CUDA Profiling

Profiling & Debugging tools

Albert García García

agarcia @ dtic.ua.es

Objectives

- To learn to debug CUDA kernels
  - cuda-gdb
  - Printf debug info
  - Nsight
  - Cuda-memcheck
- To learn to profile CUDA programs
  - Visual Profiler
  - NVProof
- Some practical examples

Profiling and debugging

- The developed software does not always work as we intended
- Need to know the execution flow as well as the status of the variables being used
- Although the code runs on a memory and processor different from the primary one, it is possible to debug code in CUDA!!
- There are several tools available for debugging and analyzing CUDA code :
  - Available at the NVIDIA website
  - Nsight for Visual Studio and Eclipse
  - NVProfiler
  - cuda-gdb
  - nvprof

Debugging

- Unix-based Operating System
  - **Cuda-gdb**
    - Before the appearance of this tool, programming was very tedious due to the impossibility of debugging CUDA code
    - Supports real-time debugging while maintaining similar operation to the popular C GDB code debugger.
    - Allows kernels to be debugged directly on the GPU

Debugging

- Unix-based Operating System
  - **Cuda-gdb**
    - CUDA Status: Information about the GPUs installed on your system and their computing capabilities
    - Allows you to set breakpoints in CUDA C code
    - Inspection of GPU memory, including global and shared memory
    - Inspection of blocks and threads residing on the GPU
    - Step by step execution (warp-level)

Debugging

– **Cuda-gdb: enable debugging flags**
  – To enable debugging on the host and gpu code you need to specify the -g and -G flags respectively

```
$nvcc -g -G seqRuntime.cu -o seqRuntime
```

Compile code with the debug flags:
– Host code : -g
– Device code: -G

Here are the basic commands for using the debugger :

- **breakpoint (b):** Sets a breakpoint in the code to stop the execution at that point. The function name or the line of code can be passed as an argument
- **run (r):** Run the application in the debugger

Debugging

- **Cuda-gdb**:
  - **next (n):** Moves to the next line of code
  - **continue (c):** Continue execution to the next breakpoint or to the end of the program
  - **backtrace (bt):** displays the contents of the status stack including method calls
  - **thread:** list of the cpu threads that are running
  - **cuda thread:** list of active GPU threads (if any)
    - (cuda-gdb) cuda thread 5 [Switching focus to CUDA kernel 2 block (2,0,0), thread (5,0,0)
  - **cuda kernel:** lists the active kernels in gpu and also allows you to switch the focus to a particular GPU thread.
    - (cuda-gdb) cuda kernel 2 block 1,0,0 thread 3,0,0
      [Switching focus to CUDA kernel 2 block (1,0,0), thread (3,0,0)

Debugging

– **Cuda-gdb: initialization**
  – We use cuda-gdb to initialize the debugger

```
$ cuda-gdb seqRuntime
NVIDIA (R) CUDA Debugger
4.0 release
Portions Copyright (C) 2007-2011 NVIDIA Corporation
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu"…
Using host libthread_db library "/lib/libthread_db.so.1".
```

Debugging

- **Cuda-gdb: code inspection**
  - Using the command l fill we can show the source code

```
(cuda-gdb) l fill
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    if (tid < n) a[tid] = tid;
  }

  void fill(int* d_a, int n)
  {
    int nThreadsPerBlock= 512;
    int nBlocks= n/nThreadsPerBlock + ((n%nThreadsPerBlock)?1:0);

    fillKernel <<< nBlocks, nThreadsPerBlock >>> (d_a, n);
```

