OPTIMIZING APPLICATION PERFORMANCE
WITH CUDA® PROFILING TOOLS

Swapna Matwankar, April 7, 2016
CUDA PROFILING TOOLS

• NVIDIA® Visual Profiler
  • Standalone (nvvp)
  • Integrated into NVIDIA® Nsight™ Eclipse Edition (nsight)

• nvprof

• NVIDIA® Nsight™ Visual Studio Edition

• Old environment variable based command-line profiler is discontinued from 8.0.

* Android CUDA APK profiling not supported (yet)
3rd PARTY PROFILING TOOLS

TAU Performance System ®

VampirTrace

PAPI CUDA Component

HPC Toolkit
PERFORMANCE OPPORTUNITIES

Application level opportunities

• Overall application performance
  • Overlap CPU and GPU work, identify the bottlenecks (CPU or GPU)
• Overall GPU utilization and efficiency
  • Overlap compute and memory copies
  • Utilize compute and copy engines effectively

Kernel level opportunities

• Use memory bandwidth efficiently
• Use compute resources efficiently
• Hide instruction and memory latency

Iterate
PERFORMANCE OPPORTUNITIES

Application level

API invocation

NVTX markers and ranges

GPU activities
PERFORMANCE OPPORTUNITIES

Kernel level

Guided Analysis

---

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel “Compute Bound Fp32” is most likely limited by compute.

Perform Compute Analysis

The most likely bottleneck to performance for this kernel is compute, so you should first perform compute analysis to determine how it is limiting performance.

Perform Latency Analysis

Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform these analyses.

Perform Memory Bandwidth Analysis

If you modify the kernel, you need to rerun your application to update this analysis.

---

Unguided Analysis
What’s new in 8.0?

- Dependency Analysis
- NVLink Analysis
- Unified memory profiling
- Instruction Level Profiling (PC sampling)
- Combined source-assemble view
- FP16 Analysis
- OpenAcc on Timeline
- CPU profiling
- Nvidia Tools Extension V2

Features listed in green are Pascal specific features
DEPENDENCY ANALYSIS
DEPENDENCY ANALYSIS

Motivation

Not always

- GPU kernels are bottleneck in application
- Optimizing kernel taking highest time will give more performance benefits

It is important to identify right bottlenecks in the application to get good ROI
In 8.0, profiling tools supports identifying critical path in the application

- Analyzes CPU threads (POSIX) and GPU activities
- Graph is generated by post-processing execution traces of application (negligible execution overhead)
- Dependencies are defined by CUDA API contract
DEPENDENCY ANALYSIS

Results

• Critical path that includes CUDA APIs, GPU activities, thread activities
• For all CUDA APIs, GPU activities and thread activities
  • Time on Critical Path - Optimizing this will improve overall execution time
  • Waiting time - Reducing waiting time will improve load imbalance
  • Inbound/outbound dependencies - To traverse the issues in both directions
DEPENDENCY ANALYSIS

nvprof

Command: ./nvprof --dependency-analysis --cpu-thread-tracing on ./jacobi_cuda 4096 4096 0.005

Output:

<table>
<thead>
<tr>
<th>Critical path(%)</th>
<th>Critical path (ms)</th>
<th>Waiting time (ns)</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>70.99%</td>
<td>994.326674ms</td>
<td>0ns</td>
<td>&lt;Other&gt;</td>
</tr>
<tr>
<td>13.95%</td>
<td>195.570089ms</td>
<td>0ns</td>
<td>cudaMalloc</td>
</tr>
<tr>
<td>5.69%</td>
<td>79.785085ms</td>
<td>0ns</td>
<td>jacobi_kernel(float const <em>, float</em>, int, int, float*)</td>
</tr>
<tr>
<td>4.71%</td>
<td>66.064614ms</td>
<td>0ns</td>
<td>copy_kernel(float*, float const *, int, int)</td>
</tr>
<tr>
<td>2.87%</td>
<td>40.197672ms</td>
<td>19.969822ms</td>
<td>cudaMemcpy</td>
</tr>
<tr>
<td>1.42%</td>
<td>19.969622ms</td>
<td>0ns</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
</tbody>
</table>

