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

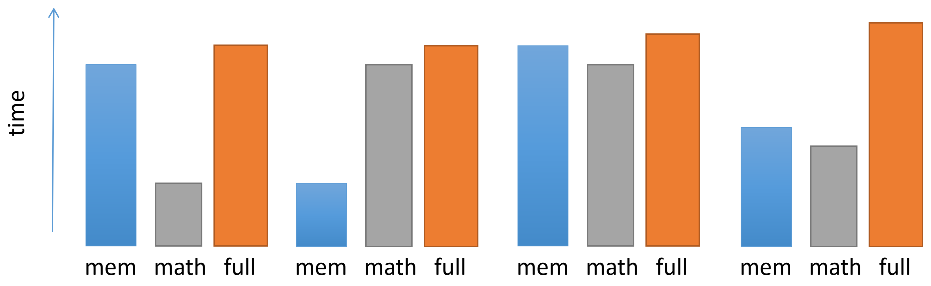
- Time memory-only and math-only versions
  - Not so easy for kernels with data-dependent control flow
  - Good to estimate time spent on accessing memory or executing instructions
- Shows whether kernel is memory or compute bound
- Put an “if” statement depending on kernel argument around math/mem instructions
  - Use dynamic shared memory to get the same occupancy

Analysis with Source Code Modification

```
__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 $\frac{1}{273}$
- 4096^2 grid

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

```
__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;
}
```

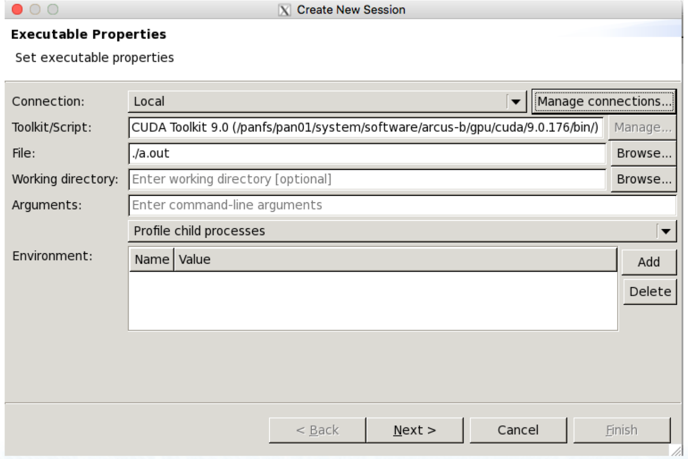
Meet the test setup

- NVIDIA K40
  - GK110B
  - SM 3.5
  - ECC on
  - Graphics clocks at 745MHz, Memory clocks at 3004MHz
- CUDA 9.0