Debugging

- **Example**
- Set a breakpoint in line 12
- Run the program

```
(cuda-gdb) b 12
Breakpoint 1 at 0x401e30: file seqRuntime.cu, line 12.
(cuda-gdb) r
Starting program: /home/rmfarber/foo/ex1-3
[Thread debugging using libthread_db enabled]
[New process 3107]
[New Thread 139660862195488 (LWP 3107)]
[Context Create of context 0x1ed03a0 on Device 0]
Breakpoint 1 at 0x20848a8: file seqRuntime.cu, line 12.
[Launch of CUDA Kernel 0 (thrust::detail::device::cuda::detail::
launch_closure_by_value<thrust::detail::device::cuda::for_each_n_
closure<thrust::device_ptr<unsigned long long>, unsigned int, thrust::
detail::generate_functor<thrust::detail::fill_functor<unsigned long
long> > > ><<<(28,1,1),(768,1,1)>>>) on Device 0]
[Launch of CUDA Kernel 1 (fillKernel<<<(1954,1,1),(512,1,1)>>>) on
Device 0]
[Switching focus to CUDA kernel 1, grid 2, block (0,0,0), thread (0,0,0),
device 0, sm 0, warp 0, lane 0]

Breakpoint 1, fillKernel<<<(1954,1,1),(512,1,1)>>> (a=0x200100000,
n=50000)
    at seqRuntime.cu:12
12      if (tid < n) a[tid] = tid;
```

Debugging

- Show thread ID on screen (threadId)

```
(cuda-gdb) p tid
$1 = 0
```

- We change threads and show the content of a variable per screen

```
(cuda-gdb) cuda thread(403)
[Switching focus to CUDA kernel 1, grid 2, block (0,0,0),
thread (403,0,0), device 0, sm 0, warp 12, lane 19]
12       if (tid < n) a[tid] = tid;
(cuda-gdb) p tid
$2 = 403
```

- Finish debugging and exit

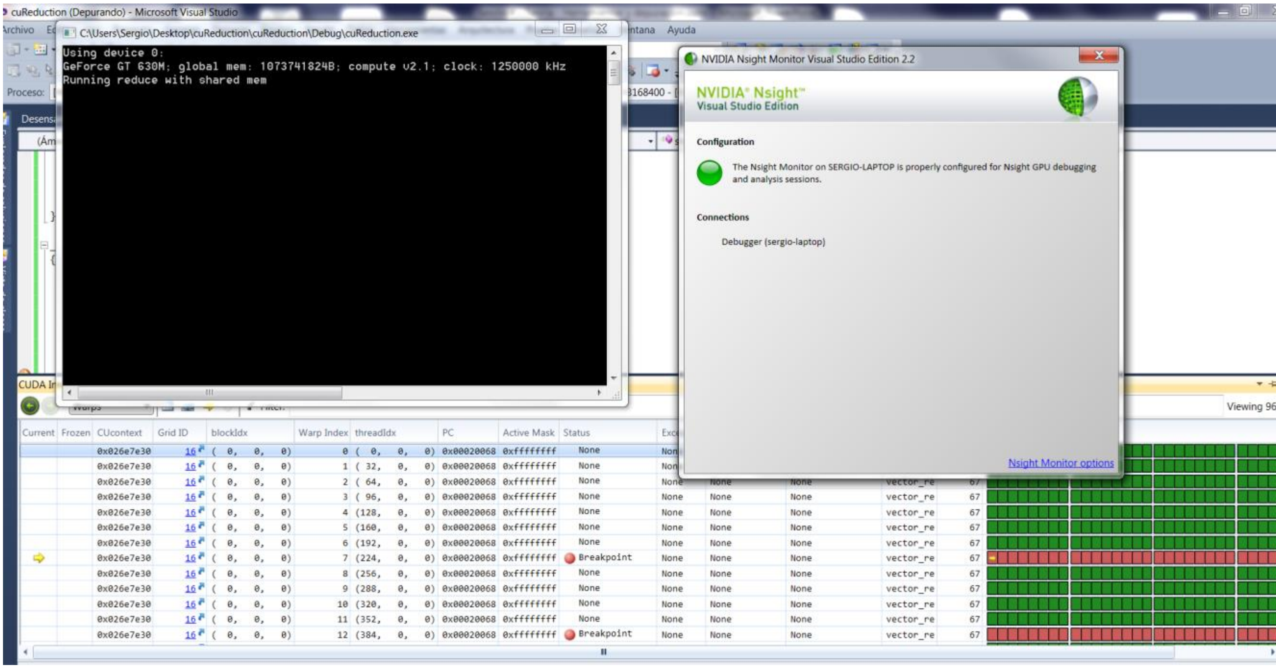
```
(cuda-gdb) quit
The program is running.    Exit anyway? (y or n) y
```