Note: --cpu-thread-tracing on option is required only for multithreaded applications
DEPENDENCY ANALYSIS

Use --print-dependency-analysis-trace argument along with --dependency-analysis to get the time on critical path and waiting time of each instance of a function

Command: ./nvprof --print-dependency-analysis-trace --dependency-analysis --cpu-thread-tracing on ./jacobi_cuda 4096 4096 0.005
DEPENDENCY ANALYSIS

Visual Profiler: Critical path

Unguided Analysis

Dependency Analysis

Functions on critical path
DEPENDENCY ANALYSIS

Visual Profiler

APIs, GPU activities not in critical path are greyed out
DEPENDENCY ANALYSIS

Visual Profiler

Launch jacobi_kernel
MemCpy HtoD [sync]
MemCpyDtoH [sync]

Inbound dependencies
Outbound dependencies
DEPENDENCY ANALYSIS
Visual Profiler

GPU kernel properties

Property view

API properties
DEPENDENCY ANALYSIS

Example: Step 1

Iterative execution pattern: 1. compute GPU+CPU 2. copy GPU+CPU
DEPENDENCY ANALYSIS

Example: Step 1

Guided analysis: Optimize jacobi_kernel

Kernel duration 6% of total session duration, kernel optimization may not impact application performance
Dependency analysis feature points that ‘Other’ CPU accounts for 73% of critical path
## DEPENDENCY ANALYSIS

### Example: Step 1

The following table shows metrics collected from a dependency analysis of the program execution. The data is summarized per function type. Use the "Dependency Analysis" menu on the main toolbar to visualize analysis results on the timeline.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Time on Critical Path (%)</th>
<th>Time on Critical Path</th>
<th>Waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaDeviceSynchronize</td>
<td>0.00 %</td>
<td>5.952 µs</td>
<td>64.056 ms</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>3.15 %</td>
<td>40.154 ms</td>
<td>19.273 ms</td>
</tr>
<tr>
<td>pthread_enter</td>
<td>0.00 %</td>
<td>0 ns</td>
<td>0 ns</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>0.00 %</td>
<td>0 ns</td>
<td>0 ns</td>
</tr>
<tr>
<td>cuDeviceGetCount</td>
<td>0.00 %</td>
<td>3.345 µs</td>
<td>0 ns</td>
</tr>
<tr>
<td>cuDeviceGet</td>
<td>0.00 %</td>
<td>926 ns</td>
<td>0 ns</td>
</tr>
<tr>
<td>cuDeviceGetAttribute</td>
<td>0.03 %</td>
<td>234.525 µs</td>
<td>0 ns</td>
</tr>
<tr>
<td>cuDeviceGetName</td>
<td>0.00 %</td>
<td>17.665 µs</td>
<td>0 ns</td>
</tr>
<tr>
<td>cuDeviceTotalMem_v2</td>
<td>0.01 %</td>
<td>182.71 µs</td>
<td>0 ns</td>
</tr>
<tr>
<td>cudaGetDeviceProperties</td>
<td>0.03 %</td>
<td>322.93 µs</td>
<td>0 ns</td>
</tr>
</tbody>
</table>

Critical path sorted by waiting time
DEPENDENCY ANALYSIS

Example: Step 1

cudaMemcpy waiting for jacobi_kernel to finish
DEPENDENCY ANALYSIS

Example: Step 1

cudaDeviceSynchronize waiting for copy_kernel to finish
DEPENDENCY ANALYSIS

Sample code

Step 1 code

```c
jacobi_kernel<<< ... >>> (...);
cudaMemcpy(...);

compute_cpu

copy_kernel<<< ... >>> (...);
cudaDeviceSynchronize(...);

copy_cpu
```

Step 2 code

```c
jacobi_kernel<<< ... >>> (...);

compute_cpu

cudaMemcpy(...);
copy_kernel<<< ... >>> (...);

copy_cpu
cudaDeviceSynchronize(...);
```
DEPENDENCY ANALYSIS

Example: Step 2

CPU and GPU activities are overlapped
DEPENDENCY ANALYSIS

Example: Step 2

GPU kernels are no more in critical path

Session time is reduced from 1.3s to 1.15s due to overlap but kernel time is still very less compared to session time
Offload more work on GPU activity by changing CPU compute ratio from 5% to 0.5%
DEPENDENCY ANALYSIS

GPU kernels are on critical path. Time to optimize GPU kernels!