```
nvcc profiling_lecture.cu -O2 -arch=sm_35 -I. -lineinfo -DIT=0
```

Interactive demo of tuning process

Launch a profiling session



Create New Session

Executable Properties

Set executable properties

Connection: Local

Toolkit/Script: CUDA Toolkit 9.0 (/panfs/pan01/system/software/arcus-b/gpu/cuda/9.0.176/bin/)

File: ./a.out

Working directory: Enter working directory [optional]

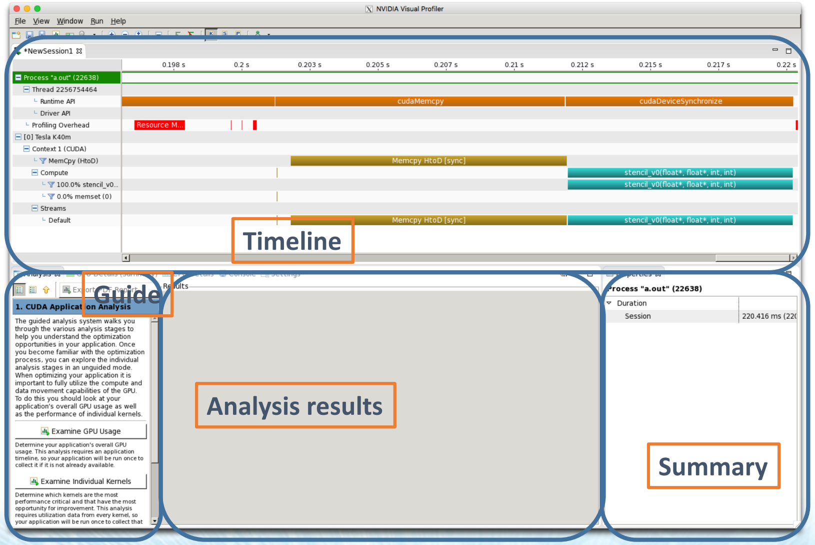
Arguments: Enter command-line arguments

Profile child processes

Environment:

Name	Value

First look



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.

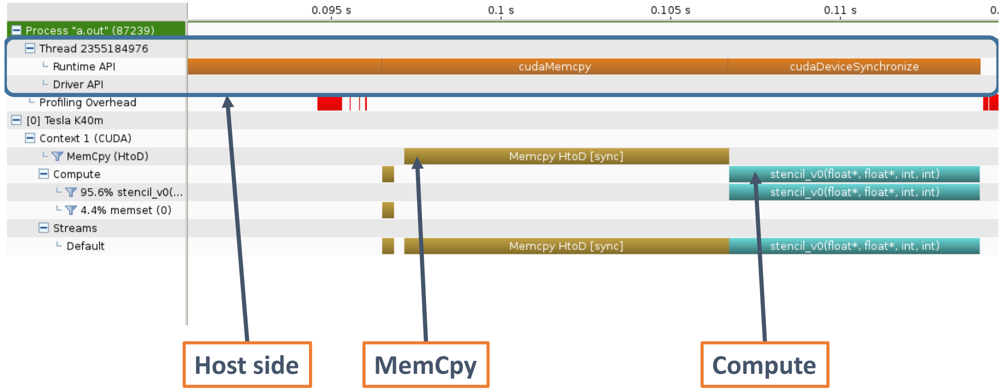
Examine Individual Kernels

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

Process "a.out" (22638)

Duration	
Session	220.416 ms (220

The Timeline



Host side

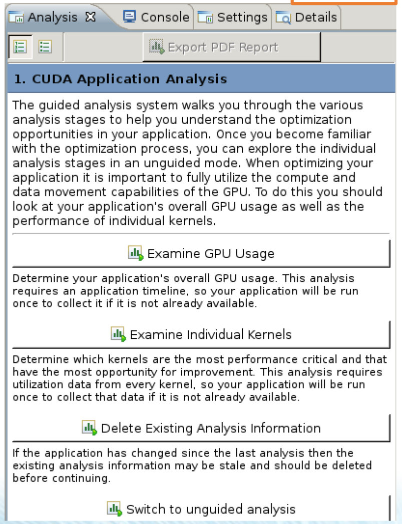
API calls

MemCpy

Compute

Analysis

Guided



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

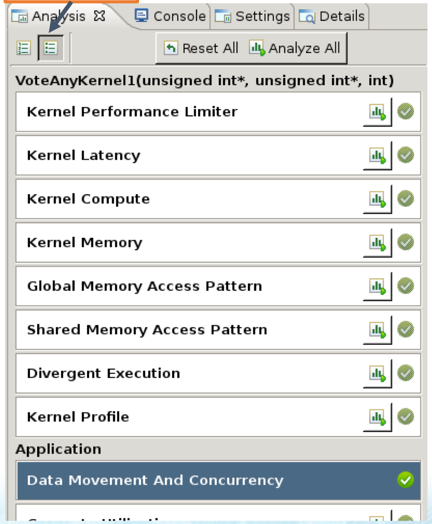
Determine which kernels are the most performance critical and that have the most opportunity for improvement. This analysis requires utilization data from every kernel, so your application will be run once to collect that data if it is not already available.

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

Unguided



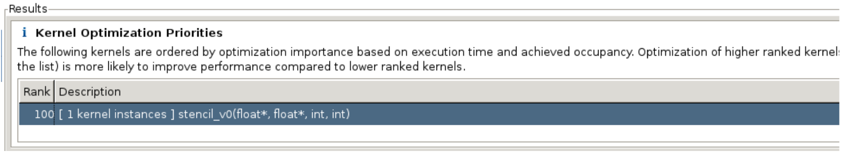
VoteAnyKernel1(unsigned int*, unsigned int*, int)

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

Application

- Data Movement And Concurrency ✓