Debugging

- Debugging on Windows-based systems
  - **NVIDIA Parallel NSIGHT:**
    - Not everyone develops on Unix-based platforms
    - Around the end of 2009, NVIDIA created a debugging tool for Windows developers and specifically for Visual Studio IDE users
    - Like cuda-gdb, it allows you to debug applications with thousands of threads.
    - breakpoints can be set anywhere in the code
    - Visual and direct inspection of GPU memory, checking for out-of-bounds access

Debugging

– NVIDIA Parallel NSIGHT: example



NVIDIA
3DPERCEPTIONLAB

Debugging

- Printf inside a CUDA Kernel
  - Devices with compute capability 2.x or higher support calls to printf from within a CUDA kernel
  - The in-kernel printf() function behaves in a similar way to the standard C-library printf() function
  - An important thing to note is that every CUDA thread will call printf
  - It is up to the programmer to limit the output to a single thread if only a single output string is desired

```
#include <stdio.h>

__global__ void print_kernel() {
    printf("Hello from block %d, thread %d\n", blockIdx.x, threadIdx.x);
}

int main() {
    print_kernel<<<10, 10>>>();
    cudaDeviceSynchronize();
}
```

Debugging

- Printf inside a CUDA Kernel
  - It's generally a good idea to limit the number of threads calling printf to avoid getting spammed

```
if (threadIdx.x == 0) {
    printf(...);
}
```

  - Printf output is stored in a circular buffer of a fixed size. If the buffer fills, old output will be overwritten
  - The buffer's size defaults to 1MB and can be configured with `cudaDeviceSetLimit(cudaLimitPrintfFifoSize, size_t size)`
  - This buffer is flushed only for
    - the start of a kernel launch
    - synchronization (e.g. cudaDeviceSynchronize())
    - blocking memory copies (e.g. cudaMemcpy(...))
    - context destruction

Debugging

- **Cuda-memcheck:**
  - Stand-alone run-time error checker tool
  - Detects memory errors like stack overflow
  - Same spirit as valgrind
  - No need to recompile the application
  - Not all the error reports are precise
  - Once used within cuda-gdb, the kernel launches are blocking

- Integrated in CUDA-GDB
  - More precise errors when used from CUDA-GDB
  - Must be activated before the application is launched

    (cuda-gdb) set cuda memcheck on

    Similar flag in Visual Studio

Debugging

- **Cuda-memcheck (Errors):**
  - Illegal global address
  - Misaligned global address
  - Stack memory limit exceeded
  - Illegal shared/local address
  - Misaligned shared/local address
  - Instruction accessed wrong memory
  - PC set to illegal value
  - Illegal instruction encountered
  - Illegal global address

Debugging

– **Example:**

(cuda-gdb) set cuda memcheck on

(cuda-gdb) run [Launch of CUDA Kernel 0 (applyStencil1D) on Device 0]
Program received signal CUDA_EXCEPTION_1, Lane Illegal Address.
applyStencil1D<<<>>> at stencil1d.cu:60

(cuda-gdb) info line stencil1d.cu:60 out[i] += weights[j + RADIUS] * in[i + j];

Debugging Tips

Always check the return code of the CUDA API routines

```
inline
cudaError_t checkCuda(cudaError_t result)
{
#if defined(DEBUG) || defined(_DEBUG)
      if (result != cudaSuccess) {
            fprintf(stderr, "CUDA Runtime Error: %sn",
            cudaGetErrorString(result));
            assert(result == cudaSuccess);
      }
#endif
return result;
}

…

checkCuda( cudaMemcpy(. . . ) )
```

Debugging Tips

- Use printf from the device code
  - make sure to synchronize so that buffers are flushed
- Repro with a debug build
  - Compile your app with –g –G
- Performance Issues???
  - Use the visual profiler