Session time is reduced significantly. 2.7X performance improvement without changing kernel.
DEPENDENCY ANALYSIS

Limitations

• Doesn’t take into account wait states caused by CPU synchronization methods

• Doesn’t account for synchronization done by polling memory location that will be updated by GPU activity

• Doesn’t include synchronization caused by resource contention

• Limited support for dynamic parallelism - No dependency tracking for device launched kernels
NVLINK ANALYSIS
NVIDIA NVLINK HIGH-SPEED INTERCONNECT

• High-bandwidth, energy-efficient interconnect

• Enables ultra-fast communication between the CPU and GPU, and between GPUs

• Allows data sharing at rates 5 to 12 times faster than the traditional PCIe Gen3 interconnect
NVLINK ANALYSIS

Topology

CPU (NVLink enabled)

Pascal

Pascal

PCIe Switch

Physical NVLink

Logical NVLink
Gang of physical NVLinks between A & B

Port

#
NVLINK ANALYSIS

nvprof

• nvprof supports a new event collection mode “continuous”
• Supported only on Tesla GPUs
• Collects event samples every 2ms (fixed period for now)
• Metrics are collected at device level
  • Example: `./nvprof --aggregate-mode off --event-collection-mode continuous -metrics nvlink_total_data_transmitted,nvlink_total_data_received,nvlink_transmit_throughput,nvlink_receive_throughput -f -o memcpy.out ./memcpy`
• To get detailed output i.e metric value along with timestamp for each sample use `--print-gpu-trace`
NVLINK ANALYSIS

nvprof

• Nvprof also gives topology information

• Example: nvprof --print-nvlink-topology ./app_name

• Output:

Graphics Device 1 port 0, 1, CPU, Nvlink Bandwidth 40.00GB/s, Physical Links 2, Sysmem Access True, Sysmem Atomic Access False, Peer Access False, Peer Atomic Access False
Graphics Device 0 port 2, 3, CPU, Nvlink Bandwidth 40.00GB/s, Physical Links 2, Sysmem Access True, Sysmem Atomic Access False, Peer Access False, Peer Atomic Access False
Graphics Device 0 port 0, 1, Graphics Device 1 port 3, 2, Nvlink Bandwidth 40.00GB/s, Physical Links 2, Sysmem Access False, Sysmem Atomic Access False, Peer Access True, Peer Atomic Access True

• NVLink metrics have to be correlated by matching port number in topology record with instance number of metric
NVLINK ANALYSIS

Visual Profiler

Bidirectional memory transfers between CPU and GPU0
NVLINK ANALYSIS

Visual Profiler

Unguided Analysis

Option to collect NVLink information

Topology

Achieved throughput

Static properties

Runtime values
NVLINK ANALYSIS

Visual Profiler

Memcpy P2P

Kernel accessing peer memory
NVLINK ANALYSIS

Visual Profiler

The following NVLink topology diagram shows logical NVLink connections between GPUs and CPU. The logical NVLink can contain one or more physical links. For a logical NVLink between two devices A and B, the average receive throughput for device B is shown close to device B. Receive throughput of device A is shown as transmit throughput of device B.

The tables on the right-hand side show the properties, bandwidth utilization, and throughput of each logical NVLink.

