Profiling & Tuning Applications

CUDA Course

István Reguly
Introduction

• Why is my application running slow?
• Work it out on paper
• Instrument code
• Profile it
  • NVIDIA Visual Profiler
    • Works with CUDA, needs some tweaks to work with OpenCL
  • nvprof – command line tool, can be used with MPI applications
Identifying Performance Limiters

- CPU: Setup, data movement
- GPU: Bandwidth, compute or latency limited
- Number of instructions for every byte moved
- Algorithmic analysis gives a good estimate
- Actual code is likely different
  - Instructions for loop control, pointer math, etc.
  - Memory access patterns
- How to find out?
  - Use the profiler (quick, but approximate)
  - Use source code modification (takes more work)
Analysis with Source Code Modification

• Time memory-only and math-only versions
  • Not so easy for kernels with data-dependent control flow
  • Good to estimate time spent on accessing memory or executing instructions

• Shows whether kernel is memory or compute bound

• Put an “if” statement depending on kernel argument around math/mem instructions
  • Use dynamic shared memory to get the same occupancy
__global__ void kernel(float *a) {
    int idx = threadIdx.x + blockDim.x + blockIdx.x;
    float my_a;
    my_a = a[idx];
    for (int i = 0; i < 100; i++) my_a = sinf(my_a+i*3.14f);
    a[idx] = my_a;
}

__global__ void kernel(float *a, int prof) {
    int idx = threadIdx.x + blockDim.x + blockIdx.x;
    float my_a;
    if (prof & 1) my_a = a[idx];
    if (prof & 2)
        for (int i = 0; i < 100; i++) my_a = sinf(my_a+i*3.14f);
    if (prof & 1) a[idx] = my_a;
}
Example scenarios

- **Memory-bound**
  - Good overlap between mem-math.
  - Latency is not a problem.

- **Math-bound**
  - Good overlap between mem-math.

- **Well balanced**
  - Good overlap between mem-math.

- **Mem and latency bound**
  - Poor overlap, latency is a problem.
NVIDIA Visual Profiler

• Collects metrics and events during execution
  • Calls to the CUDA API
  • Overall application:
    • Memory transfers
    • Kernel launches
  • Kernels
    • Occupancy
    • Computation efficiency
    • Memory bandwidth efficiency
  • Source-level profiling

• Requires deterministic execution!
Meet the test setup

- 2D gaussian blur with a 5x5 stencil
- 4096^2 grid

```c
__global__ void stencil_v0(float *input, float *output,
                         int sizex, int sizey) {

    const int x = blockIdx.x*blockDim.x + threadIdx.x + 2;
    const int y = blockIdx.y*blockDim.y + threadIdx.y + 2;
    if ((x >= sizex-2) || (y >= sizey-2)) return;
    float accum = 0.0f;
    for (int i = -2; i < 2; i++) {
        for (int j = -2; j < 2; j++) {
            accum += filter[i+2][j+2]*input[sizey*(y+j) + (x+i)];
        }
    }
    output[sizey*y+x] = accum/273.0f;
}
```
Meet the test setup

- NVIDIA K40
  - GK110B
  - SM 3.5
  - ECC on
  - Graphics clocks at 745MHz, Memory clocks at 3004MHz

- CUDA 7.0
  nvcc profiling_lecture.cu -02 -arch=sm_35 -I. -lineinfo -DIT=0
Interactive demo of tuning process
Launch a profiling session

Launch a profiling session.
First look

Timeline

Summary

Guide

Analysis results

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.

Examine GPU Usage

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

Examine individual Kernels

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.
The Timeline

- Host side API calls
- MemCpy
- Compute
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.

- Examine GPU Usage
  Determine your application's overall GPU usage. This analysis requires an application timeline, so your application will be run once to collect it if it is not already available.

- Examine Individual Kernels
  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.

- Delete Existing Analysis Information
  If the application has changed since the last analysis then the existing analysis information may be stale and should be deleted before continuing.