Profiling

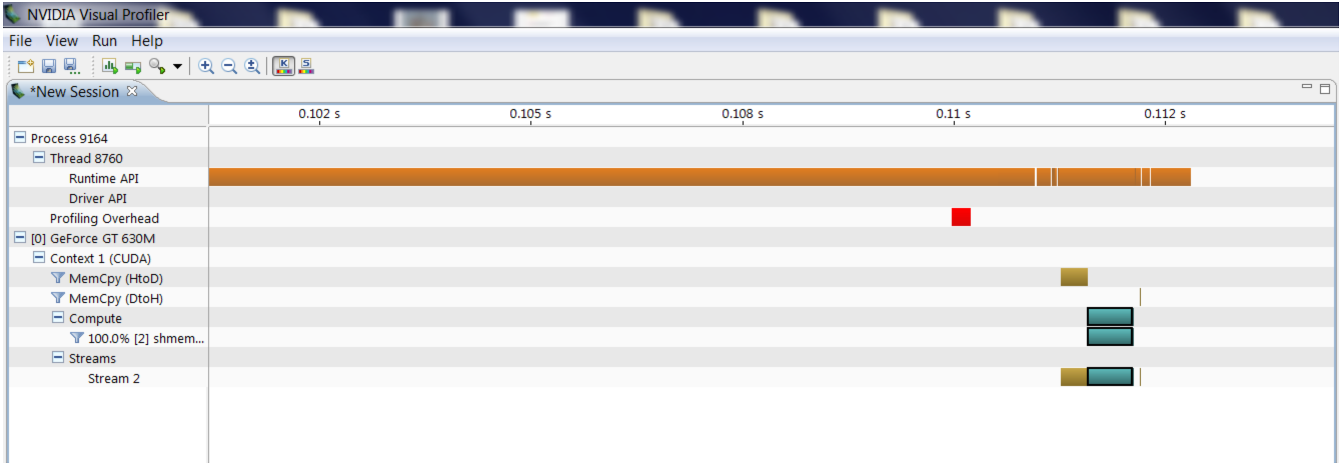
- Analysis
  - Although we talk about high-performance computing, it is not always easy to achieve this, many times we are left with just slower runtimes
  - Often, we can't find out where is our application bottleneck, or what part of the code is consuming more time.
  - **Need for a kernel and code analysis tool developed for the GPU**

Profiling

- Compute Visual Profiler
  - Step-by-step optimization guidance
  - Optimization opportunities
  - Analysis system estimates which kernels are best candidates for speedup
    - Execution time, achieved occupancy
  - Primary performance limiters
    - Memory bandwidth, compute resources, instruction/memory latency
  - Memory bandwidth utilization/Compute resource utilization
  - Transfers runtime
  - Resources usage
  - Occupancy / Divergence
  - Coalesced acceses
  - …

