Performance Optimization Strategies For GPU-Accelerated Applications
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
Identifying Performance Opportunities

- NVIDIA® Nsight™ Eclipse Edition (nsight)
- NVIDIA® Visual Profiler (nvvp)
- nvprof command-line profiler
Automatic Performance Analysis

- NEW in 5.5 Step-by-step optimization guidance

1. CUDA Application Analysis

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application’s overall GPU usage as well as the performance of individual kernels. There are also a number of profiling and optimization technologies that you can leverage to simplify your optimization efforts.

Examine GPU Utilization

Determine your application’s overall GPU utilization. This analysis requires an application timeline, so your application will be run once to collect it if it is not already available.

Examine Kernel Performance

Determine which kernels are the most performance critical and that have the most opportunity for improvement. This analysis requires utilization data from every kernel, so your application will be run once to collect that data if it is not already available.
Improved Analysis Visualization

- **NEW in 5.5** More clearly present optimization opportunities

---

**Kernel Performance is Bound By Memory**

For device "Tesla C2050" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by memory.

---

![Graphs showing varying block size and register count](image_url)
Command-Line Metric Collection

- NEW in 5.5 Collect metrics using `nvprof`

```bash
$ nvprof --metrics gld_throughput,flops_dp minife -nx 50 -ny 50 -nz 50
...
==24954== Profiling application: minife -nx 50 -ny 50 -nz 50
==24954== Profiling result:
==24954== Metric result:

<table>
<thead>
<tr>
<th>Invocations</th>
<th>Metric Name</th>
<th>Metric Description</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel:</td>
<td>launch_closure_by_value...(double)</td>
<td>gld_throughput</td>
<td>Global Load Throughput</td>
<td>30.385MB/s</td>
<td>32.438MB/s</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>flops_dp</td>
<td>FLOPS(Double)</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>Kernel:</td>
<td>spmv_ell_kernel&lt;double, int&gt;(int*, double*, ...)</td>
<td>gld_throughput</td>
<td>Global Load Throughput</td>
<td>159.07GB/s</td>
<td>159.77GB/s</td>
</tr>
<tr>
<td>51</td>
<td></td>
<td>flops_dp</td>
<td>FLOPS(Double)</td>
<td>6885902</td>
<td>6885902</td>
</tr>
<tr>
<td>Kernel:</td>
<td>add_to_diagonal_kernel...(int, doubleScalarType, ...)</td>
<td>gld_throughput</td>
<td>Global Load Throughput</td>
<td>30.688GB/s</td>
<td>30.688GB/s</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>flops_dp</td>
<td>FLOPS(Double)</td>
<td>132651</td>
<td>132651</td>
</tr>
</tbody>
</table>
```
Optimization Scenarios

- **Full application**
  - Optimize overall GPU utilization, efficiency, etc.
  - Optimize individual kernels, most likely optimization candidates first

- **Algorithm**
  - Using a test harness
  - Optimize individual kernels

Iterate
Outline: Kernel Optimization

- Discuss an overall application optimization strategy
  - Based on identifying primary performance limiter

- Discuss common optimization opportunities

- How to use CUDA profiling tools
  - Execute optimization strategy
  - Identify optimization opportunities
Identifying Candidate Kernels

- Analysis system estimates which kernels are best candidates for speedup
  - Execution time, achieved occupancy

[Image of CUDA Application Analysis and Kernel Optimization Priorities]
Primary Performance Bound

- Most likely limiter to performance for a kernel
  - Memory bandwidth
  - Compute resources
  - Instruction and memory latency

- Primary bound should be addressed first
- Often beneficial to examine secondary bounds as well
Calculating Performance Bounds

- Memory bandwidth utilization
- Compute resource utilization

- High utilization value $\rightarrow$ likely performance limiter
- Low utilization value $\rightarrow$ likely not performance limiter

- Taken together to determine performance bound
Memory Bandwidth Utilization

- Traffic to/from each memory subsystem, relative to peak
- Maximum utilization of any memory subsystem
  - L1/Shared Memory
  - L2 Cache
  - Texture Cache
  - Device Memory
  - System Memory (via PCIe)