Examine Individual Kernels



Results

Kernel Optimization Priorities

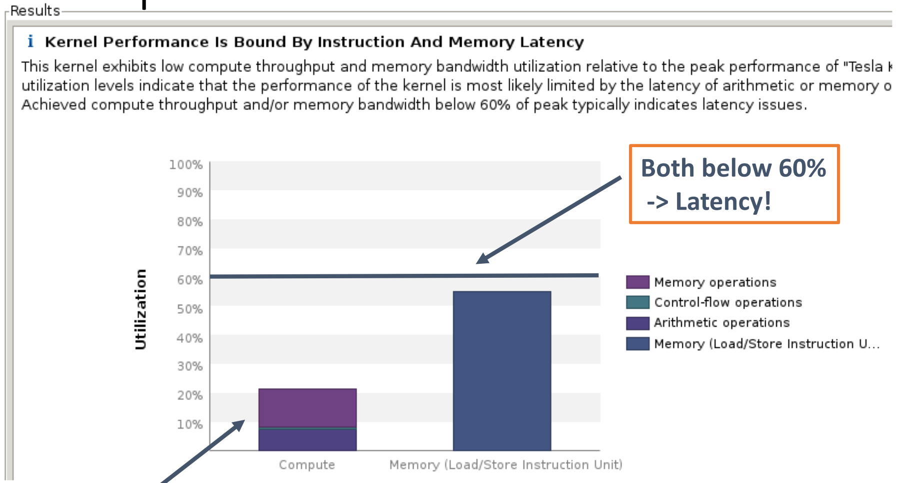
The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernel the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[1 kernel instances] stencil_v0(float*, float*, int, int)

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

Initial unoptimised (v0)	**8.25ms**

Utilisation – Warp Issue Efficiency & Pipe Utilisation



Results

Kernel Performance Is Bound By Instruction And Memory Latency

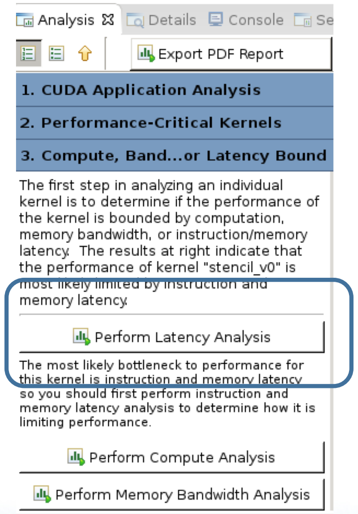
This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Tesla k utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory o Achieved compute throughput and/or memory bandwidth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

Both below 60% -> Latency!

Most of it is memory ops

Let’s investigate

Latency analysis



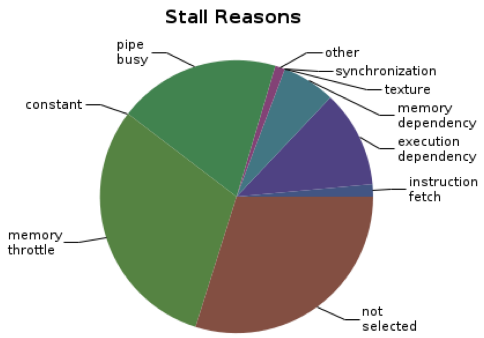
1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Band...or Latency Bound

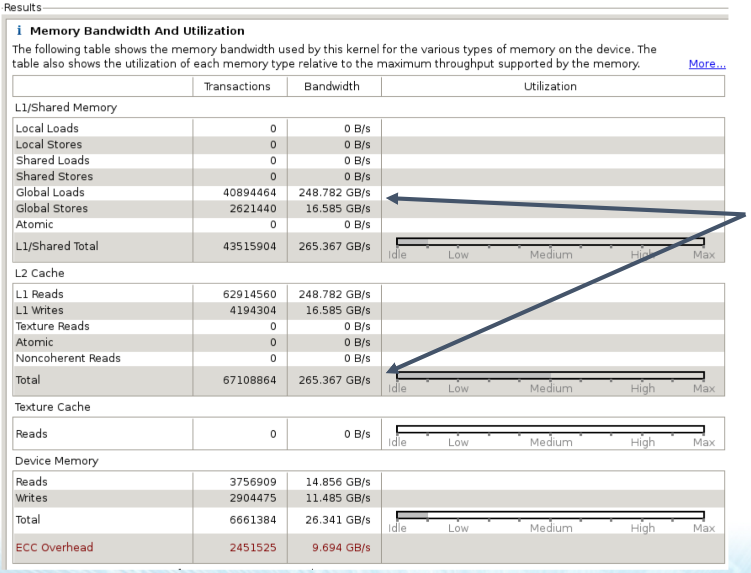
The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "stencil_v0" is most likely limited by instruction and memory latency.

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.



Memory throttle -> perform BW analysis

Memory Bandwidth analysis


Results

Memory Bandwidth And Utilization

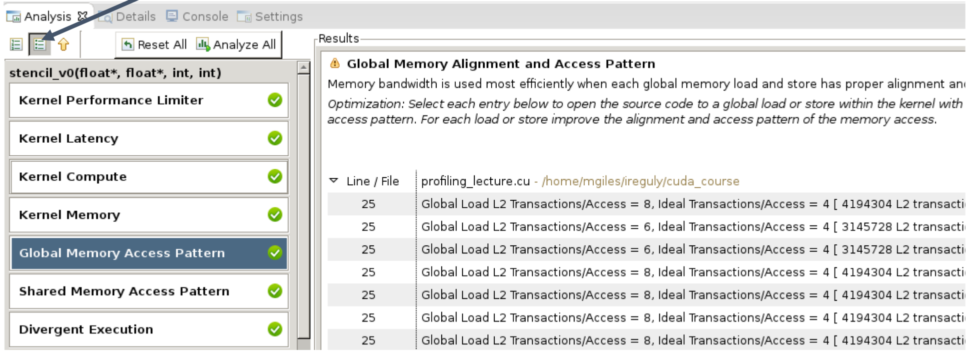
The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	40894464	248.782 GB/s	
