other kernel romein_kernel() which does not use shared memory. Only the former function is launched when the threads are spawned via kernel call/launch. Although this does not lead to a performance improvement, it is a clean-up measure for the code to remove parts of the code that are not used.

In the current updated version of the kernel, the above issues are addressed and the duration of the kernel is reduce to ~ 35 % that of the original romein kernel. The kernel run-time duration is a key parameter and can be considered as an indirect measure of real-time performance in-terms of maximum operating bandwidth that can be processed.

1. **Number of threads per block** :

The original romein kernel implementation spawns 8 threads per thread block defined by the following line in the code

"dim3 block(8,1);
dim3 grid(nbatch*npol,1);" **(to include github repos, discuss with Adam)**

The number of thread blocks is set to be the total number of time samples and frequency channels. The individual threads of the thread block are set to grid the voltage samples of the antennas. The above was modified such that there are as many threads per thread block as there are antennas in the array. In the present case, it is set to 256 threads with each thread gridding the individual 256 antennas of the LWA-SV.

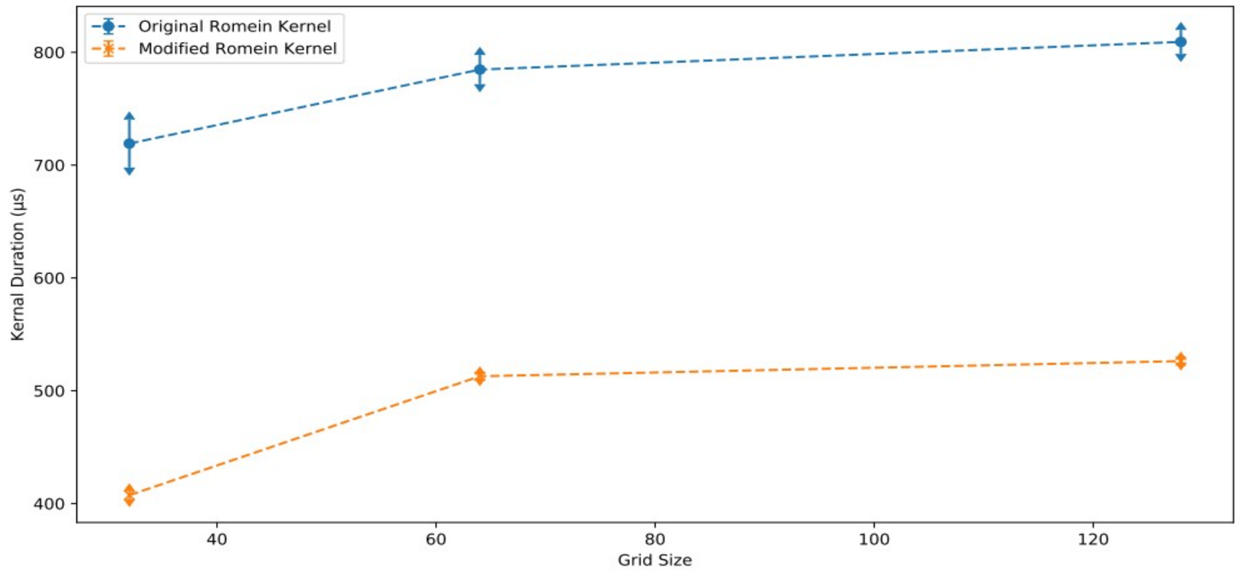


Figure 3: Comparison of the Duration of the original (blue) and modified (orange) Romein kernel after eliminating the thread divergence

2. **Intra-Warp Divergence :**

Threads in a thread block are executed in multiples of 32 threads which is referred as warps. When not all of the threads within a warp or a block pass a loop or a conditional statement, the warps enter a condition called divergence, where some threads execute the loop statements while the rest don't.

The for loop in original romein : **(to include github repos)**

```
for(int i = threadIdx.x; i < maxsupport * maxsupport; i += blockDim.x) {}
```

The above "for" loop ensures the inclusion of a matrix for the imprint of an antenna, however since the iterator is thread number dependent, the condition introduces intra-warp divergence. This loop was eliminated and a non-thread dependent loop was introduced to retain this feature as follows

for(int mm=0;mm<maxsupport*maxsupport;++mm){} ; this line is to iterate for the footprint matrix of the antenna, may not be the ideal implementation as not-parallelized

This was a major update which decreased the kernel run-time duration significantly by upto 50 % (refer figure 3).

