CUDA PROFILING TOOLS
GRAPHICAL AND COMMAND-LINE 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 still available

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

- TAU Performance System ®
- VampirTrace
- PAPI CUDA Component
- HPC Toolkit
CUDA PROFILING TOOLS INTERFACE (CUPTI)

- Enables creation of profiling and tracing tools that target CUDA applications
- Designed for tools vendors, not CUDA developers
PERFORMANCE OPPORTUNITIES

- Application level opportunities
  - Overall GPU utilization and efficiency
  - Memory copy efficiency

- Kernel level opportunities
  - Instruction and memory latency hiding/reduction
  - Efficient use of memory bandwidth
  - Efficient use of compute resources

Iterate
VISUAL PROFILER

GPU/CPU Timeline
VISUAL PROFILER

Unguided Analysis

Kernel Latency
- Kernel Compute
- Kernel Memory
- Global Memory Access Pattern
- Shared Memory Access Pattern
- Divergent Execution
- Kernel Profile

Application
- Data Movement And Concurrency
- Compute Utilization
- Kernel Performance
VISUAL PROFILER

Guided Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

4. Compute Resources

GPU compute resources limit the performance of a kernel when those resources are insufficient or poorly utilized.

Show Kernel Profile

The kernel profile shows the execution count, inactive threads, and predicated threads for each source and assembly line of the kernel. Using this information you can pinpoint portions of your kernel that are making inefficient use of compute resource due to divergence and predication.

Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.
## VISUAL PROFILER: APPLICATION LEVEL OPPORTUNITIES

<table>
<thead>
<tr>
<th>Results</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low Compute / Mempcy Efficiency</strong> [17.639 ms / 26.429 ms = 0.667]</td>
<td>The amount of time performing compute is low relative to the amount of time required for mempcy. More...</td>
</tr>
<tr>
<td><strong>Low Mempcy/Compute Overlap</strong> [0 ns / 17.639 ms = 0%]</td>
<td>The percentage of time when mempcy is being performed in parallel with compute is low. More...</td>
</tr>
<tr>
<td><strong>Low Kernel Concurrency</strong> [0 ns / 17.639 ms = 0%]</td>
<td>The percentage of time when two kernels are being executed in parallel is low. More...</td>
</tr>
<tr>
<td><strong>Low Mempcy Throughput</strong> [2.296 GB/s avg, for mempcys accounting for 82.9% of all mempcy time]</td>
<td>The memory copies are not fully using the available host to device bandwidth. More...</td>
</tr>
<tr>
<td><strong>Low Mempcy Overlap</strong> [0 ns / 12.211 ms = 0%]</td>
<td>The percentage of time when two memory copies are being performed in parallel is low. More...</td>
</tr>
<tr>
<td><strong>Low Compute Utilization</strong> [17.639 ms / 484.212 ms = 3.6%]</td>
<td>The multiprocessors of one or more GPUs are mostly idle. More...</td>
</tr>
<tr>
<td><strong>Compute Utilization</strong></td>
<td>The device timeline shows an estimate of the amount of the total compute capacity being used by the kernels executing on the device. More...</td>
</tr>
</tbody>
</table>

You can also examine the performance of individual kernels to expose additional optimization opportunities.

---

**Profiler Suggestions**
VISUAL PROFILER: KERNEL LEVEL OPPORTUNITIES

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels can significantly improve performance compared to lower ranked kernels.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>void mergeElementaryIntervalsKernel(unsigned int, unsigned int*, unsigned int*, unsigned int*, unsigned int*)</td>
</tr>
<tr>
<td>25</td>
<td>void mergeSortSharedKernel(unsigned int, unsigned int*, unsigned int*, unsigned int*, unsigned int*)</td>
</tr>
<tr>
<td>15</td>
<td>void generateSampleRanksKernel(unsigned int*, unsigned int*, unsigned int*, unsigned int*, unsigned int*, unsigned int*)</td>
</tr>
<tr>
<td>10</td>
<td>mergeRanksAndIndicesKernel(unsigned int*, unsigned int*, unsigned int*, unsigned int*, unsigned int*, unsigned int*)</td>
</tr>
<tr>
<td>1</td>
<td>mergeRanksAndIndicesKernel(unsigned int*, unsigned int*, unsigned int*, unsigned int*, unsigned int*, unsigned int*)</td>
</tr>
<tr>
<td>1</td>
<td>mergeRanksAndIndicesKernel(unsigned int*, unsigned int*, unsigned int*, unsigned int*, unsigned int*, unsigned int*)</td>
</tr>
<tr>
<td>1</td>
<td>mergeRanksAndIndicesKernel(unsigned int*, unsigned int*, unsigned int*, unsigned int*, unsigned int*, unsigned int*)</td>
</tr>
</tbody>
</table>
WHAT’S NEW IN 6.0?

- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Shared Memory Source Level Analysis
- Mix of Instructions for a Kernel
- Inefficient SM Utilization Detection
- Remote Profiling
WHAT’S NEW IN 6.0?

- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Shared Memory Source Level Analysis
- Mix of Instructions for a Kernel
- Inefficient SM Utilization Detection
- Remote Profiling
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);
}
NVPROF: UNIFIED MEMORY

• Use **--unified-memory-profiling** to collect Unified Memory profiling data.

```
$ nvprof --unified-memory-profiling per-process-device UnifiedMemoryStreams
...
==25394== Profiling application: UnifiedMemoryStreams
==25394== Profiling result:
==25394== Unified Memory profiling result:
Device "Quadro K6000 (0)"
```

<table>
<thead>
<tr>
<th>Count</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host To Device (bytes)</td>
<td>13</td>
<td>5.2297e+07</td>
<td>0</td>
</tr>
<tr>
<td>Device To Host (bytes)</td>
<td>140</td>
<td>2.5555e+07</td>
<td>0</td>
</tr>
<tr>
<td>CPU Page faults</td>
<td>140</td>
<td>13728.75</td>
<td>0</td>
</tr>
</tbody>
</table>

• Use **--print-gpu-trace** for detailed information
$ nvprof --unified-memory-profiling per-process-device --print-gpu-trace UnifiedMemoryStreams
...
==25394== Profiling application: UnifiedMemoryStreams
==25394== Profiling result:

   Start ... Unified Memory Name
  348.87ms ... 0 B [Unified Memory MemcpyDtoH]
  348.87ms ... 0 [Unified Memory CPU page faults]
  350.94ms ... 610304 B [Unified Memory MemcpyDtoH]
  350.94ms ... 149 [Unified Memory CPU page faults]
  ...
  608.23ms ... 0 B [Unified Memory MemcpyHtoD]
  610.30ms ... 6320128 B [Unified Memory MemcpyHtoD]
  ...
  632.71ms ... - void gemv2N_kernel_val<double, int=128, int=8, int=4, int=4> ...
  633.00ms ... 24876 [Unified Memory CPU page faults]
  633.00ms ... 101892096 B [Unified Memory MemcpyDtoH]
$ nvprof --unified-memory-profiling per-process-device --print-gpu-trace ./UnifiedMemoryStreams

==25394== Profiling application: UnifiedMemoryStreams
==25394== Profiling result:

<table>
<thead>
<tr>
<th>Start</th>
<th>Unified Memory Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>348.87ms</td>
<td>0 B [Unified Memory MemcpyDtoH]</td>
</tr>
<tr>
<td>348.87ms</td>
<td>0 [Unified Memory CPU page faults]</td>
</tr>
<tr>
<td>350.94ms</td>
<td>610304 B [Unified Memory MemcpyDtoH]</td>
</tr>
<tr>
<td>350.94ms</td>
<td>149 [Unified Memory CPU page faults]</td>
</tr>
<tr>
<td>608.23ms</td>
<td>0 B [Unified Memory MemcpyHtoD]</td>
</tr>
<tr>
<td>610.30ms</td>
<td>6320128 B [Unified Memory MemcpyHtoD]</td>
</tr>
<tr>
<td>632.71ms</td>
<td>- void gemv2N_kernel_val&lt;double, int=128, int=8, int=4, int=4&gt;</td>
</tr>
<tr>
<td>633.00ms</td>
<td>24876 [Unified Memory CPU page faults]</td>
</tr>
<tr>
<td>633.00ms</td>
<td>101892096 B [Unified Memory MemcpyDtoH]</td>
</tr>
</tbody>
</table>

