Geant4 GPU Code Analysis

Azamat Mametjanov, Paul Hovland and Boyana Norris

LANS Performance Group
Mathematics and Computer Science Division
Argonne National Laboratory

*Collaboration between US DOE **HEP** Geant4 Reengineering and*
*US DOE SciDAC institute **SUPER**: Sustained Performance, Energy and Resilience*

February 5, 2014

Geant4

- Geant4 is open-source software package for accurate simulation of particles passing through matter with tools for:
  - Geometry of the system
  - Properties and composition of materials
  - Properties of fundamental particles: neutrons, protons, ions, hadrons
  - Physics of interaction of beam particles with detector matter
  - Tracking and detection of collision events
  - Capture, visualization and analysis of particle tracks and events
- Applications are built using Geant4 tools by extending/adding new physics and time-stepping of particle interactions
- Geant4 is a C++ object-oriented framework

Motivation

- Geant4 enables high-precision particle tracking
  - Very high computational intensity
- Recent advances in computer architecture
  - Multi-threaded CPUs
  - Multi-core CPUs
  - Many-core coprocessors/accelerators: e.g. GPUs, MICs
  - Deeper SIMD/vectorization units: e.g. AVX-512
- Need to adapt existing code to new hardware
  - Our current focus is on performance optimization on NVidia GPUs

GPU Prototype

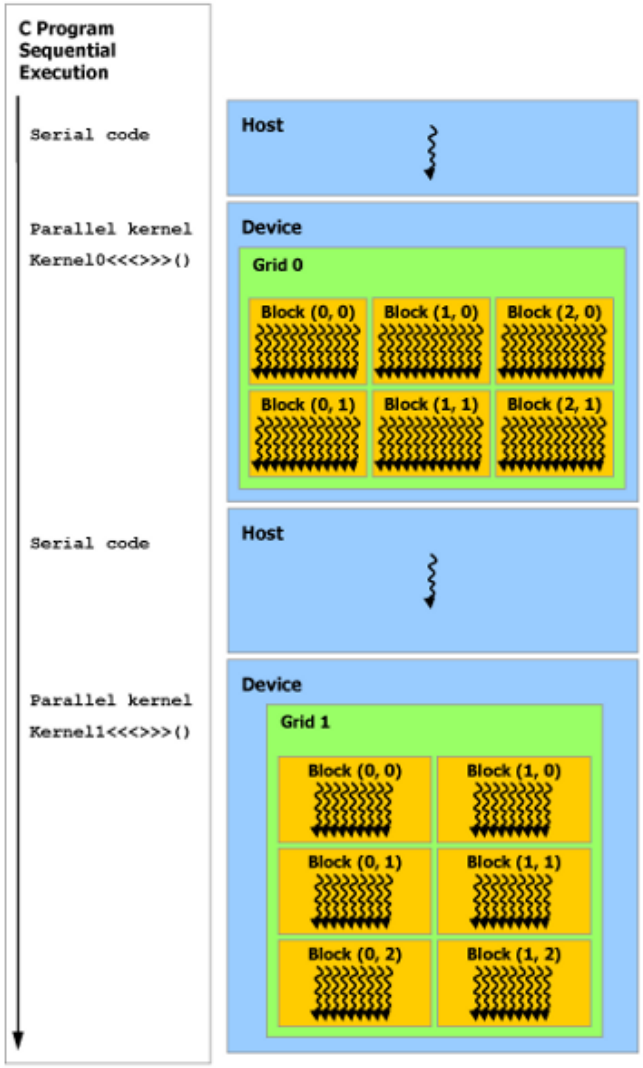
- Available from the Git repo at
  - http://cdcvs.fnal.gov/projects/g4hpcbenchmarks
- Build and run with instructions at
  - g4hpcbenchmarks/GXTracking/gpu/cuda/README
- GPU application prototype – trackingTest2
  - Tracking of 2^{16}=65536 electrons and photons through a magnetic field
- Promising initial acceleration results
  - Simulation time ratio of 1-core CPU vs. GPU: T_{CPU}/T_{GPU} = 18.5x

- Can we accelerate it further?