<table>
<thead>
<tr>
<th>Logical NVLink Properties</th>
<th>Physical</th>
<th>PeerAccess</th>
<th>SystemAccess</th>
<th>PeerAtomic</th>
<th>SystemAtomic</th>
<th>Utilization %</th>
<th>Idle time %</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU0&lt;----&gt;CPU</td>
<td>40 GB/s</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>43</td>
<td>64</td>
</tr>
<tr>
<td>GPU0&lt;----&gt;GPU1</td>
<td>40 GB/s</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>GPU1&lt;----&gt;CPU</td>
<td>40 GB/s</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>34</td>
<td>74</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical NVLink Throughput</th>
<th>Avg Throughput</th>
<th>Max Throughput</th>
<th>Min Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU0&lt;----&gt;GPU1</td>
<td>10.3 GB/s</td>
<td>37.881 GB/s</td>
<td>7.808 kB/s</td>
</tr>
<tr>
<td>GPU0&lt;----&gt;GPU1</td>
<td>7.033 GB/s</td>
<td>38.001 GB/s</td>
<td>11.712 kB/s</td>
</tr>
<tr>
<td>GPU0&lt;----&gt;GPU1</td>
<td>13.555 GB/s</td>
<td>75.76 GB/s</td>
<td>15.212 kB/s</td>
</tr>
<tr>
<td>GPU0&lt;----&gt;GPU1</td>
<td>13.968 GB/s</td>
<td>76.002 GB/s</td>
<td>10.34 kB/s</td>
</tr>
<tr>
<td>GPU1&lt;----&gt;GPU1</td>
<td>11.79 GB/s</td>
<td>66.441 GB/s</td>
<td>23.04 kB/s</td>
</tr>
<tr>
<td>GPU1&lt;----&gt;GPU1</td>
<td>2.19 GB/s</td>
<td>11.72 GB/s</td>
<td>7.688 kB/s</td>
</tr>
</tbody>
</table>
UNIFIED MEMORY
UNIFIED MEMORY
Starting with Kepler and CUDA 6

Custom Data Management

System Memory

GPU Memory

Developer View With Unified Memory

Unified Memory
UNIFIED MEMORY

• Single allocation, single pointer accessible everywhere
• Pascal GPUs support demand paging
  • Pages populated and data migrated on first touch, overhead of transferring entire allocation is eliminated
  • Concurrent access to memory from CPU and GPU
  • Enables applications with large data models by allowing to oversubscribe GPU memory by spilling over to CPU memory
  • Can access OS controlled memory on supporting system
UNIFIED MEMORY

CUDA 6.0+ code

```c
void sortfile(FILE *fp, int N) {
    char *data;
    cudaMallocManaged(&data, N);
    fread(data, 1, N, fp);
    qsort<<<...>>>(data, N, 1, compare);
    cudaDeviceSynchronize();
    use_data(data);
    cudaFree(data);
}
```

CUDA 8.0 Code *

```c
void sortfile(FILE *fp, int N) {
    char *data;
    cudaMallocManaged(&data, N);
    fread(data, 1, N, fp);
    qsort<<<...>>>(data, N, 1, compare);
    cudaDeviceSynchronize();
    use_data(data);
    cudaFree(data);
}
```

- **Pages allocated in GPU memory**
- **CPU page fault, data migrates to CPU**
- **Kernel launch, data migrates to GPU**
- **GPU page fault, data migrates to GPU**
- **Empty, no pages anywhere**
- **CPU page fault, data allocates on CPU**
- **GPU page fault, data allocates on CPU**
UNIFIED MEMORY

Visual profiler - 6.0+ unified memory
UNIFIED MEMORY
Visual profiler - 8.0 unified memory timeline

Work in progress mockup slides
UNIFIED MEMORY
Visual profiler – Properties of faults and migrations