Copy buffer (HtoD)  

Kernel Launch
$ nvprof --unified-memory-profiling per-process-device --print-gpu-trace ./UnifiedMemoryStreams

==25394== Profiling application: UnifiedMemoryStreams

==25394== Profiling result:

<table>
<thead>
<tr>
<th>Start</th>
<th>Unified Memory</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>348.87ms</td>
<td>0 B</td>
<td>[Unified Memory MemcpyDtoH]</td>
</tr>
<tr>
<td>348.87ms</td>
<td>0</td>
<td>[Unified Memory CPU page faults]</td>
</tr>
<tr>
<td>350.94ms</td>
<td>610304 B</td>
<td>[Unified Memory MemcpyDtoH]</td>
</tr>
<tr>
<td>350.94ms</td>
<td>149</td>
<td>[Unified Memory CPU page faults]</td>
</tr>
<tr>
<td>608.23ms</td>
<td>0 B</td>
<td>[Unified Memory MemcpyHtoD]</td>
</tr>
<tr>
<td>610.30ms</td>
<td>6320128 B</td>
<td>[Unified Memory MemcpyHtoD]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>632.71ms</td>
<td></td>
<td>void gemv2N_kernel_val&lt;double, int=128, int=8, int=4, int=4&gt; ...</td>
</tr>
<tr>
<td>633.00ms</td>
<td>24876</td>
<td>[Unified Memory CPU page faults]</td>
</tr>
<tr>
<td>633.00ms</td>
<td>101892096 B</td>
<td>[Unified Memory MemcpyDtoH]</td>
</tr>
</tbody>
</table>
NVVP: UNIFIED MEMORY

Enable Unified Memory Profiling
NVVP: UNIFIED MEMORY

Unified Memory Properties

Unified Memory Timeline
NVVP: UNIFIED MEMORY

Unified Memory Properties:
- Duration: 690.47 ms (690,469,874 ns)
- Total Bytes (HtoD): 88.183 MB
- Total Bytes (DtoH): 88.408 MB
- Total Page Faults: 21584
NVVP: UNIFIED MEMORY
### Sample Properties

<table>
<thead>
<tr>
<th>Data Migration (DtoH)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample Start</td>
<td>564.255 ms (564,255,466 ns)</td>
</tr>
<tr>
<td>Sample End</td>
<td>566.317 ms (566,317,358 ns)</td>
</tr>
<tr>
<td>Sample Duration</td>
<td>2.062 ms (2,061,892 ns)</td>
</tr>
<tr>
<td>Size</td>
<td>184.32 kB</td>
</tr>
<tr>
<td>Process</td>
<td>16763</td>
</tr>
</tbody>
</table>
WHAT’S NEW IN 6.0?

- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Shared Memory Source Level Analysis
- Mix of Instructions for a Kernel
- Inefficient SM Utilization Detection
- Remote Profiling
MULTI-PROCESS SERVICE (MPS)

OS Process
MPS Client
CUDA Context

OS Process
MPS Client
CUDA Context

MPS Server Process

GPU Memory

GPU
MPS PROFILING WITH NVPROF

**Step 1:** Launch MPS daemon

```
$ nvidia-cuda-mps-control -d
```

**Step 2:** Run nvprof with `--profile-all-processes`

```
$ nvprof --profile-all-processes -o maxConcurrency_%p
```

Profiling all processes launched by user "sanjain"

Type "Ctrl-c" to exit

**Step 3:** Run application in different terminal normally

```
$ maxConcurrency
```

**Step 4:** Exit nvprof by typing Ctrl+c

```
==5844== NVPROF is profiling process 5844, command: maxConcurrency
==5840== NVPROF is profiling process 5840, command: maxConcurrency
...
==5844== Generated result file: /data/sanjain/r6.0/maxConcurrency_5844
==5840== Generated result file: /data/sanjain/r6.0/maxConcurrency_5840
^C========= Exiting nvprof
```
VIEWING MPS TIMELINE IN VISUAL PROFILER

Import Multi-Process nvprof Data
VIEWING MPS TIMELINE IN VISUAL PROFILER

Browse and Select Files
VIEWING MPS TIMELINE IN VISUAL PROFILER
VIEWING MPS TIMELINE IN VISUAL PROFILER
Overall GPU utilization

Per-Process CUDA API Invocation

Color by Process
WHAT’S NEW IN 6.0?

- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Shared Memory Source Level Analysis
- Mix of Instructions for a Kernel
- Inefficient SM Utilization Detection
- Remote Profiling
WHERE CAN I FIND THE NEW OPTION?

Unguided Analysis

Guided Analysis

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound
4. Compute Resources
   
   GPU compute resources limit the performance of a kernel when those resources are insufficient or poorly utilized.

   The kernel profile shows the execution count, inactive threads, and predicated threads for each source and assembly line of the kernel. Using this information you can pinpoint portions of your kernel that are making inefficient use of compute resource due to divergence and predication.

   If you modify the kernel you need to rerun your application to update this analysis.
DETAILED KERNEL PROFILE
VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE

TIP: Don’t forget to compile your application with -lineinfo
DETAILED KERNEL PROFILE
VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE

Kernel Profile

The kernel profile shows the execution count, inactive threads, and predicated threads for each source and assembly line of the kernel. Using this information you can pinpoint portions of your kernel that are making inefficient use of compute resource due to divergence and predication.

Optimization: Select a kernel or source file listed below to view the profile. Examine portions of the kernel that have high execution counts and inactive or predicated threads to identify optimization opportunities.

CUDA Functions:

shfl_intimage_rows(uint4*, uint4*)

Maximum instruction execution count: 4320
Average instruction execution count: 4017

Source files:

/data/sanjain/6.0/NVIDIA_CUDA-6.0_Samples/6_Advanced/shfl_scan/shfl_scan.cu
/data/sanjain/6.0/bin/../include/sm_30_intrinsics.h
/data/sanjain/6.0/NVIDIA_CUDA-6.0_Samples/6_Advanced/shfl_scan/shfl_integral_image.cuh
DETAILED KERNEL PROFILE
VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE

CUDA Functions:
DETAILED KERNEL PROFILE
VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE

Buttons to Set the View
DETAILED KERNEL PROFILE
VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE

CUDA Source

Corresponding SASS
DETAILED KERNEL PROFILE
VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE

Hot-Spots
Inefficient Execution
WHAT’S NEW IN 6.0?

- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Mix of Instructions for a Kernel
- Shared Memory Source Level Analysis
- Inefficient SM Utilization Detection
- Remote Profiling
WHERE CAN I FIND THE NEW OPTION?
MIX OF INSTRUCTIONS FOR A KERNEL

**Instruction Execution Counts**

The following chart shows the mix of instructions executed by the kernel. The instructions are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing instructions in that class. The "Inactive" result shows the thread executions that did not execute any instruction because the thread was predicated or inactive due to divergence.

**Floating-Point Operation Counts**

The following chart shows the mix of floating-point operations executed by the kernel. The operations are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing operations in that class. The results do not sum to 100% because non-floating-point operations executed by the kernel are not shown in this chart.
WHAT’S NEW IN 6.0?

- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Mix of Instructions for a Kernel
- Shared Memory Source Level Analysis
- Inefficient SM Utilization Detection
- Remote Profiling
No Bank Conflicts

- No Bank Conflicts
**SHARED MEMORY BANK CONFLICT**

- **2-way Bank Conflicts**
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7

- **8-way Bank Conflicts**
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 5
  - Thread 6
  - Thread 7
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7
  - Bank 8
  - Bank 9
  - Bank 31
WHERE CAN I FIND THE NEW OPTION?