3. **Job per thread :**

Every single thread iterates across all antennas of the array in a loop adding to the duration of the kernel. This loop was also eliminated to include gridding only two polarizations of an antenna as opposed. So effectively one thread block grids all antennas and both X-Y polarizations. Each thread is spawned to grid two polarizations of one antenna, effectively gridding 256x2 voltage samples per thread block.Current version of the line " dim3 block(tile_ant_x,tile_ant_y); – Distributes threads across a 2-D dim3 grid(tile_block_x,tile_block_y,blk_cnt);" – Distributes the thread blocks in a 3D-kernel grid

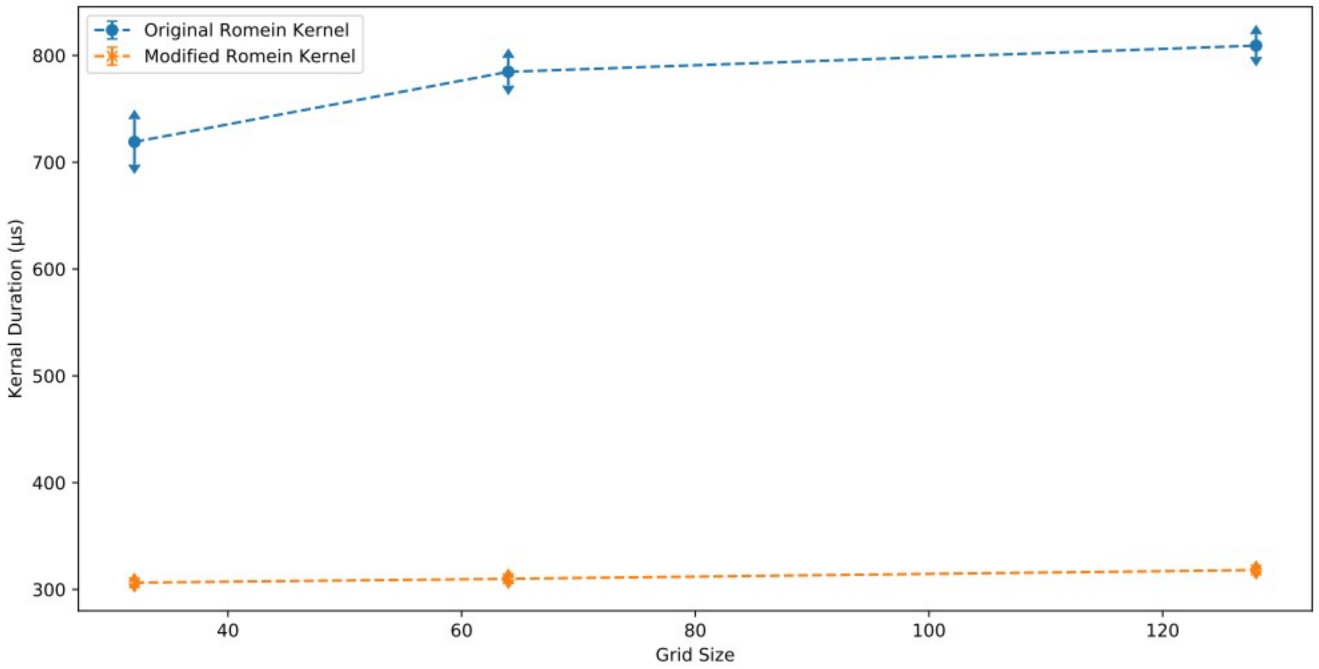


Figure 4: Comparison of the Duration of the original (blue) and modified (orange) Romein kernel after increasing jobs per thread

Comparison of Images :

A sample sky image produced using EPIC with original and modified Romein kernels is shown in Figures 5 & 6. Figure 7 shows the difference image of the two.