- `nvprof` metrics: l2_utilization, texture_utilization, sysmem_utlization, ...
Compute Resource Utilization

- Number of instructions issued, relative to peak capabilities of GPU
  - Some resources shared across all instructions
  - Some resources specific to instruction “classes”: integer, FP, control-flow, etc.
  - Maximum utilization of any resource

- `nvprof` metrics: `issue_slot_utilization`, `int_fu_utilization`, `fp_fu_utilization`, ...
Calculating Performance Bounds

- Utilizations
  - Both high → compute and memory highly utilized

![Utilization Chart](chart.png)
Calculating Performance Bounds

- **Utilizations**
  - Memory high, compute low \(\rightarrow\) memory bandwidth bound

![Graph showing utilizations for compute and memory]
Calculating Performance Bounds

- Utilizations
  - Compute high, memory low → compute resource bound
Calculating Performance Bounds

- Utilizations
  - Both low $\rightarrow$ latency bound
Visual Profiler: Performance Bound

1. CUDA Application Analysis
2. Find Performance-Critical Kernels
3. Compute, Memory, 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, or latency. The results at right indicate that the performance of kernel "MemoryBoundDeviceMemory" is most likely limited by memory.

- Perform Memory Analysis
- Perform Compute Analysis
- Perform Latency Analysis

Kernel Performance Is Bound By Memory

For device "Tesla C2050" the kernel’s compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by memory.

![Graph showing compute and memory utilization](image-url)
Bound-Specific Optimization Analysis

- Memory Bandwidth Bound
- Compute Bound
- Latency Bound
Bound-Specific Optimization Analysis

- Memory Bandwidth Bound
- Compute Bound
- Latency Bound
Memory Bandwidth Bound

```cpp
template<typename Scalar, typename GlobalOrdinal>
__global__ void spmv_ell_kernel(GlobalOrdinal* column_indices,
                                GlobalOrdinal gridsize=blockDim.x*gridDim.x;
for(GlobalOrdinal row_idx=blockIdx.x*blockDim.x+threadIdx.x;
    row_idx<N; row_idx+=gridsize)
{
    Scalar sum=0;
    GlobalOrdinal offset = row_idx;
    for(int j=0;j<cols;++j)
    {
        GlobalOrdinal c=column_indices[offset];
        if(c!=-1) {
            Scalar A=values[offset];
            Scalar x=ldg(X+c);
            sum+=A*x;
        }
        offset+=pitch;
    }
    Y[row_idx]=sum;
}
```

**Kernel Performance is Bound By Memory**

For device “Tesla C2050” the kernel’s compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by memory.
Memory Bandwidths

- Understand memory usage patterns of kernel
  - L1/Shared Memory
  - L2 Cache
  - Texture Cache
  - Device Memory
  - System Memory (via PCIe)

- `nvprof` metrics: `gld_throughput`, `tex_cache_throughput`, ..., `l2_read_transactions`, `sysmem_read_transactions`, ...
<table>
<thead>
<tr>
<th></th>
<th>Transactions</th>
<th>Bandwidth</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1/Shared Memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local Loads</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Local Stores</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Shared Loads</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Shared Stores</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Global Loads</td>
<td>1158709</td>
<td>251.08 GB/s</td>
<td></td>
</tr>
<tr>
<td>Global Stores</td>
<td>8624</td>
<td>1.86 GB/s</td>
<td></td>
</tr>
<tr>
<td>L1/Shared Total</td>
<td>1167333</td>
<td>252.94 GB/s</td>
<td></td>
</tr>
<tr>
<td><strong>Texture Cache</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reads</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reads</td>
<td>1635340</td>
<td>91.75 GB/s</td>
<td></td>
</tr>
<tr>
<td>Writes</td>
<td>33163</td>
<td>1.86 GB/s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>1668503</td>
<td>93.61 GB/s</td>
<td></td>
</tr>
<tr>
<td><strong>Device Memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reads</td>
<td>2065469</td>
<td>115.9 GB/s</td>
<td></td>
</tr>
<tr>
<td>Writes</td>
<td>39472</td>
<td>2.23 GB/s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>2104941</td>
<td>118.13 GB/s</td>
<td></td>
</tr>
<tr>
<td><strong>System Memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reads</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Writes</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
</tbody>
</table>
Memory Access Pattern And Alignment

- Access pattern
  - Sequential: 1 memory access for entire warp
  - Strided: up to 32 memory accesses for warp

- Alignment

- `nvprof` metrics: gld_efficiency, gst_efficiency, shared_efficiency
Visual Profiler: Memory Efficiency

Global Memory Alignment and Access Pattern
Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern.