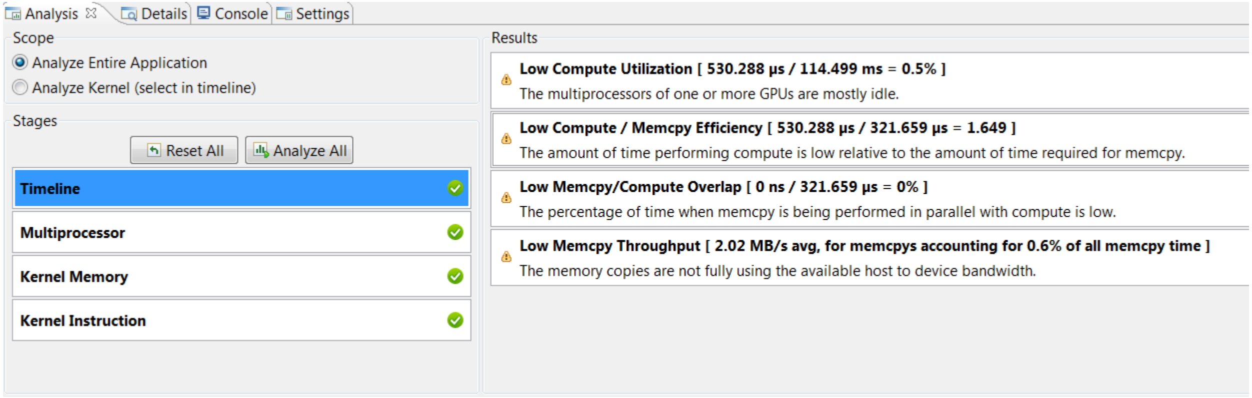
Profiling

– Analysis
  – Compute Visual Profiler: Timeline



Profiling

- Analysis
  - Multi-level analysis: multiprocessor, kernel, memory, instructions, etc.
  - Tips / suggestions to improve code's performance



Profiling

- Analysis
  - Detailed reports

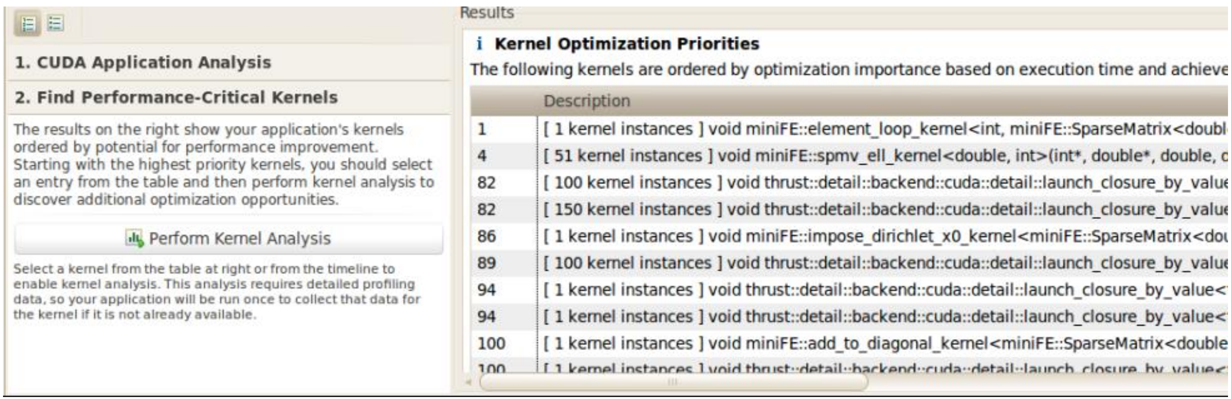
Properties | Detail Graphs

shmem_reduce_kernel(float*, float const *)

Name	Value
Start	111.584 ms
End	112.111 ms
Duration	526.497 µs
Grid Size	[512,1,1]
Block Size	[512,1,1]
Registers/Thread	7
Shared Memory/Block	2 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	12.5%
Local Memory Overhead	0%
DRAM Utilization	7.7% (2.36 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	0%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	0%
Global Cache Replay Overhead	0.8%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	95.1%
Theoretical	100%
L1 Cache Configuration	
Shared Memory Requested	48 KB
Shared Memory Executed	48 KB

Profiling

- Guided Analysis
  - Based on identifying primary performance limiter
  - Analysis system estimates which kernels are best candidates for speedup: execution time, achieved occupancy



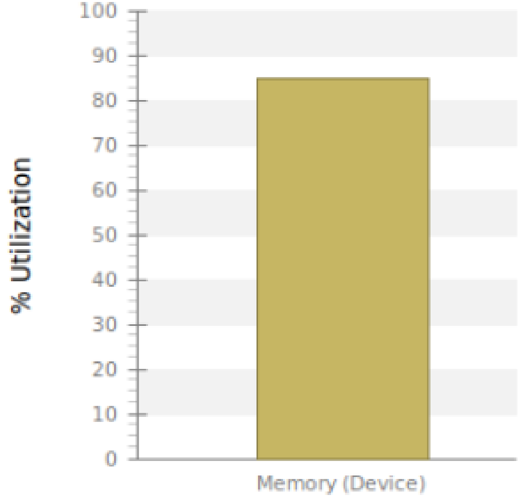
Profiling

- Primary Performance Limiter
  - Memory bandwidth
  - Compute resources
  - Instruction and memory latency