Global Stores	2621440	16.585 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	43515904	265.367 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	62914560	248.782 GB/s	
L1 Writes	4194304	16.585 GB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Total	67108864	265.367 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	3756909	14.856 GB/s	
Writes	2904475	11.485 GB/s	
Total	6661384	26.341 GB/s	Idle Low Medium High Max
ECC Overhead	2451525	9.694 GB/s	

L1 cache not used…

Investigate further…

Unguided



stencil_v0(float*, float*, int, int)

- Kernel Performance Limiter ✓
- Kernel Latency ✓
- Kernel Compute ✓
- Kernel Memory ✓
- Global Memory Access Pattern ✓
- Shared Memory Access Pattern ✓
- Divergent Execution ✓

Results

Global Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each global memory load and store has proper alignment an

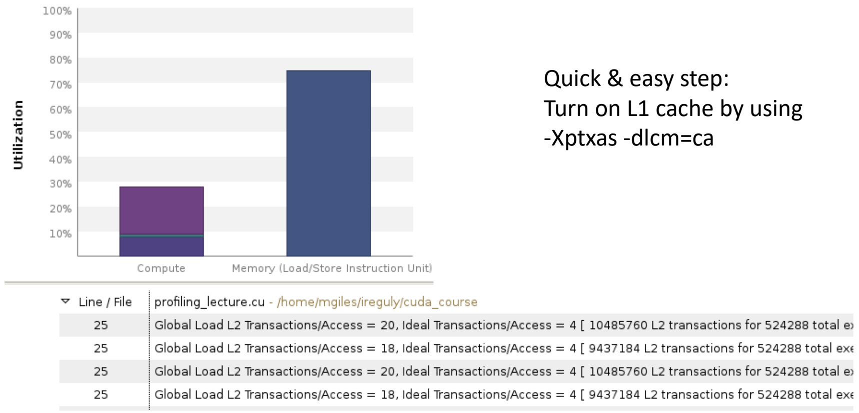
Optimization: Select each entry below to open the source code to a global load or store within the kernel with access pattern. For each load or store improve the alignment and access pattern of the memory access.

Line / File	profiling_lecture.cu - /home/mgiles/ireguly/cuda_course
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 6, Ideal Transactions/Access = 4 [3145728 L2 transacti
25	Global Load L2 Transactions/Access = 6, Ideal Transactions/Access = 4 [3145728 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti

6-8 transactions per access – something is wrong with how we access memory

Global memory load efficiency 53.3%
L2 hit rate 96.7%

Iteration 1 – turn on L1