GPU benchmarking environment

- Hardware
  - Intel Quad-core CPU Q8400 @2.66GHz
  - NVidia GeForce GTX-480 @1.4GHz with 480 cores and 1.5GB GRAM
- Software
  - Linux x86_64 Ubuntu v12.04
  - GNU GCC compiler v4.6.3
  - NVidia CUDA SDK v5.0

Parallelization on a GPU

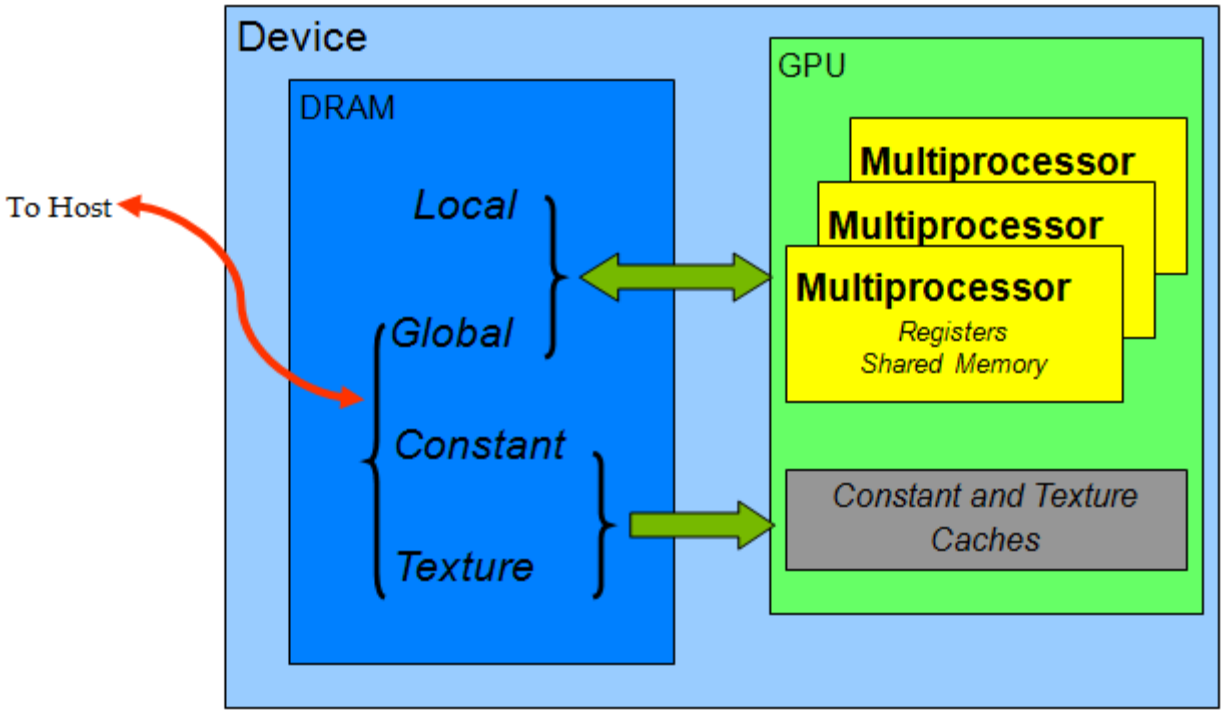


Serial code executes on the host while parallel code executes on the device.

Performance improvements

- Tracking test prototype splits the tracking into
  - curand_setup_kernel: 5.8% of total application time
  - elec_GPIL_kernel: 47%
  - count_by_process_kernel: 2.2%
  - sort_by_process_kernel: 2.7%
  - elec_doit_kernel: 42.3%
- GPU function count_by_process_kernel traverses particle array tracks[] and counts two different types of processes attached to each track
- This was initially prototyped with atomicAdd counters to ensure GPU thread coherency
- Re-implemented the kernel using block-based addition
  - This produced ~17x speedup from 633us to 37us or 0.1% of total time

GPU memory spaces



Performance improvements

```
void count_by_process_gpu(G4int nTracks, GXTrack *tracks,
                          G4int *nbrem, G4int *nioni,
                          int blocksPerGrid, int threadsPerBlock)
 {
-  count_by_process_kernel<<< blocksPerGrid, threadsPerBlock >>>
+  count_by_process_kernel<<< blocksPerGrid, threadsPerBlock, 2*threadsPerBlock*sizeof(int) >>>
     (nTracks, tracks, nbrem, nioni);
 }
```