- Switch to unguided analysis
Examine Individual Kernels

Lists all kernels sorted by total execution time: the higher the rank the higher the impact of optimisation on overall performance

<table>
<thead>
<tr>
<th>Rank</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>[1 kernel instances] stencil_v0(float*, float*, int, int)</td>
</tr>
</tbody>
</table>

Initial unoptimised (v0) 8.122ms
Utilisation – Warp Issue Efficiency & Pipe Utilisation

Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Tesla" utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

Both below 60% -> Latency!

Most of it is memory ops

Let’s investigate
Latency analysis

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 "stencil_v0" is most likely limited by instruction and memory latency.

Perform Latency Analysis

The most likely bottleneck to performance for this kernel is instruction and memory latency so you should first perform instruction and memory latency analysis to determine how it is limiting performance.

Perform Compute Analysis

Perform Memory Bandwidth Analysis

Memory throttle -> perform BW analysis
Memory Bandwidth analysis

<table>
<thead>
<tr>
<th>L1/Shared Memory</th>
<th>Transactions</th>
<th>Bandwidth</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<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>40894464</td>
<td>248.782 GB/s</td>
<td></td>
</tr>
<tr>
<td>Global Stores</td>
<td>2621440</td>
<td>16.585 GB/s</td>
<td></td>
</tr>
<tr>
<td>Atomic</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>L1/Shared Total</td>
<td>43515904</td>
<td>265.367 GB/s</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L2 Cache</th>
<th>Transactions</th>
<th>Bandwidth</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Reads</td>
<td>62914560</td>
<td>248.782 GB/s</td>
<td></td>
</tr>
<tr>
<td>L1 Writes</td>
<td>4194304</td>
<td>16.585 GB/s</td>
<td></td>
</tr>
<tr>
<td>Texture Reads</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Atomic</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Noncoherent Reads</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>67108864</td>
<td>265.367 GB/s</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Texture Cache</th>
<th>Transactions</th>
<th>Bandwidth</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Device Memory</th>
<th>Transactions</th>
<th>Bandwidth</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>3756909</td>
<td>14.856 GB/s</td>
<td></td>
</tr>
<tr>
<td>Writes</td>
<td>2904475</td>
<td>11.485 GB/s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>5661384</td>
<td>26.341 GB/s</td>
<td></td>
</tr>
<tr>
<td>ECC Overhead</td>
<td>2451525</td>
<td>9.594 GB/s</td>
<td></td>
</tr>
</tbody>
</table>
Investigate further...

Global memory load efficiency 53.3%
L2 hit rate 96.7%

6-8 transactions per access – something is wrong with how we access memory
Iteration 1 – turn on L1

Memory unit is utilized, but Global Load efficiency became even worse: 20.5%

<table>
<thead>
<tr>
<th>Line / File</th>
<th>profiling_lecture.cu · /home/mgiles/regexy/cuda_course</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 20, Ideal Transactions/Access = 4</td>
</tr>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 18, Ideal Transactions/Access = 4</td>
</tr>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 20, Ideal Transactions/Access = 4</td>
</tr>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 18, Ideal Transactions/Access = 4</td>
</tr>
</tbody>
</table>

Quick & easy step:
Turn on L1 cache by using -Xptxas -dlcm=ca

<table>
<thead>
<tr>
<th></th>
<th>Initial unoptimised (v0)</th>
<th>8.122ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable L1</td>
<td>6.57ms</td>
<td></td>
</tr>
</tbody>
</table>
Cache line utilization

128 bytes (32 floats)
Unit of transaction

Each time a transaction requires more than 1 128B cache line: re-issue

L1 cache enabled:
- 128B transactions
- 4*32B to L2

Min 16, Max 32 transactions
Cache line utilization

128 bytes (32 floats)
Unit of transaction

L1 cache enabled:
- 128B transactions
- 4*32B to L2
Min 4, Max 8 transactions
Iteration 2 – 32x2 blocks

Memory utilization decreased 10%
Performance almost doubles
Global Load Efficiency 50.8%