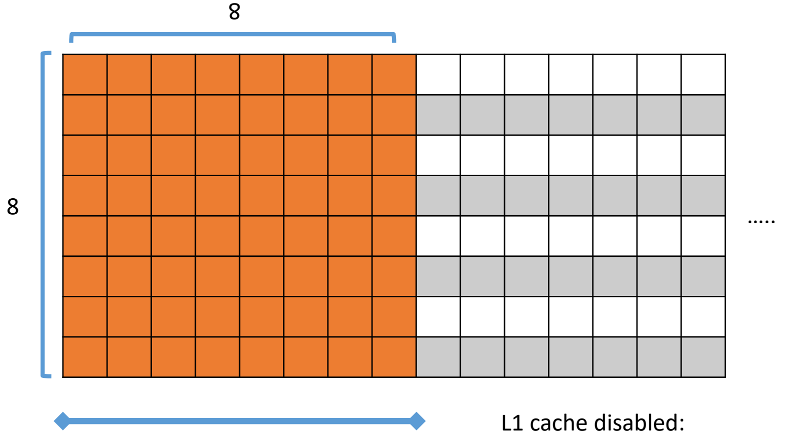
Line / File	profiling_lecture.cu - /home/mgiles/ireguly/cuda_course
25	Global Load L2 Transactions/Access = 20, Ideal Transactions/Access = 4 [10485760 L2 transactions for 524288 total ex
25	Global Load L2 Transactions/Access = 18, Ideal Transactions/Access = 4 [9437184 L2 transactions for 524288 total exe
25	Global Load L2 Transactions/Access = 20, Ideal Transactions/Access = 4 [10485760 L2 transactions for 524288 total ex
25	Global Load L2 Transactions/Access = 18, Ideal Transactions/Access = 4 [9437184 L2 transactions for 524288 total exe

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

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

Initial unoptimised (v0)	**8.25ms**
Enable L1	**6.57ms**

Cache line utilization

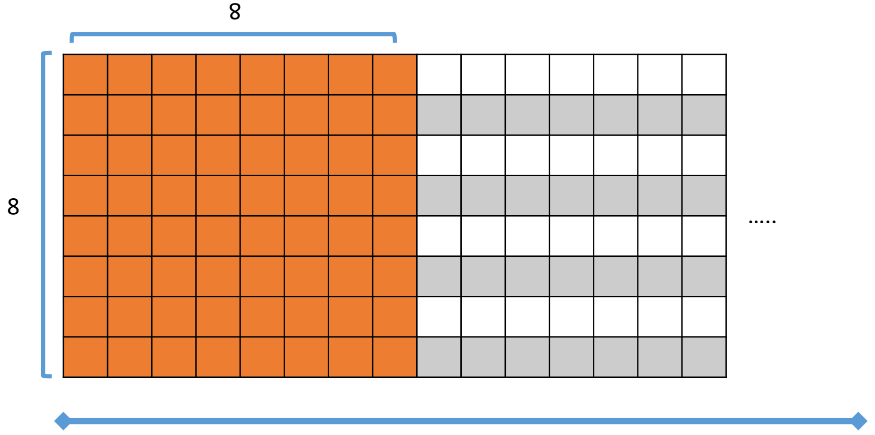


32 bytes (8 floats)
Unit of transaction

L1 cache disabled:
-> 32B transactions

Min 4, Max 8 transactions

Cache line utilization



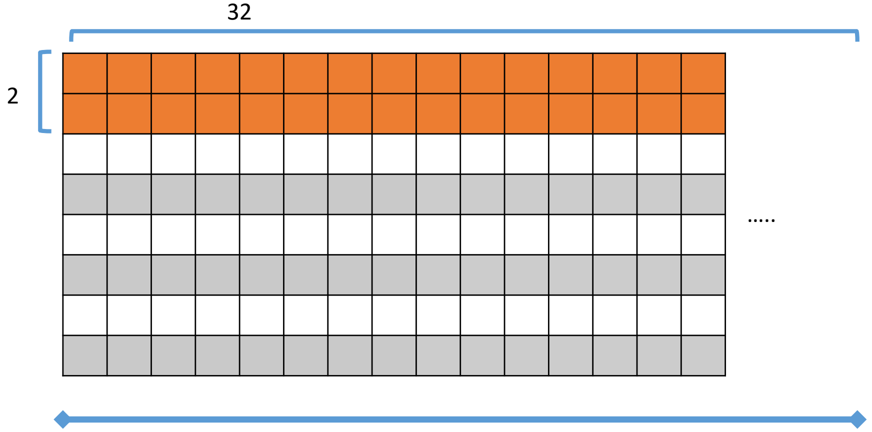
128 bytes (32 floats)
Unit of transaction

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

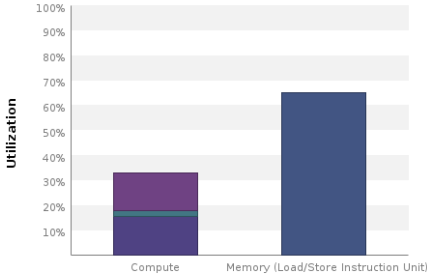


128 bytes (32 floats)
Unit of transaction

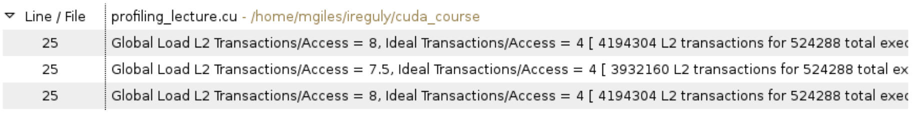
L1 cache enabled:
-> 128B transactions
-> 4*32B to L2

Min 4, Max 8 transactions

Iteration 2 – 32x2 blocks



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



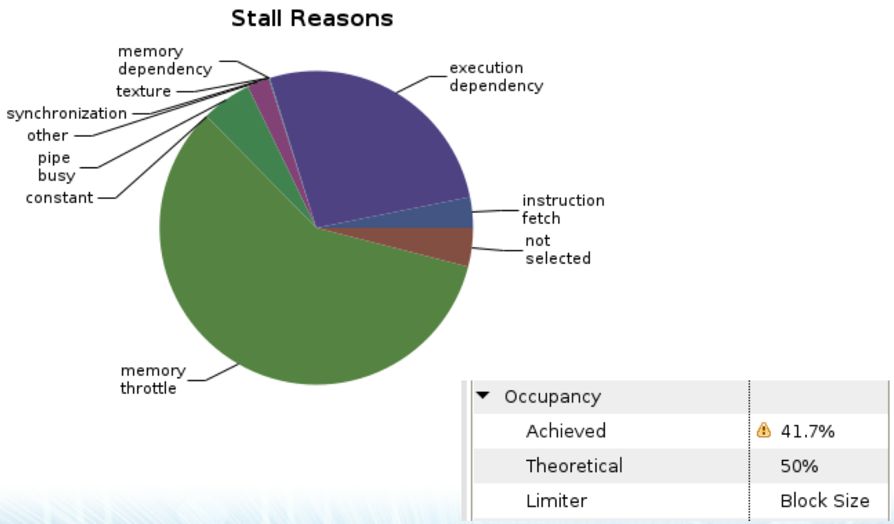
```
Line / File   profiling_lecture.cu - /home/mgiles/ireguly/cuda_course