<table>
<thead>
<tr>
<th>CPU Fault</th>
<th></th>
<th>GPU Fault</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault generated</td>
<td>19.250ms</td>
<td>Fault generated</td>
<td>19.236ms</td>
</tr>
<tr>
<td>Fault resolved</td>
<td>19.251ms</td>
<td>Fault resolved</td>
<td>19.240ms</td>
</tr>
<tr>
<td>Page address</td>
<td>0x23456000</td>
<td>Page address</td>
<td>0x23456000</td>
</tr>
<tr>
<td>Page size</td>
<td>4KB</td>
<td>Page size</td>
<td>4KB</td>
</tr>
<tr>
<td>Access type</td>
<td>Read</td>
<td>Access type</td>
<td>RW</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Migration (DtoH)</th>
<th></th>
<th>Data Migration (HtoD)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>19.250 ms</td>
<td>Start</td>
<td>19.237 ms</td>
</tr>
<tr>
<td>End</td>
<td>19.251 ms</td>
<td>End</td>
<td>19.239 ms</td>
</tr>
<tr>
<td>Duration</td>
<td>1us</td>
<td>Duration</td>
<td>2us</td>
</tr>
<tr>
<td>Start address</td>
<td>0x23456000</td>
<td>Start address</td>
<td>0x23456000</td>
</tr>
<tr>
<td>Size</td>
<td>4KB</td>
<td>Size</td>
<td>8KB</td>
</tr>
<tr>
<td>Process</td>
<td>16763</td>
<td>Process</td>
<td>16763</td>
</tr>
</tbody>
</table>
UNIFIED MEMORY
Visual profiler - Fault-migration correlation

Read dataset1

Use dataset 1

Hto

D D D D

Hto

C C C C

GPU Pag

GPU Pag

GPU P

GPU P

qsort

qsort

Marker and ranges
Unified Memory
Data Migration DtoH
Data Migration HtoD
CPU page fault
GPU page fault
qsort

Work in progress mockup slides
## Unified Memory

**Visual profiler - Fault-migration correlation**

### Marker and ranges
- **Read dataset 1**
- **Use dataset 1**

<table>
<thead>
<tr>
<th>Marker and ranges</th>
<th>Unified Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Migration DtoH</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Data Migration HtoD</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CPU page fault</strong></td>
<td></td>
</tr>
<tr>
<td><strong>GPU page fault</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>qsort</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPU Pag</th>
<th>GPU Pag</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU P</td>
<td>GPU P</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hto</th>
<th>Hto</th>
</tr>
</thead>
<tbody>
<tr>
<td>D D D D</td>
<td>C C C C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPU Pag</th>
</tr>
</thead>
<tbody>
<tr>
<td>qsort</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>qsort</th>
</tr>
</thead>
<tbody>
<tr>
<td>qsort</td>
</tr>
</tbody>
</table>
Manually map the GPU page faults to kernels and CPU page faults to NVTX annotated regions on timeline.
## UNIFIED MEMORY

### Visual profiler - Correlating fault to source

<table>
<thead>
<tr>
<th>Marker and ranges</th>
<th>Read dataset1</th>
<th>Use dataset 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unified Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data MigrationDtoH</td>
<td>HtoD D</td>
<td>DtoH D</td>
</tr>
<tr>
<td>Data MigrationHtoD</td>
<td>HtoD D</td>
<td></td>
</tr>
<tr>
<td>CPU page fault</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU page fault</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KernelA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KernelB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KernelC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Use VA range of allocations used in kernels to correlate with page address from corresponding page fault

Work in progress mockup slides
INSTRUCTION LEVEL PROFILING
(PC SAMPLING)
PC SAMPLING

- PC sampling feature is introduced in 7.5, available for CC >= 5.2
- Provides CPU PC sampling parity + additional information for warp states/stalls reasons for GPU kernels
- Effective in optimizing large kernels, pinpoints performance bottlenecks at specific lines in source code or assembly instructions
- Maxwell architecture gives overall view of scheduling in GPU
  - Samples warp states periodically in round robin order over all active warps
  - Sampling rate is fixed in visual profiler for a GPU
  - No overheads in kernel runtime, CPU overheads to parse the records
## PC SAMPLING ALGORITHM

<table>
<thead>
<tr>
<th>Time in cycles</th>
<th>Warp scheduler 0</th>
<th>Warp scheduler 1</th>
<th>Warp scheduler 2</th>
<th>Warp scheduler 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>w0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>w1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
<td></td>
<td>w2</td>
<td></td>
</tr>
<tr>
<td>768</td>
<td></td>
<td></td>
<td></td>
<td>w3</td>
</tr>
<tr>
<td>1024</td>
<td>w4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1280</td>
<td></td>
<td>w5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1536</td>
<td></td>
<td></td>
<td>w6</td>
<td></td>
</tr>
<tr>
<td>1792</td>
<td></td>
<td></td>
<td></td>
<td>w7</td>
</tr>
</tbody>
</table>
Iterative Closest Point algorithm