<table>
<thead>
<tr>
<th>Line / File</th>
<th>profiling_lecture.cu - /home/mgiles/ireguly/cuda_course</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [ 4194304 L2 transactions for 524288 total exec]</td>
</tr>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 7.5, Ideal Transactions/Access = 4 [ 3932160 L2 transactions for 524288 total exec]</td>
</tr>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [ 4194304 L2 transactions for 524288 total exec]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Initial unoptimised (v0)</th>
<th>Enable L1</th>
<th>Blocksize</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8.122ms</td>
<td>6.57ms</td>
<td>3.4ms</td>
</tr>
</tbody>
</table>
Key takeaway

• **Latency/Bandwidth bound**

• Inefficient use of memory system and bandwidth

• **Symptoms:**
  • Lots of transactions per request (low load efficiency)

• **Goal:**
  • Use the whole cache line
  • Improve memory access patterns (coalescing)

• **What to do:**
  • Align data, change block size, change data layout
  • Use shared memory/shuffles to load efficiently
Latency analysis

Stall Reasons

- memory throttle
- execution dependency
- instruction fetch not selected
- memory dependency texture
- other
- pipe busy
- constant
- synchronization

<table>
<thead>
<tr>
<th>Occupancy</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Achieved</td>
<td>41.7%</td>
</tr>
<tr>
<td>Theoretical</td>
<td>50%</td>
</tr>
<tr>
<td>Limiter</td>
<td>Block Size</td>
</tr>
</tbody>
</table>
### Latency analysis

*Optimization: Increase the number of threads in each block to increase the number of warps that can execute on each SM.*

<table>
<thead>
<tr>
<th>Variable</th>
<th>Achieved</th>
<th>Theoretical</th>
<th>Device Limit</th>
<th>Grid Size: [128, 2048, 1] (262144 blocks)</th>
<th>Block Size: [3, 32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occupancy Per SM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active Blocks</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active Warps</td>
<td>26.67</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>41.7%</td>
<td>50%</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threads/Block</td>
<td>64</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warps/Block</td>
<td>2</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block Limit</td>
<td>32</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Latency analysis

Increase the block size so more warps can be active at the same time.

Kepler:
Max 16 blocks per SM
Max 2048 threads per SM
Occupancy – using all “slots”

Increase block size to 32x4

Illustrative only, reality is a bit more complex...
Iteration 3 – 32x4 blocks

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial unoptimised (v0)</td>
<td>8.122</td>
</tr>
<tr>
<td>Enable L1</td>
<td>6.57</td>
</tr>
<tr>
<td>Blocksize</td>
<td>3.4</td>
</tr>
<tr>
<td>Blocksize 2</td>
<td>2.36</td>
</tr>
</tbody>
</table>

Up 10% Full occupancy
Key takeaway

• **Latency bound – low occupancy**
• Unused cycles, exposed latency
• **Symptoms:**
  • High execution/memory dependency, low occupancy
• **Goal:**
  • Better utilise cycles by: having more warps
• **What to do:**
  • Determine occupancy limiter (registers, block size, shared memory) and vary it
Improving memory bandwidth

• L1 is fast, but a bit wasteful (128B loads)
  • 8 transactions on average (minimum would be 4)

• Load/Store pipe stressed
  • Any way to reduce the load?

• Texture cache
  • Dedicated pipeline
  • 32 byte loads
  • const __restrict__ *
  • __ldg()
Iteration 4 – texture cache

<table>
<thead>
<tr>
<th>Texture Cache</th>
<th>Reads</th>
<th>65536000</th>
<th>1,382.851 GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial unoptimised (v0)</td>
<td></td>
<td>8.122ms</td>
<td></td>
</tr>
<tr>
<td>Blocksize 2</td>
<td></td>
<td>2.36ms</td>
<td></td>
</tr>
<tr>
<td>Texture cache</td>
<td></td>
<td>1.53ms</td>
<td></td>
</tr>
</tbody>
</table>
Key takeaway

• Bandwidth bound – Load/Store Unit
• LSU overutilised
• Symptoms:
  • LSU pipe utilisation high, others low
• Goal:
  • Better spread the load between other pipes: use TEX
• What to do:
  • Read read-only data through the texture cache
  • const __restrict__ or __ldg()
Compute utilization could be higher (~78%)
Lots of Integer & memory instructions, fewer FP
Integer ops have lower throughput than FP
Try to amortize the cost: increase compute per byte
Instruction Level Parallelism