25            Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [ 4194304 L2 transactions for 524288 total exec
25            Global Load L2 Transactions/Access = 7.5, Ideal Transactions/Access = 4 [ 3932160 L2 transactions for 524288 total ex
25            Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [ 4194304 L2 transactions for 524288 total exec
```

Initial unoptimised (v0)	**8.25ms**
Enable L1	**6.57ms**
Blocksize	**3.4ms**

Key takeaway

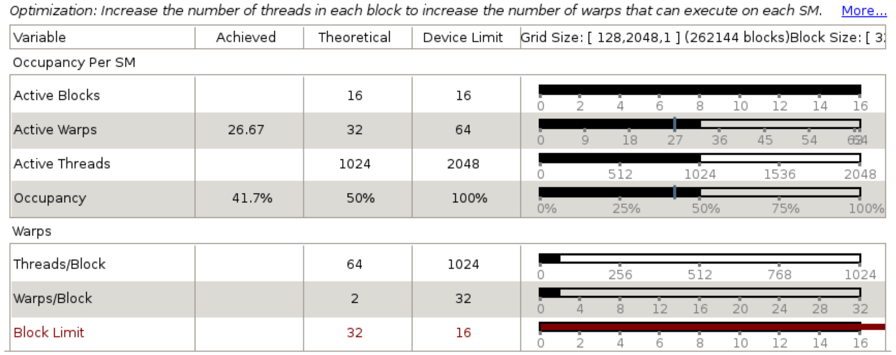
- **Latency/Bandwidth bound**
- Inefficient use of memory system and bandwidth
- Symptoms:
  - Lots of transactions per request (low load efficiency)
- Goal:
  - Use the whole cache line
  - Improve memory access patterns (coalescing)
- What to do:
  - Align data, change block size, change data layout
  - Use shared memory/shuffles to load efficiently

Latency analysis



Occupancy	
Achieved	⚠ 41.7%
Theoretical	50%
Limiter	Block Size

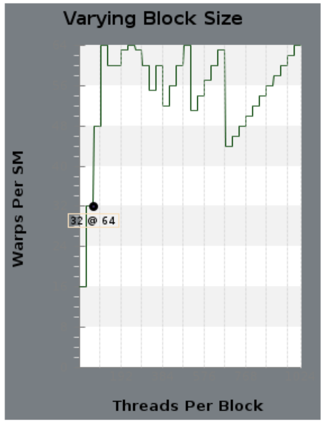
Latency analysis



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

Variable	Achieved	Theoretical	Device Limit	Grid Size: [128,2048,1] (262144 blocks)Block Size: [3
Occupancy Per SM				
Active Blocks		16	16	
Active Warps	26.67	32	64	
Active Threads		1024	2048	
Occupancy	41.7%	50%	100%	
Warps				
Threads/Block		64	1024	
Warps/Block		2	32	
Block Limit		32	16	

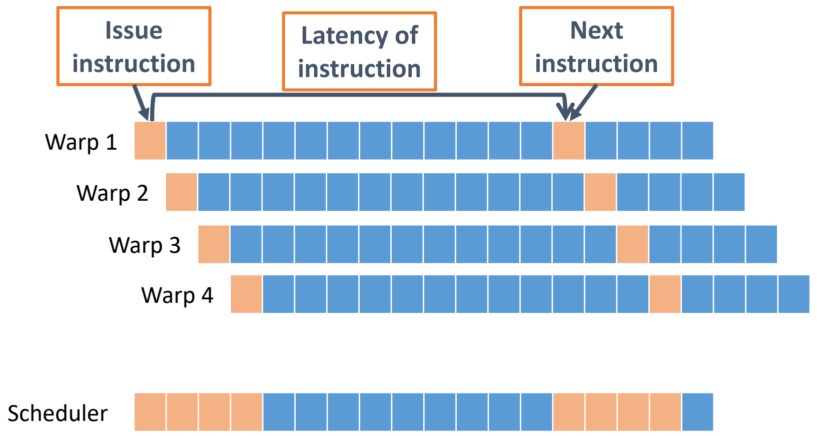
Latency analysis



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

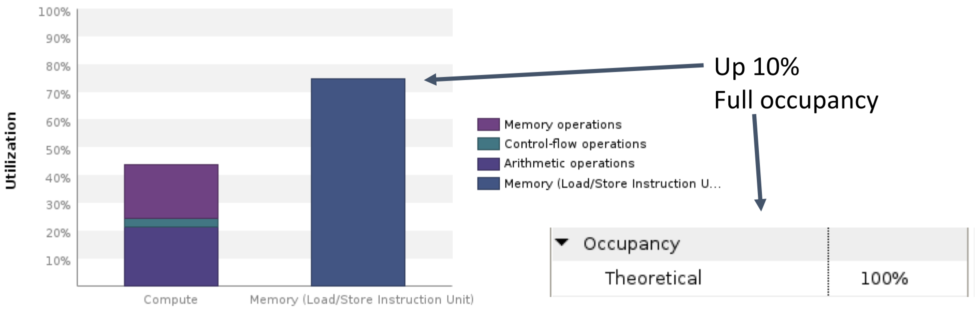
Occupancy – using all "slots"