Primary stall reasons:

- Memory dependency:
  - LDL (“load local”) instructions.
  - Not because of register spilling
  - Local memory is used for local variables with indexed access
- Synchronization stalls
  - BAR.SYNC barrier instruction i.e. __syncthreads()
PC SAMPLING EXAMPLE

Original Code

```c
float row[7]  
//Initialize array row
int shift = 0;
__shared__ float smem[CTA_SIZE];
for (int i = 0; i < 6; ++i) { // rows
  #pragma unroll
  for (int j = i; j < 7; ++j) { // cols + b
    __syncthreads();  // sync
    smem[tid] = row[i] * row[j]; // local load
    __syncthreads();
    reduce(smem);
    if (tid == 0)
      gbuf.ptr(shift++)[blockIdx.x + blockDim.x * blockIdx.y] = smem[0];
  }
}
```

New Code (LDL removed)

```c
float row0, row1, row2, row3, row4, row5, row6;
//Initialize all elements
#define UNROLL_REDUCE(val, buf) 
  do {
    smem[tid] = val;
    __syncthreads();
    reduce(smem);
    if (tid == 0)
      buf.ptr(shift++)[blockIdx.x + blockDim.x * blockIdx.y] = smem[0];
  } while(0)

UNROLL_REDUCE(row0*row0, gbuf);
UNROLL_REDUCE(row0*row1, gbuf);
UNROLL_REDUCE(row0*row2, gbuf);
UNROLL_REDUCE(row0*row3, gbuf);
UNROLL_REDUCE(row0*row4, gbuf);

Perf: 1.6x (2.3ms vs 3.9ms)
COMBINED SOURCE LEVEL ANALYSIS
COMBINED SOURCE LEVEL ANALYSIS

Visual profiler

All the source level analysis are combined in the same view

- Global access
- Shared access
- Divergent branch
- Instruction level execution
- PC sampling
- Register pressure

Easy analysis, can pinpoint issues for stalls in some cases
COMBINED SOURCE LEVEL ANALYSIS

Add/hide source level analysis
Selects hotspot when multiple analyses are enabled

Shared memory load/store bank conflicts cause execution dependency and memory throttle stalls
COMBINED SOURCE LEVEL ANALYSIS

Register pressure

Hotspot only for assembly

Register is the limiting factor for occupancy
COMPUTE PREEMPTION
COMPUTE PREEMPTION

Pascal architecture introduces a new feature compute to give fair chance for all compute contexts while running long tasks.

How it affects profiling results?

• If multiple contexts are running in parallel it is possible that long kernels will get preempted.
• Some kernels may get preempted occasionally due to timeslice expiry for the context
• In CUDA 8.0, if kernel has been preempted mid execution, the time the kernel spends preempted is still counted towards kernel duration
• This can affect the kernel optimization priorities given by visual profiler as there is randomness introduced due to preemption
COMPUTE PREEMPTION

Kernel taking long time due to compute preemption
COMPUTE PREEMPTION
How to get accurate results?

• Run only one context at a time
  • use as secondary GPU
  • unload display driver in linux
  • run only one process (that uses GPU) at one time
FP16 ANALYSIS
FP16 ANALYSIS

- FP16 (half precision) support added in CC 5.3 and 6.0 (Pascal architecture)
- Stores up to 2x larger models in GPU memory.
- Reduce memory bandwidth requirements by up to 2x.
- Profiler gives the instruction counts, half precision function unit utilization and floating point operations count to analyze performance of fp16
FP16 ANALYSIS

Function unit utilization

Instruction Mix

FLOP count
NVIDIA TOOLS EXTENSION (NVTX) V2
NVTX V2