Performance improvements

```
 __global__
 void count_by_process_kernel(G4int nTracks, GXTrack *tracks,
                                                G4int *nbrem, G4int *nioni)
 {
+  extern __shared__ int sb[];
+  int *si = &sb[blockDim.x];
+  sb[threadIdx.x] = 0;
+  si[threadIdx.x] = 0;
+  __syncthreads();
+
   unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

Performance improvements

```
 while(tid < nTracks) {
    //offset is a global counter for the last array position of secondaries
    if(tracks[tid].proc == 0 ) {
-     atomicAdd(nbrem,1);
-   }
+     //atomicAdd(nbrem,1);
+     sb[threadIdx.x] += 1;
+   }
    if(tracks[tid].proc == 1 ) {
-     atomicAdd(nioni,1);
-   }
+     //atomicAdd(nioni,1);
+     si[threadIdx.x] += 1;
+   }
    tid += blockDim.x * gridDim.x;
  }
+ __syncthreads();
```

Performance improvements

```
+  if (blockDim.x > 512 && threadIdx.x < 512) {
            sb[threadIdx.x]+=sb[threadIdx.x+512]; __syncthreads(); }
+  if (blockDim.x > 256 && threadIdx.x < 256) {
            sb[threadIdx.x]+=sb[threadIdx.x+256]; __syncthreads(); }
+  if (blockDim.x > 128 && threadIdx.x < 128) {
            sb[threadIdx.x]+=sb[threadIdx.x+128]; __syncthreads(); }
+  if (blockDim.x >  64 && threadIdx.x <  64) {
            sb[threadIdx.x]+=sb[threadIdx.x+ 64]; __syncthreads(); }
+  if (threadIdx.x < 32) {
            warpReduce(sb, threadIdx.x); }
+  if (threadIdx.x == 0) {
            atomicAdd(nbrem, sb[0]); }
```

Performance improvements