Increase block size to 32x4

Illustrative only, reality is a bit more complex…

Iteration 3 – 32x4 blocks



Up 10%
Full occupancy

Occupancy	
Theoretical	100%

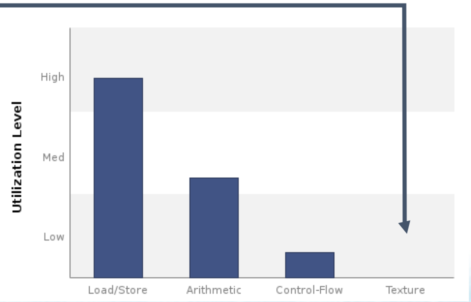
Initial unoptimised (v0)	**8.25ms**
Enable L1	**6.57ms**
Blocksize	**3.4ms**
Blocksize 2	**2.36ms**

Key takeaway

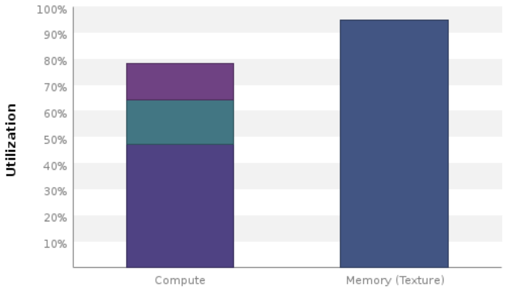
- **Latency bound – low occupancy**
- Unused cycles, exposed latency
- Symptoms:
  - High execution/memory dependency, low occupancy
- Goal:
  - Better utilise cycles by: having more warps
- What to do:
  - Determine occupancy limiter (registers, block size, shared memory) and vary it

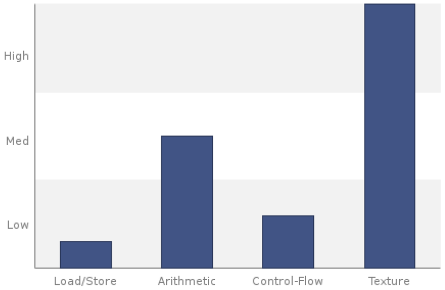
Improving memory bandwidth

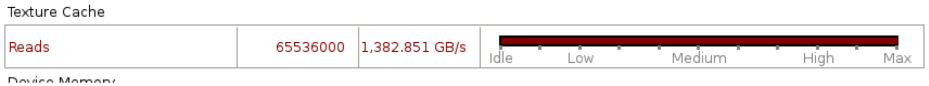
- L1 is fast, but a bit wasteful (128B loads)
  - 8 transactions on average (minimum would be 4)
- Load/Store pipe stressed
  - Any way to reduce the load?
- Texture cache
  - Dedicated pipeline
  - 32 byte loads
  - const __restrict__ *
  - __ldg()



Iteration 4 – texture cache







Texture Cache

Reads	65536000	1,382.851 GB/s	Idle Low Medium High Max

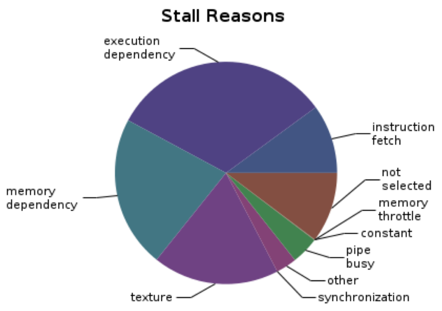
Device Memory

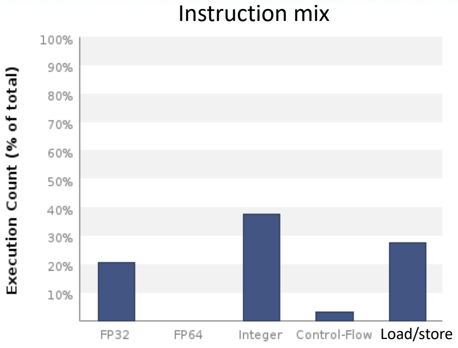
Initial unoptimised (v0)	**8.25ms**
Blocksize 2	**2.36ms**
Texture cache	**1.53ms**

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()

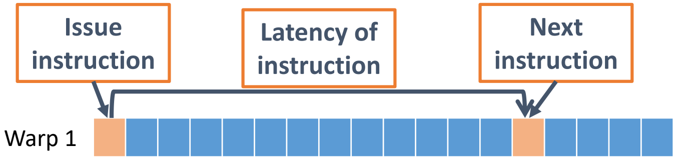
Compute analysis





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:

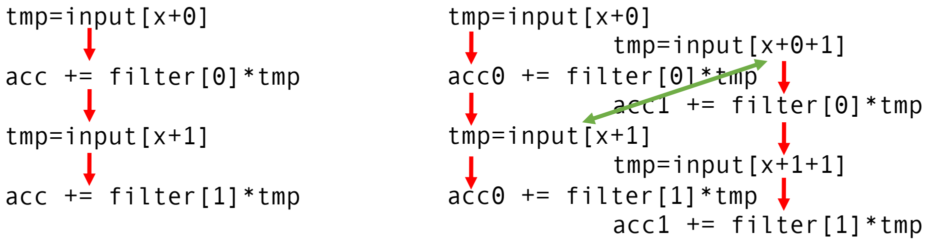
```
a=b+c     a=b+c