- Calculating Performance Limiter
  - Memory bandwidth utilization
  - Compute resource utilization
  - One of them high utilization value : likely performance limiter

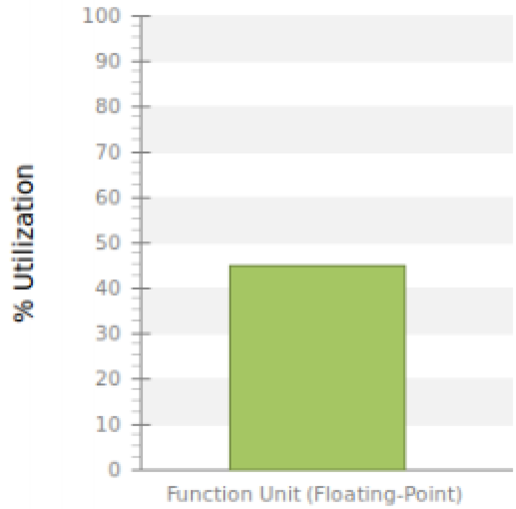
Profiling

- 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)

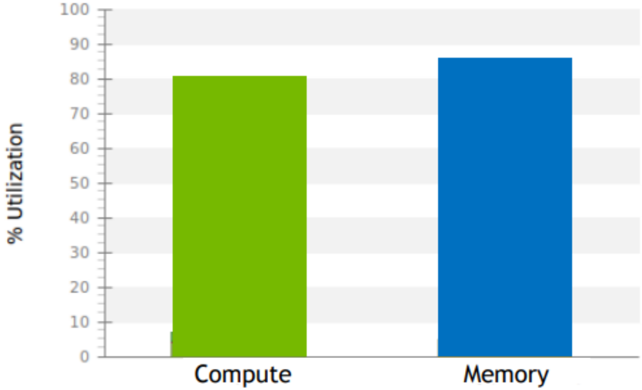


Profiling

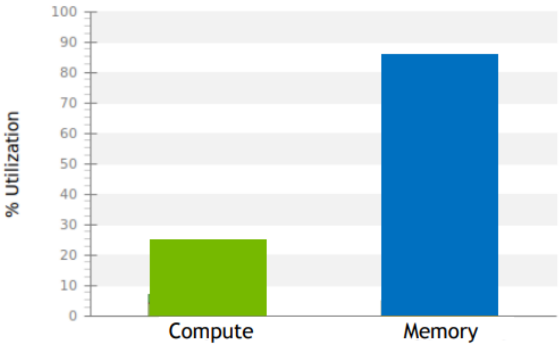
- Number of instructions issued, relative to peak capabilities of GPU
  - Some resources shared across all instructions
  - Some resources specific to instruction "classes": integer, FP, control-flow, etc.
  - Maximum utilization of any resource