- Remember, GPU is in-order:
  - $a = b + c$
  - $d = a + e$

- Second instruction cannot be issued before first
  - But it can be issued before the first finishes – if there is no dependency

- Applies to memory instructions too – latency much higher (counts towards stall reasons)
Instruction Level Parallelism

for (j=0; j<2; j++)
    acc += filter[j]*input[x+j];

tmp = input[x+0]
    acc += filter[0]*tmp

tmp = input[x+1]
    acc += filter[1]*tmp

#pragma unroll can help ILP
Create two accumulators
Or...

for (j=0; j<2; j++) {
    acc0 += filter[j]*input[x+j];
    acc1 += filter[j]*input[x+j+1];
}

tmp = input[x+0]
    acc0 += filter[0]*tmp

tmp = input[x+1]
    acc0 += filter[1]*tmp

tmp = input[x+0+1]
    acc1 += filter[0]*tmp

tmp = input[x+1+1]
    acc1 += filter[1]*tmp

Process 2 points per thread
Bonus data re-use (register caching)
Iteration 5 – 2 points per thread

<table>
<thead>
<tr>
<th>Description</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial unoptimised (v0)</td>
<td>8.122ms</td>
</tr>
<tr>
<td>Texture cache</td>
<td>1.53ms</td>
</tr>
<tr>
<td>2 points</td>
<td>1.07ms</td>
</tr>
</tbody>
</table>
Key takeaway

• Latency bound – low instruction level parallelism
• Unused cycles, exposed latency
• Symptoms:
  • High execution dependency, one “pipe” saturated
• Goal:
  • Better utilise cycles by: increasing parallel work per thread
• What to do:
  • Increase ILP by having more independent work, e.g. more than 1 output value per thread
  • #pragma unroll
Iteration 6 – 4 points per thread

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Execution Count (% of total)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FP32</td>
</tr>
<tr>
<td>Initial unoptimised</td>
<td>8.122ms</td>
<td></td>
</tr>
<tr>
<td>2 points</td>
<td>1.07ms</td>
<td></td>
</tr>
<tr>
<td>4 points</td>
<td>0.95ms</td>
<td></td>
</tr>
</tbody>
</table>

168 GB/s device BW
Checklist

• cudaDeviceSynchronize()
  • Most API calls (e.g. kernel launch) are asynchronous
  • Overhead when launching kernels
  • Get rid of cudaDeviceSynchronize() to hide this latency
  • Timing: events or callbacks  CUDA 5.0+

• Cache config 16/48 or 48/16 kB L1/shared (default is 48k shared!) on Kepler
  • cudaSetDeviceCacheConfig
  • cudaFuncSetCacheConfig
  • Check if shared memory usage is a limiting factor
Checklist

• Occupancy
  • Max 1536 threads or 8 blocks per SM on Fermi (2048/16 for Kepler, 2048/32 for Maxwell)
  • Limited amount of registers and shared memory
    • Max 255 registers/thread, rest is spilled to global memory
    • You can explicitly limit it (-maxregcount=xx)
    • 48kB/16kB shared/L1: don’t forget to set it
  • Visual Profiler tells you what is the limiting factor
  • In some cases though, it is faster if you don’t maximise it (see Volkov paper) -> Autotuning!
Verbose compile

• Add –Xptxas=-v

ptxas info : Compiling entry function '_Z10fem_kernelPiS_' for 'sm_20'
ptxas info : Function properties for _Z10fem_kernelPiS_
  856 bytes stack frame, 980 bytes spill stores, 1040 bytes spill loads
ptxas info : Used 63 registers, 96 bytes cmem[0]

• Check profiler figures for best occupancy
Checklist

• Precision mix (e.g. 1.0 vs 1.0f) – cuobjdump
  • F2F.F64.F32 (6* the cost of a multiply)
  • IEEE standard: always convert to higher precision
  • Integer multiplications are now expensive (6*)

• cudaMemcpy
  • Introduces explicit synchronisation, high latency
  • Is it necessary?
    • May be cheaper to launch a kernel which immediately exits
  • Could it be asynchronous? (Pin the memory!)
Auto-tuning

• Several parameters that affect performance
  • Block size
  • Amount of work per block
  • Application specific

• Which combination performs the best?