```
+__device__
+void warpReduce(volatile int *sdata, unsigned int tid) {
+  sdata[tid] += sdata[tid + 32];
+  sdata[tid] += sdata[tid + 16];
+  sdata[tid] += sdata[tid +  8];
+  sdata[tid] += sdata[tid +  4];
+  sdata[tid] += sdata[tid +  2];
+  sdata[tid] += sdata[tid +  1];
+}
```

Before

File View Run Help

nvvp_session2 | nvvp_session2_ex2 | *nvvp_session2_ex1

- Process 24077
  - Thread 46909248
    - Runtime API
    - Driver API
  - Profiling Overhead
- [0] GeForce GTX 480
  - Context 1 (CUDA)
    - MemCpy (HtoD)
    - MemCpy (DtoH)
    - Compute
      - 47.0% [10] elec_GPIL_kernel(curandStateXORWOW*, ...
      - 42.3% [10] elec_doit_kernel(curandStateXORWOW*, ...
      - 5.8% [20] curand_setup_kernel(curandStateXORWOW...
      - 2.7% [10] sort_by_process_kernel(int, GXTrack*, GXTr...
      - 2.2% [10] count_by_process_kernel(int, GXTrack*, int*...
      - 0.0% [60] memset (0)
    - Streams
      - Stream 2
      - Stream 7

Properties | Detail Graphs

count_by_process_kernel(int, GXTrack*, int*, int*)

Name	Value
Start	441.478 ms
End	442.111 ms
Duration	632.847 µs
Grid Size	[32,1,1]
Block Size	[128,1,1]
Registers/Thread	10
Shared Memory/Block	0 bytes
Memory	
Global Load Efficiency	4.5%
Global Store Efficiency	n/a
Local Memory Overhead	0%
DRAM Utilization	4.5% (7.55 GB/s)
Instruction	
Branch Divergence Overhead	0.2%
Total Replay Overhead	84.9%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	64.5%
Global Cache Replay Overhead	20.4%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	20.1%
Theoretical	66.7%
Limiter	Grid Size
L1 Cache Configuration	
Shared Memory Requested	48 KB
Shared Memory Executed	48 KB

Analysis | Details | Console | Settings

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput	instructions issued	DRAM Write Throughput	DRAM Read Throughput	Instruction Replay O
Memcpy HtoD [sync]	359.536 ms	10.368 µs	n/a	n/a	n/a	n/a	n/a	33.82 KB	3.11 GB/s	n/a	n/a	n/a	
Memcpy HtoD [sync]	360.047 ms	535.476 µs	n/a	n/a	n/a	n/a	n/a	1.346 MB	2.46 GB/s	n/a	n/a	n/a	
curand_setup_kernel(curandStateXORWOW*, unsigned long)	420.834 ms	839.378 µs	[32,1,1]	[128,1,1]	30	0	0	n/a	n/a	6125766	3.08 GB/s	655.32 MB/s	
curand_setup_kernel(curandStateXORWOW*, unsigned long)	422.003 ms	840.421 µs	[32,1,1]	[128,1,1]	30	0	0	n/a	n/a	6091000	3.14 GB/s	545.66 MB/s	
memset (0)	423.304 ms	3.68 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	1.04 MB/s	n/a	n/a	n/a	
memset (0)	423.327 ms	1.152 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	3.31 MB/s	n/a	n/a	n/a	
Memcpy HtoD [async]	423.37 ms	1.728 ms	n/a	n/a	n/a	n/a	n/a	4.409 MB	2.49 GB/s	n/a	n/a	n/a	
memset (0)	427.148 ms	1.952 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	1.95 MB/s	n/a	n/a	n/a	
memset (0)	427.231 ms	1.152 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	3.31 MB/s	n/a	n/a	n/a	
memset (0)	427.254 ms	1.152 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	3.31 MB/s	n/a	n/a	n/a	
memset (0)	427.275 ms	1.184 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	3.22 MB/s	n/a	n/a	n/a	
elec_GPIL_kernel(curandStateXORWOW*, GXTrack*, GXTrackLiason*, unsigned l	427.733 ms	13.547 ms	[32,1,1]	[128,1,1]	63	0	0	n/a	n/a	58248984	50.75 GB/s	52.29 GB/s	
count_by_process_kernel(int, GXTrack*, int*, int*)	441.478 ms	632.847 µs	[32,1,1]	[128,1,1]	10	0	0	n/a	n/a	192607	290.07 MB/s	7.26 GB/s	
Memcpy DtoH [sync]	442.113 ms	1.696 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	2.25 MB/s	n/a	n/a	n/a	

After

File View Run Help

nvvp_session2 | nvvp_session2_ex2

- Process 21608
  - Thread -1260112064
    - Runtime API
    - Driver API
  - Profiling Overhead
- [0] GeForce GTX 480
  - Context 1 (CUDA)
    - MemCpy (HtoD)
    - MemCpy (DtoH)
    - Compute
      - 48.0% [10] elec_GPIL_kernel(curandStateXORWOW*...
      - 6.0% [20] curand_setup_kernel(curandStateXORWO...
      - 0.0% [60] memset (0)
      - 43.2% [10] elec_doit_kernel(curandStateXORWOW*, ...
      - 2.7% [10] sort_by_process_kernel(int, GXTrack*, GXT...
      - 0.1% [10] count_by_process_kernel(int, GXTrack*, in...
    - Streams
      - Stream 2
      - Stream 7

Properties | Detail Graphs