- NVTX is used for annotating events, code ranges, resources
- Multiple middleware annotating using same strings cause collision
- NVTX V2 introduces domain concept, each middleware can use its own domain
  - Now middleware and your application don’t need to collide
  - Visual profiler shows markers/ranges of each domain on separate timeline
- Synchronization primitives can also be named
  - Tools can track and present why you are blocked with a custom message
Module A

eventAttrib.message.ascii = "Range1";
nvtxRangeId_t idx0 = nvtxRangeStartEx(&eventAttrib);
//CPU code
nvtxRangeEnd(idx0);

Module B

eventAttrib.message.ascii = "Range1";
nvtxRangeId_t idx1 = nvtxRangeStartEx(&eventAttrib);
//CPU code
nvtxRangeEnd(idx1);

Module A

nvtxDomainHandle_t domain_a = nvtxDomainCreateA("ModuleA");

eventAttrib.message.ascii = "Range1";
nvtxRangeId_t idx0 = nvtxDomainRangeStartEx(domain_a, &eventAttrib);
//CPU code
nvtxDomainRangeEnd(domain_a, idx0);

Module B

nvtxDomainHandle_t domain_b = nvtxDomainCreateA("ModuleB");

eventAttrib.message.ascii = "Range1";
nvtxRangeId_t idx1 = nvtxDomainRangeStartEx(domain_b, &eventAttrib);
//CPU code
nvtxDomainRangeEnd(domain_b, idx1);
Range information is grouped based on range name.
NVTX V2
Visual Profiler

Same range names from different modules

Domain names

From domain 1
From domain 2
## CPU PROFILING

<table>
<thead>
<tr>
<th>Event</th>
<th>%</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bench_staggeredleapfrog2_</td>
<td>95.83%</td>
<td>689.695 ms</td>
</tr>
<tr>
<td>CCTKi_BindingsFortranWrapperBenchADM</td>
<td>95.83%</td>
<td>689.695 ms</td>
</tr>
<tr>
<td>CCTK_CallFunction</td>
<td>95.83%</td>
<td>689.695 ms</td>
</tr>
<tr>
<td>__open_nocancel</td>
<td>1.389%</td>
<td>9.996 ms</td>
</tr>
<tr>
<td>InitialFlat</td>
<td>1.389%</td>
<td>9.996 ms</td>
</tr>
<tr>
<td>__c_mcopy8</td>
<td>1.389%</td>
<td>9.996 ms</td>
</tr>
</tbody>
</table>

**Multiple markers at this line**

- Generated 6 prefetch instructions for the loop
- Generated vector sse code for the loop
- Generated 5 alternate versions of the loop
- 2 loop-carried redundant expressions removed with 2 operations and 4 arrays
- Intensity = 1.93

100 CONTINUE
OPENACC PROFILING
OPENACC PROFILING

OpenAcc->Driver API->Compute correlation

OpenAcc timeline

OpenAcc->Source Code correlation

OpenAcc Properties
OTHER PRESENTATIONS

CUDA 8.0 features:
• S6224 - Featured Presentation: CUDA 8 and Beyond

Unified memory:
• S6216 - The Future of Unified Memory
• S6134 - High Performance and Productivity with Unified Memory and OpenACC: A LBM Case Study

Tools presentations:
• S6615 - Developer Tools Arsenal for Tegra Platforms
• S6784 - Maximize OpenACC Performance with the PGPROF Profiler
• S6531 - CUDA® Debugging Tools in CUDA 8
• S6111 - NVIDIA CUDA® Optimization with NVIDIA Nsight™ Eclipse Edition: A Case Study
• S6112 - NVIDIA CUDA® Optimization with NVIDIA Nsight™ Visual Studio Edition: A Case Study
REFERENCES

NVIDIA toolkit documentation:
• http://docs.nvidia.com/

Pascal architecture:
• https://devblogs.nvidia.com/parallelforall/inside-pascal/

PC sampling blog:
THANK YOU

JOIN THE CONVERSATION
#GTC16 -twitter-facebook-linkedin

JOIN THE NVIDIA DEVELOPER PROGRAM AT developer.nvidia.com/join