Optimization: Select each entry below to open the source code to a global load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.

<table>
<thead>
<tr>
<th>Line</th>
<th>SparseMatrix_functions.hpp</th>
<th>Global Load L2 Transactions/Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>552</td>
<td>878884 L2 transactions for 111942 total executions</td>
<td>7.9</td>
</tr>
<tr>
<td>554</td>
<td>1676736 L2 transactions for 109443 total executions</td>
<td>15.3</td>
</tr>
<tr>
<td>555</td>
<td>1895720 L2 transactions for 109443 total executions</td>
<td>17.3</td>
</tr>
</tbody>
</table>

```
for(GlobalOrdinal row_idx=blockIdx.x*blockDim.x+threadIdx.x; row_idx<N; row_idx+=gridsize)
{
    Scalar sum=0;
    GlobalOrdinal offset = row_idx;
    for(int j=0;j<cols;++j)
    {
        GlobalOrdinal c=column_indices[offset];
        if(c!=1) {
            Scalar A=values[offset];
            Scalar x=___lgd(X+c);
            sum+=A*x;
        }
        offset+=pitch;
    }
    Y[row_idx]=sum;
}
```
Bound-Specific Optimization Analysis

- Memory Bandwidth Bound
- Compute Bound
- Latency Bound
Warp Execution Efficiency

- All threads in warp may not execute instruction
  - Divergent branch
  - Divergent predication

- `nvprof` metrics: `warp_execution_efficiency`, `warp_nonpred_execution_efficiency` (SM3.0+)

1 of 32 threads = 3%
32 of 32 threads = 100%
Low Warp Execution Efficiency

Warp execution efficiency is the average percentage of active threads in each executed warp. Increasing warp execution efficiency will increase utilization of the GPU's compute resources. The kernel's warp execution efficiency of 67% is less than 100% due to divergent branches and predicated instructions.

Optimization: Reduce the amount of intra-warp divergence and predication in the kernel.
Visual Profiler: Warp Execution Efficiency

Divergent Branches

Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence.

File: dct8x8_kernel_quantization.cuh  /home/david/depot/davidg-linux-sw/sw/gpgpu/samples/3_imaging/dct8x8  103  Divergence = 100.0%  [8192 divergent executions out of 8192 total executions]

```
short curCoef = SrcDst[(by * BLOCK_SIZE + ty) * Stride + (bx * BLOCK_SIZE + tx)];
short curQuant = Q[ty * BLOCK_SIZE + tx];

// quantize the current coefficient
if (curCoef < 0)
{
    curCoef = -curCoef;
    curCoef += curQuant>>1;
    curCoef /= curQuant;
    curCoef = -curCoef;
}
else
{
    curCoef += curQuant>>1;
    curCoef /= curQuant;
}
__syncthreads();
curCoef = curCoef * curQuant;

// copy quantized coefficient back to the DCT-plane
SrcDst[(by * BLOCK_SIZE + ty) * Stride + (bx * BLOCK_SIZE + tx)] = curCoef;
```
Bound-Specific Optimization Analysis

- Memory Bandwidth Bound
- Compute Bound
- Latency Bound
Latency

- Memory load: delay from when load instruction executed until data returns
- Arithmetic: delay from when instruction starts until result produced
Hiding Latency

- SM can do many things at once...