count_by_process_kernel(int, GXTrack*, int*, int*)

Name	Value
Start	446.189 ms
End	446.226 ms
Duration	37.18 µs
Grid Size	[32,1,1]
Block Size	[128,1,1]
Registers/Thread	14
Shared Memory/Block	1 KB
Memory	
Global Load Efficiency	4.5%
Global Store Efficiency	n/a
Local Memory Overhead	0%
DRAM Utilization	69.7% (117.88 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	53.8%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	3.2%
Global Cache Replay Overhead	50.6%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	19.2%
Theoretical	66.7%
Limiter	Grid Size
L1 Cache Configuration	
Shared Memory Requested	48 KB
Shared Memory Executed	48 KB

Analysis | Details | Console | Settings

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput	DRAM Write Throughput	DRAM Read Throughput	Instruction Replay Overhead	Global Memory Cache Re
Memcpy HtoD [sync]	365.214 ms	10.528 µs	n/a	n/a	n/a	n/a	n/a	33.82 KB	3.06 GB/s	n/a	n/a	n/a	
Memcpy HtoD [sync]	365.712 ms	536.561 µs	n/a	n/a	n/a	n/a	n/a	1.346 MB	2.45 GB/s	n/a	n/a	n/a	
curand_setup_kernel(curandStateXORWOW*, unsigned long)	425.703 ms	852.002 µs	[32,1,1]	[128,1,1]	30	0	0	n/a	n/a	3.07 GB/s	662.63 MB/s	4.4%	
curand_setup_kernel(curandStateXORWOW*, unsigned long)	426.885 ms	839.646 µs	[32,1,1]	[128,1,1]	30	0	0	n/a	n/a	3.19 GB/s	453.18 MB/s	3.9%	
memset (0)	428.045 ms	2.976 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	1.28 MB/s	n/a	n/a	n/a	
memset (0)	428.088 ms	1.184 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	3.22 MB/s	n/a	n/a	n/a	
Memcpy HtoD [async]	428.13 ms	1.727 ms	n/a	n/a	n/a	n/a	n/a	4.409 MB	2.49 GB/s	n/a	n/a	n/a	
memset (0)	431.709 ms	1.92 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	1.99 MB/s	n/a	n/a	n/a	
memset (0)	431.735 ms	1.184 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	3.22 MB/s	n/a	n/a	n/a	
memset (0)	431.756 ms	1.184 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	3.22 MB/s	n/a	n/a	n/a	
memset (0)	431.781 ms	1.12 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	3.41 MB/s	n/a	n/a	n/a	
elec_GPIL_kernel(curandStateXORWOW*, GXTrack*, GXTrackLias	432.301 ms	13.707 ms	[32,1,1]	[128,1,1]	63	0	0	n/a	n/a	50.79 GB/s	52.2 GB/s	27.4%	
count_by_process_kernel(int, GXTrack*, int*, int*)	446.189 ms	37.18 µs	[32,1,1]	[128,1,1]	14	0	1024	n/a	n/a	3.57 GB/s	114.31 GB/s	53.8%	
Memcpy DtoH [async]	446.228 ms	1.696 µs	n/a	n/a	n/a	n/a	n/a	4 bytes	2.25 MB/s	n/a	n/a	n/a	

Further performance profiling results

- Performance profiles and bottlenecks collected with NVidia's sampling-based profiler NVVP
- Identified performance issues
  - Low multiprocessor occupancy
  - High branch divergence
  - Low memory efficiency
- Each thread sets up its own OO context. Need to increase constant data sharing. Suggestions:
  - Organize tracks by logical volume so that tracks use the same geometry data
  - Each thread may need to run single physics to improve caching

NVidia compiler driver: nvcc

- Release build command:
  - nvcc --arch=sm_20 --optimize 2 --use_fast_math *.cu
- Compilation/build stages:
  - Preprocess: gcc -E *.cu -o *.cpp
  - Separate host and device code: cudafe -o *.c -o *.gpu
  - Codegen Parallel Thread Execution (PTX) code: cicc -o *.ptx
  - Assemble PTX: ptxas -o *.cubin
  - Codegen a multi-arch fatbinary image: fatbinary -o *.fatbin
  - Embed device image into host object file: gcc *.fatbin -o *.o
  - Link with CUDA driver and libraries that malloc on GPU, memcpy CPU to GPU and remote-procedure-call device kernels/functions from the host

Performance opportunity #1

- 3-D coordinates are being stored as struct's of 3 doubles
  - 3*8 bytes = 24 bytes
  - Compiler is generating 8 byte-aligned instructions
  - L1 cache transaction is 128 bytes
  - L2 cache transaction is 32 bytes
  - Alignment must be a power of two: e.g. 32-byte aligned
    - Reduce the number of memory transactions by coalescing 3 8-byte fetches into 1 32-byte fetch
    - Advantages: reduced number of load/store transactions, improved memory bandwidth
  - Evaluate using structure-of-arrays instead of array-of-structures
- Code example…

Performance opportunity #1

Source code:

```
GPThreeVector GPThreeVector_create( double x, double y, double z ) {
  GPThreeVector v = {x,y,z};
  return v;
}
```

PTX assembler verbose output:

```
ptxas info    : Function properties for GPThreeVector_create
    48 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
```

Performance opportunity #1

PTX code:

```
.visible .func  (.param .align 8 .b8 func_retval0[24]) GPThreeVector_create(
        .param .b64 GPThreeVector_create_param_0, // 8-bit-element 8-byte-aligned array
        .param .b64 GPThreeVector_create_param_1, // instead of
        .param .b64 GPThreeVector_create_param_2 ) // 8-bit-element 32-byte-aligned array
{
        .reg .f64       %fd<4>;
        ld.param.f64    %fd1, [GPThreeVector_create_param_0];
        ld.param.f64    %fd2, [GPThreeVector_create_param_1];
        ld.param.f64    %fd3, [GPThreeVector_create_param_2];
        st.param.f64    [func_retval0+0], %fd1;
        st.param.f64    [func_retval0+8], %fd2;
        st.param.f64    [func_retval0+16], %fd3;
        .loc 5 10 1
        ret;
}
```

Performance opportunity #2

- Vector arithmetic
  - PTX ISA v.3.1 doesn't provide vector arithmetic on doubles, only floats
  - nvcc compiler is not generating vector arithmetic with two sequential floats
    - vadd2.u32.u32.u32 dest.lo src1.lo src2.lo
    - vadd2.s32.s32.s32 dest.hi src1.hi src2.hi
  - Need to inline PTX assembly vector instructions into CUDA sources
- Vector prototype needs to take advantage of vector arithmetic on 3-D coordinates
- Reduce spills from registers to local memory accesses (>100 cycles)
- Code example…

Performance opportunity #2

Source code (with changes):

```
GPThreeVector GPThreeVector_add( GPThreeVector a, GPThreeVector b ) {
-  return GPThreeVector_create( a.x+b.x, a.y+b.y, a.z+b.z );
+  GPThreeVector v = {a.x+b.x, a.y+b.y, a.z+b.z};
+  return v;
 }
```

PTX assembler output:

Before the change:

```
ptxas info    : Function properties for GPThreeVector_add
   80 bytes stack frame, 8 bytes spill stores, 8 bytes spill loads
```

After the change:

```
ptxas info    : Function properties for GPThreeVector_add
   96 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
```

Spills from registers to thread-local memory lower memory bandwidth

Performance opportunity #2

PTX code:

```
.visible .func  (.param .align 8 .b8 func_retval0[24]) GPThreeVector_add(
    .param .align 8 .b8 GPThreeVector_add_param_0[24], // 8-bit-element 8-byte-aligned array
    .param .align 8 .b8 GPThreeVector_add_param_1[24] ) {
    .reg .f64      %fd<10>;
    ld.param.f64   %fd1, [GPThreeVector_add_param_0+16];
    ld.param.f64   %fd2, [GPThreeVector_add_param_0+8];
    ld.param.f64   %fd3, [GPThreeVector_add_param_0];
    ld.param.f64   %fd4, [GPThreeVector_add_param_1+16];
    ld.param.f64   %fd5, [GPThreeVector_add_param_1+8];
    ld.param.f64   %fd6, [GPThreeVector_add_param_1];
    .loc 5 60 1
    add.f64        %fd7, %fd3, %fd6; // adds 64-bit floats: %fd7 = %fd3 + %fd6
    add.f64        %fd8, %fd2, %fd5; // three separate scalar instructions
    add.f64        %fd9, %fd1, %fd4; // instead of a vector instruction
    st.param.f64   [func_retval0+0], %fd7;
    st.param.f64   [func_retval0+8], %fd8;
    st.param.f64   [func_retval0+16], %fd9;
    ret;
}
```

Future work

- Continue code optimizations for better kernel efficiency
  - Current parallelism/occupancy: avg of 17% of the peak
  - Memory utilization: avg 20% of the peak
  - High instruction replay overhead: e.g. up to 78% of all gmem instrs
  - Need to tune thread grid sizes
- Increase common data sharing across threads
- Perform refactoring of algorithms, where possible

Thank you