d=a+e     d=e+f
```

- 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];
```

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



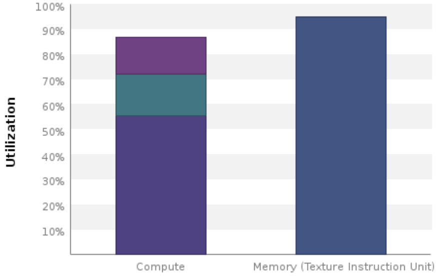
```
tmp=input[x+0]

acc += filter[0]*tmp

tmp=input[x+1]

acc += filter[1]*tmp
```

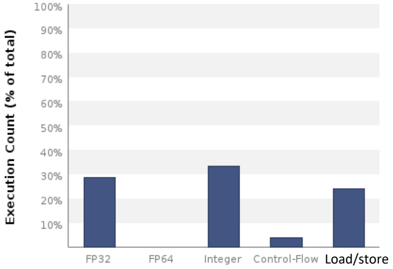
```
tmp=input[x+0]
                tmp=input[x+0+1]
acc0 += filter[0]*tmp
                acc1 += filter[0]*tmp
tmp=input[x+1]
                tmp=input[x+1+1]
acc0 += filter[1]*tmp
                acc1 += filter[1]*tmp
```

#pragma unroll can help ILP
Create two accumulators
Or…

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

Iteration 5 – 2 points per thread



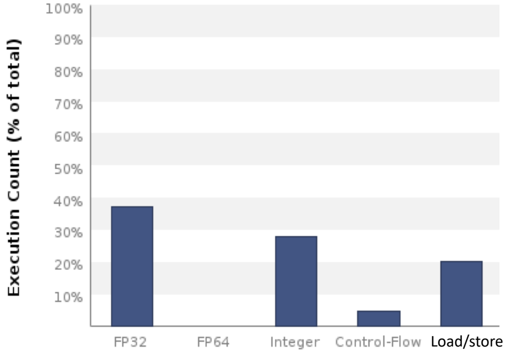


Initial unoptimised (v0)	**8.25ms**
Texture cache	**1.53ms**
2 points	**1.07ms**

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



168 GB/s device BW

Initial unoptimised (v0)	**8.25ms**
2 points	**1.07ms**
4 points	**0.95ms**

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 2048 threads or 16 blocks per SM on Kepler
  - Limited amount of registers and shared memory
    - Max 255registers/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!)

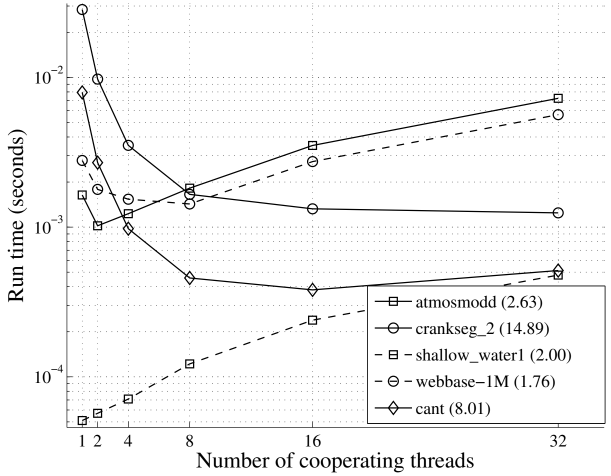
Auto-tuning

- 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



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…

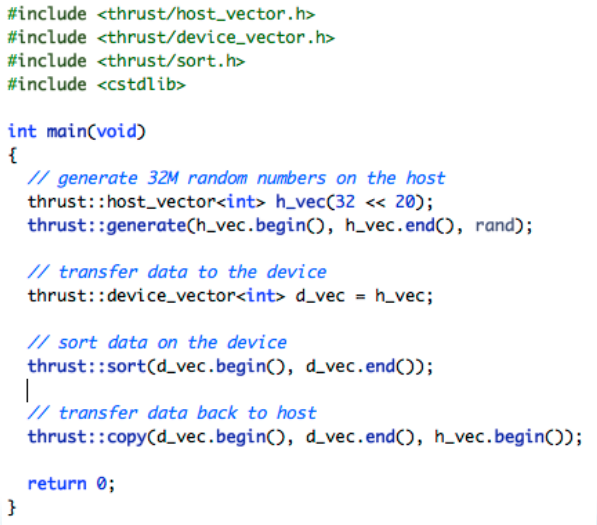
References: C. Angerer, J. Demouth, "CUDA Optimization with NVIDIA Nsight Eclipse Edition", GTC 2015

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

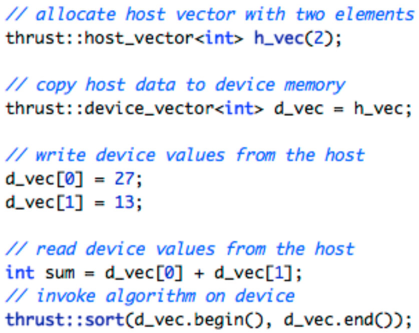
Example



```
#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



```
// 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

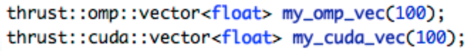
- Large set of algorithms
  - ~100 functions
  - CPU, GPU
- Flexible
  - C++ templates
  - User-defined types
  - User-defined operators

Algorithm	Description
reduce	Sum of a sequence
find	First position of a value in a sequence
mismatch	First position where two sequences differ
count	Number of instances of a value
inner_product	Dot product of two sequences
merge	Merge two sorted sequences

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
```



```
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