• Auto-tuning with Flamingo
  • #define/read the sizes, recompile/rerun combinations
Auto-tuning Case Study

• Thread cooperation on sparse matrix-vector product
  • Multiple threads doing partial dot product on the row
  • Reduction in shared memory

• Auto-tune for different matrices
  • Difficult to predict caching behavior
  • Develop a heuristic for cooperation vs. average row length
Autotuning Case Study

![Graph showing run time (seconds) vs. number of cooperating threads for different applications. The x-axis represents the number of cooperating threads ranging from 1 to 32, and the y-axis represents run time in seconds ranging from $10^{-4}$ to $10^{-2}$. The graph includes four applications: atmosmodd, crankseg_2, shallow_water1, and webbase−1M, each represented by different symbols and lines.]

- **atmosmodd (2.63)**
- **crankseg_2 (14.89)**
- **shallow_water1 (2.00)**
- **webbase−1M (1.76)**
- **cant (8.01)**
Conclusions

- Iterative approach to improving a code’s performance
  - Identify hotspot
  - Find performance limiter, understand why it’s an issue
  - Improve your code
  - Repeat

- Managed to achieve a 8.5x speedup

- Shown how NVVP guides us and helps understand what the code does

- There is more it can show...

Rapid code development with Thrust
Thrust

• Open High-Level Parallel Algorithms Library
• Parallel Analog of the C++ Standard Template Library (STL)
  • Vector containers
  • Algorithms
• Comes with the toolkit
• Productive way to use CUDA
```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```
Productivity

• Containers
  • host_vector
  • device_vector

• Memory management
  • Allocation, deallocation
  • Transfers

• Algorithm selection
  • Location is implicit

```cpp
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host data to device memory
thrust::device_vector<int> d_vec = h_vec;

// write device values from the host
d_vec[0] = 27;
d_vec[1] = 13;

// read device values from the host
int sum = d_vec[0] + d_vec[1];
// invoke algorithm on device
thrust::sort(d_vec.begin(), d_vec.end());
```
Productivity

- Large set of algorithms
  - ~100 functions
  - CPU, GPU

- Flexible
  - C++ templates
  - User-defined types
  - User-defined operators

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>Sum of a sequence</td>
</tr>
<tr>
<td>find</td>
<td>First position of a value in a sequence</td>
</tr>
<tr>
<td>mismatch</td>
<td>First position where two sequences differ</td>
</tr>
<tr>
<td>count</td>
<td>Number of instances of a value</td>
</tr>
<tr>
<td>inner_product</td>
<td>Dot product of two sequences</td>
</tr>
<tr>
<td>merge</td>
<td>Merge two sorted sequences</td>
</tr>
</tbody>
</table>
Portability

• Implementations
  • CUDA C/C++
  • Threading Building Blocks
  • OpenMP
  • Interoperable with anything CUDA based

• Recompile

• Mix backends
  
  nvcc -DTHRUST_DEVICE_SYSTEM=THRUST_HOST_SYSTEM_OMP

```cpp
thrust::omp::vector<float> my_omp_vec(100);
thrust::cuda::vector<float> my_cuda_vec(100);
```
Interoperability

• Thrust containers and raw pointers
  • Use container in CUDA kernel
    thrust::device_vector<int> d_vec(...);
    cuda_kernel<<<N, 128>>>(some_argument_d,
                        thrust::raw_pointer_cast(&d_vec[0]));
  • Use a device pointer in thrust algorithms (not a vector
    though, no begin(), end(), resize() etc.)

int *dev_ptr;
cudaMalloc((void**)dev_ptr, 100*sizeof(int));

thrust::device_ptr<int> dev_ptr_thrust(dev_ptr);
thrust::fill(dev_ptr_thrust, dev_ptr_thrust+100, 0);
Thrust

- Constantly evolving
- Reliable – comes with the toolkit, tested every day with unit tests
- Performance – specialised implementations for different hardware
- Extensible – allocators, back-ends, etc.
Thrust documentation

http://thrust.github.io/doc/modules.html