- Can “hide” latency as long as there are enough
Occupancy

- Theoretical occupancy, upper bound based on:
  - Threads / block
  - Shared memory usage / block
  - Register usage / thread

- Achieved occupancy
  - Actual measured occupancy of running kernel
  - \((active\_warps / active\_cycles) / MAX\_WARPS\_PER\_SM\)
  - `nvprof` metrics: `achieved_occupancy`
Low Theoretical Occupancy

- \# warps on SM = \# blocks on SM \times \# warps per block

- If low...
  - Not enough blocks in kernel
  - \# blocks on SM limited by threads, shared memory, registers

- CUDA Best Practices Guide has extensive discussion on improving occupancy
Visual Profiler: Low Theoretical Occupancy

**GPU Utilization Is Limited By Shared Memory Usage**

The kernel uses 10.031 KB of shared memory for each block. This shared memory usage is likely preventing the kernel from fully utilizing the GPU. Device "Tesla C2050" is configured to have 48 KB of shared memory for each SM. Because the kernel uses 10.031 KB of shared memory for each block each SM is limited to simultaneously executing 4 blocks (32 warps). Chart "Varying Shared Memory Usage" below shows how changing shared memory usage will change the number of blocks that can execute on each SM.

Optimization: Reduce shared memory usage to increase the number of blocks that can execute on each SM. You can also increase the number of blocks that can execute on each SM by increasing the amount of shared memory available to your kernel. You do this by setting the preferred cache configuration to "prefer shared".

<table>
<thead>
<tr>
<th>Variable</th>
<th>Achieved</th>
<th>Theoretical</th>
<th>Device Limit</th>
<th>Grid Size: [11,1,1] (11 blocks)</th>
<th>Block Size: [256,1,1] (256 threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Blocks</td>
<td>4</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active Warps</td>
<td>8</td>
<td>32</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active Threads</td>
<td>1024</td>
<td>2048</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Occupancy</td>
<td>12.5%</td>
<td>50%</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Visual Profiler: Low Theoretical Occupancy

<table>
<thead>
<tr>
<th>Block Limit</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared Memory/Block</td>
<td>10272</td>
<td>49152</td>
</tr>
<tr>
<td>Block Limit</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>

- Varying Block Size
- Varying Register Count
- Varying Shared Memory Usage
Low Achieved Occupancy

- In theory... kernel allows enough warps on each SM
Low Achieved Occupancy

- In theory... kernel allows enough warps on each SM
- In practice... have low achieved occupancy
- Likely cause is that all SMs do not remain equally busy over duration of kernel execution
Latency Bound, Not Occupancy

- Haar wavelet decomposition
  - Excellent occupancy: 100% theoretical, 91% achieved
  - Still latency bound...
Lots Of Warps, Not Enough Can Execute

- SM can do many things at once...

- No “ready” warp...
Latency Bound, Instruction Stalls

Instruction Latencies May Be Limiting Performance
The kernel has good theoretical and achieved occupancy indicating that there are likely sufficient warps executing on each SM. Since occupancy is not an issue it is likely that performance is limited by the instruction stall reasons described below.

Optimization: Resolve the primary stall issues; synchronization, execution dependency.

Stall Reasons

- execution dependency
- instruction fetch
- data request
- texture
- other
- synchronization
__syncthreads Stall

// global thread id (w.r.t. to total data set)
const int tld_global = (bid * bdim) + tid;
unsigned int idata = (bid * (2 * bdim)) + tid;

// read data from global memory
shared[tid] = id[idata];
shared[tid + bdim] = id[idata + bdim];
__syncthreads();

// this operation has a two way bank conflicts for all threads, this are two
// additional cycles for each warp -- all alternatives to avoid this bank
// conflict are more expensive than the one cycle introduced by serialization
float data0 = shared[2*tid];
float data1 = shared[(2*tid) + 1];
__syncthreads();

// detail coefficient, not further referenced so directly store in
// global memory
od[tid_global + slength_step_half] = (data0 - data1) * INV_SQRT_2;

// offset to avoid bank conflicts
// see the scan example for a more detailed description
unsigned int atid = tid + (tid >> LOG_NUM_BANKS);
Summary: Kernel Optimization

- Application optimization strategy based on primary performance limiter
  - Memory bandwidth
  - Compute resources
  - Instruction and memory latency

- Common optimization opportunities

- Use CUDA profiling tools to identify optimization opportunities
  - Guided analysis
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 Expert Table and Developer Demo Stations

- S3466 - Performance Optimization: Programming Guidelines and GPU Architecture Details Behind Them
  - Thur. 3/21, 9am, Room 210H
- GTC On-Demand website for all GTC talks