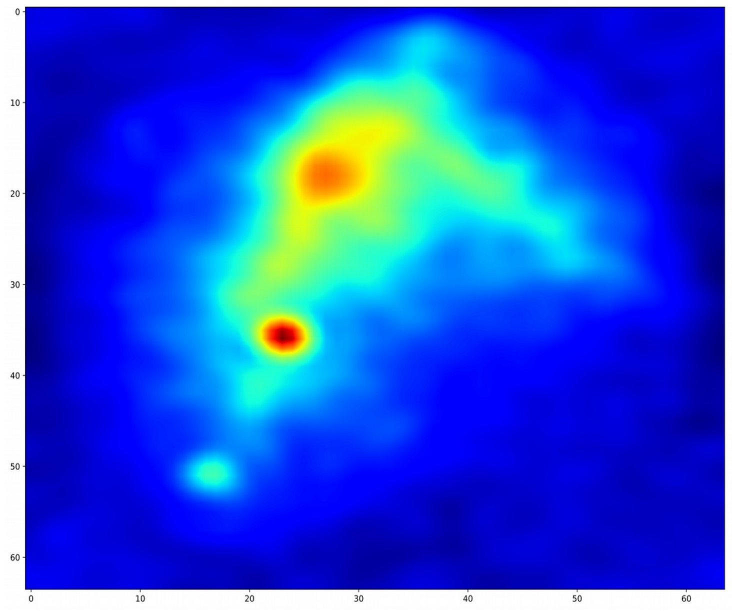


Figure 5: Sample Sky Image using the New Romein Implementation

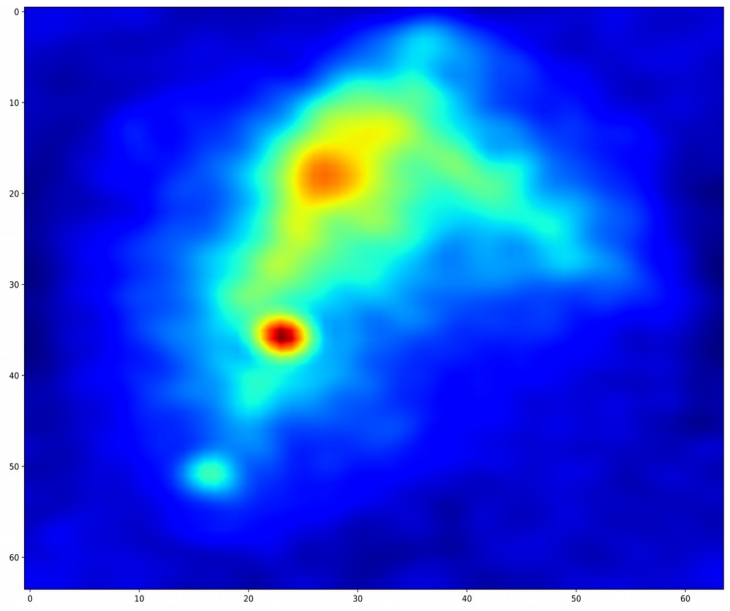


Figure 6: Sky Image using the original Romein Implementation

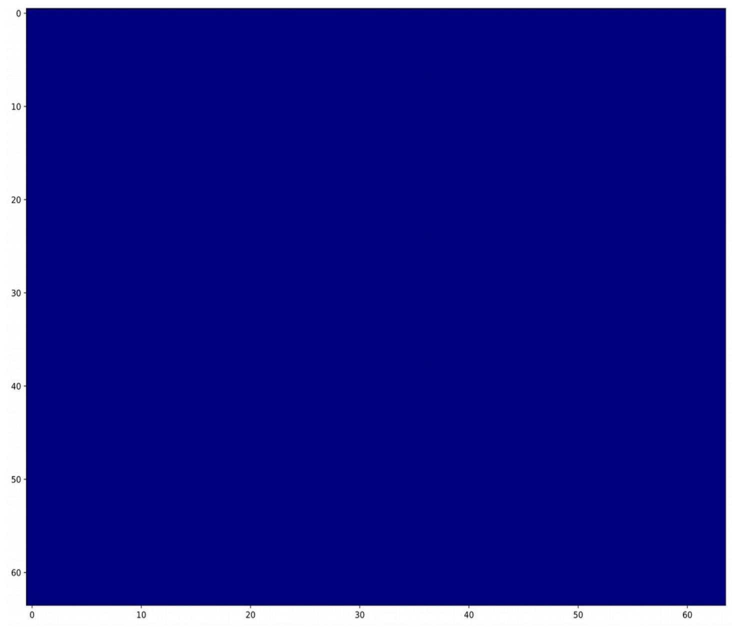


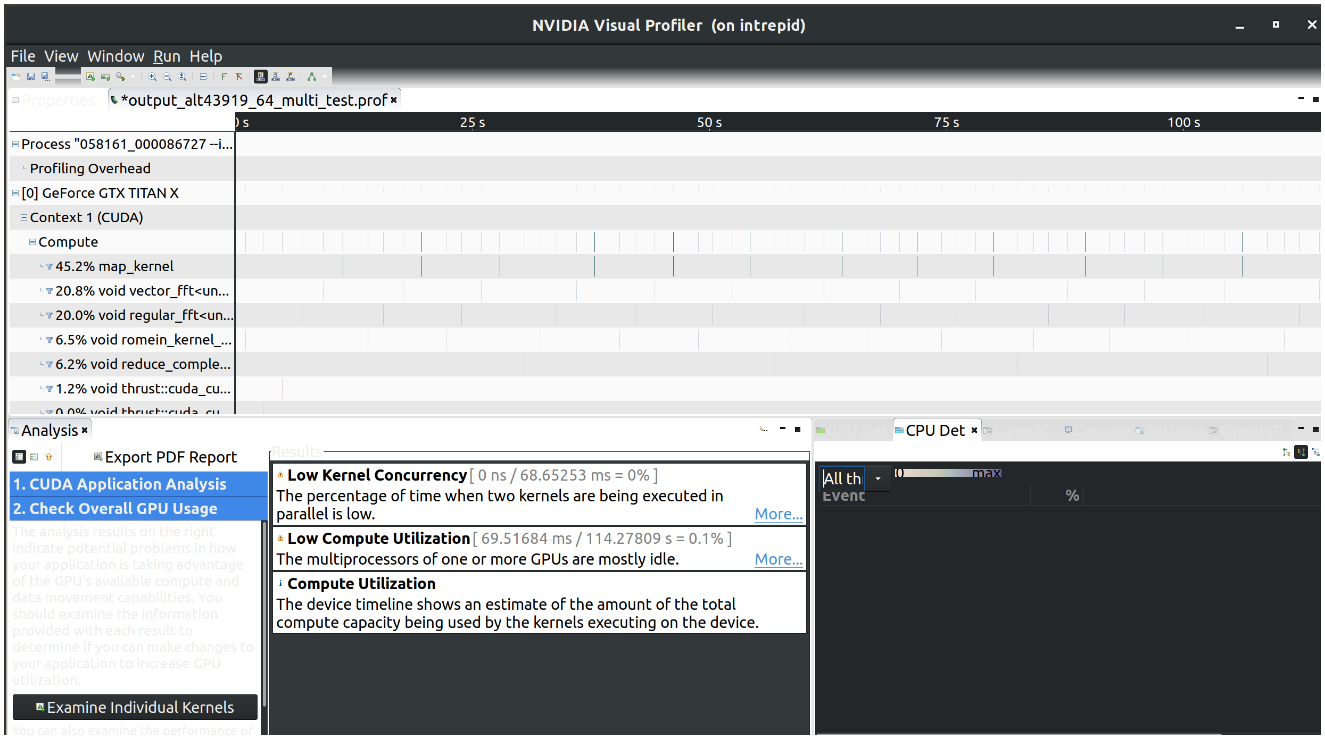
Figure 7: Difference Image of the New and Old Romein kernels

NVIDIA Profiler :

The profiler is a cross-platform toolset provided by NVIDIA to give a feedback on the performance of a CUDA code and to optimize it based on the comments from the profiler.

Command for profiling the code : **nvprof --analysis-metrics --profile-child-processes --devices 0 -o output_alt%p_64_multi_test.prof -t 120 python LWA_bifrost_alt_ordering.py --offline --tbnfile=/ data5/LWA_SV_data/data_raw/TBN/Jupiter/058161_000086727 --imagesize 64 --imageres 1.79057 --nts 512 --channels 4 --accumulate 50 --ints_per_file 40**

The generated file is then analyzed in the NVIDIA Visual profiler (NVVP) to get the feedback and hints on code optimization. Figure 8 shows a screenshot of what one can expect to see in the NVVP


NVIDIA Visual Profiler (on intrepid)
File View Window Run Help
*output_alt43919_64_multi_test.prof
0 s
25 s
50 s
75 s
100 s
Process "058161_000086727 --i...
Profiling Overhead
[0] GeForce GTX TITAN X
Context 1 (CUDA)
Compute
45.2% map_kernel
20.8% void vector_fft<un...
20.0% void regular_fft<un...
6.5% void romein_kernel_...
6.2% void reduce_comple...
1.2% void thrust::cuda_cu...
0.0% void thrust::cuda_cu...
Analysis
Export PDF Report
1. CUDA Application Analysis
2. Check Overall GPU Usage
Examine Individual Kernels
Results
CPU Det
All th
0
max
Event
%

The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.

You can also examine the performance of

Low Kernel Concurrency [0 ns / 68.65253 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Compute Utilization [69.51684 ms / 114.27809 s = 0.1%]
The multiprocessors of one or more GPUs are mostly idle. More...

Compute Utilization
The device timeline shows an estimate of the amount of the total compute capacity being used by the kernels executing on the device.

Figure 8: NVIDIA Visual Profiler Display