Unguided Analysis

Guided Analysis
SHARED MEMORY SOURCE LEVEL ANALYSIS

Bank conflict

Source Line with Inefficient Access

```c
__global__ void transposeCoalesced(float *odata, float *idata, int width, int height)
{

    __shared__ float tile[TILE_DIM][TILE_DIM];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*width;

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*height;

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
    {
        tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
    }

    __syncthreads();

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
    {
        odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
    }

}```
WHAT’S NEW IN 6.0?

- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Mix of Instructions for a Kernel
- Shared Memory Source Level Analysis
- Inefficient SM Utilization Detection
- Remote Profiling
INEFFICIENT SM UTILIZATION

- In theory... kernel allows enough warps on each SM

<table>
<thead>
<tr>
<th>Time</th>
<th>SM0</th>
<th>SM1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Block 0</td>
<td>Block 6</td>
</tr>
<tr>
<td></td>
<td>Block 1</td>
<td>Block 7</td>
</tr>
<tr>
<td></td>
<td>Block 2</td>
<td>Block 8</td>
</tr>
<tr>
<td></td>
<td>Block 3</td>
<td>Block 9</td>
</tr>
<tr>
<td></td>
<td>Block 4</td>
<td>Block 10</td>
</tr>
<tr>
<td></td>
<td>Block 5</td>
<td>Block 11</td>
</tr>
</tbody>
</table>
In theory... kernel launches sufficient warps on each SM
but why is achieved occupancy for kernel low?
Likely cause is that all SMs do not remain equally busy over duration of kernel execution
WHERE CAN I FIND THE NEW OPTION?
PER-SM ACTIVITY

Achieved Occupancy Is Low

Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel’s actual occupancy. The kernel’s achieved occupancy of 14.1% is significantly lower than its theoretical occupancy of 56.2%. Most likely this indicates that there is an imbalance in how the kernel’s blocks are executing on the SMs so that all SMs are not equally busy over the entire execution of the kernel. The following chart shows the utilization of each multiprocessor during execution of the kernel.

Optimization: Make sure that all blocks are doing roughly the same amount of work. It may also help to increase the number of blocks executed by the kernel.

Inefficient SM Utilization
WHAT’S NEW IN 6.0?

- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Mix of Instructions for a Kernel
- Shared Memory Source Level Analysis
- Inefficient SM Utilization Detection
- Remote Profiling
REMOTE PROFILING SCENARIOS

- **Host** - All supported x86 distributions
- **Target** - Can be any supported x86/ARM Linux
- Not necessary for **Host System** to have an NVIDIA GPU
NVVP REMOTE PROFILING SETUP

Manage New Remote Connection
NVVP REMOTE PROFILING SETUP

Enter New Connection Detail

Remote Connections
Manage available connections

- Host name: sanjain-linux64
- User name: sanjain
- Label: sanjain@sanjain-linux64
- System type: SSH
- Port number: 22
NVVP REMOTE PROFILING SETUP

Update Toolkit Path
NVVP REMOTE PROFILING SETUP

- Toolkit path: [blank]
- Library paths: [Add new path]

Browse and Select Toolkit Path
NVVP REMOTE PROFILING SETUP

CUDA Toolkits on sanjain@sanjain-linux64

Configure CUDA Toolkit
Configure CUDA toolkit location on the remote system

Toolkit path: /data/sanjain/r6.0/bin
Library paths:
- /data/sanjain/r6.0/lib64
- /data/sanjain/r6.0/lib
- Add new path

Customize Tools Locations...

Click on Finish
NVVP REMOTE PROFILING SETUP

Executable Properties

Connection: sanjain@sanjain-linux64
Toolkit: /data/sanjain/r6.0/bin
File: Enter executable file [required]
Working directory: Enter working directory [optional]
Arguments: Enter command-line arguments
Environment: Name | Value

Select Executable
NVVP REMOTE PROFILING SETUP

Executable Properties

Connection: sanjain@sanjain-linux64
Toolkit: /data/sanjain/r6.0/bin
File: /data/sanjain/r6.0/NVIDIA_CUDA-6.0_Samples/bin/x86_64/linux
Working directory: Enter working directory [optional]
Arguments: Enter command-line arguments
Environment:

Connect to a remote system

Finish and Continue
NVVP REMOTE PROFILING SETUP
NEXT STEPS

 Download free CUDA Toolkit: www.nvidia.com/getcuda
 Join the community: developer.nvidia.com/join
 Post at Developer Zone Forums: devtalk.nvidia.com
 Visit Hangout Pods and Developer Demo Stations

 S4794 - Optimizing CUDA Application Performance with NVIDIA® Visual Profiler
  - Wednesday, 03/26, 17:00
 S4591 - CUDA Application Development Life Cycle Using NVIDIA® Nsight™, Eclipse Edition
  - Thursday, 03/27, 15:00