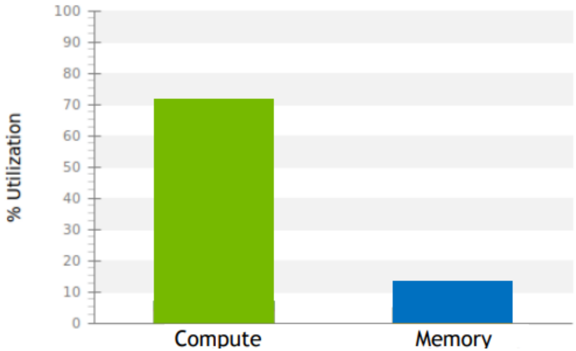
Profiling



– Both high:
compute and memory highly utilized



memory bandwidth limited



Compute resource limited

Profiling

- **nvprof**
  - command-line profiler available for Linux, Windows, and OS X
  - Maximum utilization of any memory subsystem
  - light-weight profiler
  - Default: nvprof presents an overview of the GPU kernels and memory copies in your application.
  - By using the `--output-profile` command-line option, you can output a data file for later import into either nvprof or the NVIDIA Visual Profiler
  - Enable profiling kernels written in any language (OpenAAC, CUDA Python, …)

```
==9261== Profiling application: ./tHogbomCleanHemi
==9261== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 58.73%  737.97ms      1000  737.97us  424.77us  1.1405ms  subtractPSFLoop_kernel(float const *, int, float*, int, int, int,
 38.39%  482.31ms      1001  481.83us  475.74us  492.16us  findPeakLoop_kernel(MaxCandidate*, float const *, int)
  1.87%  23.450ms         2  11.725ms  11.721ms  11.728ms  [CUDA memcpy HtoD]
  1.01%  12.715ms      1002  12.689us  2.1760us  10.502ms  [CUDA memcpy DtoH]
```

Profiling

- **nvprof**
  - command-line profiler available for Linux, Windows, and OS X
  - Matrix transpose

```
==15692== Profiling application: .\matrixTrans.exe
==15692== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 89.91%  174.84ms         1  174.84ms  174.84ms  174.84ms  [CUDA memcpy HtoD]
  9.50%  18.482ms         1  18.482ms  18.482ms  18.482ms  [CUDA memcpy DtoH]
  0.59%  1.1475ms         1  1.1475ms  1.1475ms  1.1475ms  smem_cuda_transpose(int, double const *, double*)
  0.00%  1.0560us         2     528ns     512ns     544ns  [CUDA memset]

==15692== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 55.11%  96.544ms         2  48.272ms  4.8782ms  91.666ms  cudaMalloc
 21.83%  38.241ms         2  19.121ms  19.091ms  19.150ms  cudaMemcpy
 20.31%  35.583ms         1  35.583ms  35.583ms  35.583ms  cudaDeviceReset
  1.29%  2.2522ms         2  1.1261ms  1.1122ms  1.1401ms  cudaFree
  0.68%  1.1867ms         1  1.1867ms  1.1867ms  1.1867ms  cudaDeviceSynchronize
  0.45%  782.79us        91  8.6020us       0ns  378.31us  cuDeviceGetAttribute
  0.14%  236.66us         2  118.33us  19.627us  217.03us  cudaEventSynchronize
  0.11%  187.73us         1  187.73us  187.73us  187.73us  cuDeviceGetName
  0.04%  71.111us         2  35.555us  17.351us  53.760us  cudaMemset
  0.02%  30.436us         1  30.436us  30.436us  30.436us  cudaLaunch
  0.01%  25.315us         4  6.3280us  2.5600us  15.360us  cudaEventRecord
  0.01%  21.049us         2  10.524us  10.524us  10.525us  cudaEventElapsedTime
  0.00%  8.2490us         2  4.1240us  1.1370us  7.1120us  cudaEventCreate
  0.00%  6.2570us         1  6.2570us  6.2570us  6.2570us  cuDeviceTotalMem
  0.00%  1.9930us         3     664ns     285ns  1.4230us  cuDeviceGetCount
  0.00%  1.9910us         3     663ns     284ns  1.1380us  cudaSetupArgument
  0.00%  1.4230us         1  1.4230us  1.4230us  1.4230us  cudaConfigureCall
  0.00%  1.1370us         3     379ns     284ns     569ns  cuDeviceGet
PS C:\Users\sergio\Documents\Visual Studio 2015\Projects\matrixTrans\Release>
```

References

Further reading

http://on-demand.gputechconf.com/gtc/2013/webinar/gtc-express-guided-analysis-nvidia-visual-profiler.pdf

https://devblogs.nvidia.com/cudacasts-episode-19-cuda-6-guided-performance-analysis-visual-profiler/

http://developer.download.nvidia.com/GTC/PDF/1062_Satoor.pdf

https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/

CUDA Profiling

Profiling & Debugging tools

Thanks for your attention!

Slides done in collaboration with Sergio Orts-Escolano!

Albert García García

agarcia @ dtic.ua.es