Developing, Debugging, and Optimizing GPU Codes for High Performance Computing with Allinea Forge

Ryan Hulguin
Applications Engineer
ryan.hulguin@arm.com
Agenda

• Introduction
• Overview of Allinea Products
• GPU Demonstration Examples
• Q&A
As of December 2016, Allinea is part of ARM

Our objective:
Remain the trusted leader in cross platform HPC tools

The same successful team...
• We will continue to work with our customers, partners and you!

... is stronger than ever...
• We can now respond quicker and deliver our roadmap faster

... as committed as ever...
• We remain 100% committed to providing cross-platforms tools for HPC

... and looking forward to the future.
• We are working with vendors to support the next generations of systems.
Where to find Allinea’s tools

Over 85% of Top 100 HPC systems
- From small to very large tools provision

8 of the Top 10 HPC systems
- Up to 700,000 core tools usage

Future leadership systems
- Millions of cores usage
Allinea: Industry Standard Tools for HPC

(and hundreds more)
Allinea toolkits save users’ and developers’ time

Allinea DDT (debugging)

Allinea MAP (profiling)
Analyze and tune application performance

A single-page report on application performance for users and administrators

Identify configuration problems and resource bottlenecks immediately

Track mission-critical performance over time and after system upgrades

Ensure key applications run at full speed on a new cluster or architecture

Summary: clover_leaf is CPU-bound in this configuration

CPU
80.6% Time spent running application code. High values are usually good.
This is high; check the CPU performance section for optimization advice.

MPI
19.4% Time spent in MPI calls. High values are usually bad.
This is low; this code may benefit from increasing the process count.

I/O
0.1% Time spent in filesystem I/O. High values are usually bad.
This is very low; however single-process I/O often causes large MPI wait times.

This application run was CPU-bound. A breakdown of this time and advice for investigating further is in the CPU section below. As little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU
A breakdown of the 80.6% CPU time:
Single-core code 0.4%
OpenMP regions 99.6%
Scalar numeric ops 42.4%
Vector numeric ops 4.0%
Memory accesses 53.6%
The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

I/O
A breakdown of the 0.1% I/O time:
Time in reads 0.0%
Time in writes 100.0%
Effective process read rate 0.00 bytes/s
Effective process write rate 411 kbytes/s

MPI
A breakdown of the 19.4% MPI time:
Time in collective calls 41.7%
Time in point-to-point calls 58.3%
Effective process collective rate 1.68 kbytes
Effective process point-to-point rate 24.5 kBytes
Most of the time is spent in point-to-point calls with a low transfer rate. This can be caused by inefficient message sizes, such as many small messages, or by imbalanced workloads causing processes to wait. The collective transfer rate is very low. This suggests load imbalance is causing synchronization overhead, use an MPI profiler to investigate further.

OpenMP
A breakdown of the 99.0% time in OpenMP regions:
Computation 100.0%
Synchronization 0.0%
Physical core utilization 200.0%
Intrinsics, report not available
Allinea DDT – The Debugger

• Who had a rogue behavior?
  – Merges stacks from processes and threads

• Where did it happen?
  – Leaps to source

• How did it happen?
  – Diagnostic messages
  – Some faults evident instantly from source

• Why did it happen?
  – Unique “Smart Highlighting”
  – Sparklines comparing data across processes
Allinea MAP – The Profiler

- Small data files
- <5% slowdown
- No instrumentation
- No recompilation
### How Allinea MAP is different

<table>
<thead>
<tr>
<th><strong>Adaptive sampling</strong></th>
<th>Sample frequency decreases over time</th>
<th>Data never grows too much</th>
<th>Run for as long as you want</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalable</strong></td>
<td>Same scalable infrastructure as Allinea DDT</td>
<td>Merges sample data at end of job</td>
<td>Handles very high core counts, fast</td>
</tr>
<tr>
<td><strong>Instruction analysis</strong></td>
<td>Categorizes instructions sampled</td>
<td>Knows where processor spends time</td>
<td>Shows vectorization and memory bandwidth</td>
</tr>
<tr>
<td><strong>Thread profiling</strong></td>
<td>Core-time not thread-time profiling</td>
<td>Identifies lost compute time</td>
<td>Detects OpenMP issues</td>
</tr>
<tr>
<td><strong>Integrated</strong></td>
<td>Part of Forge tool suite</td>
<td>Zoom and drill into profile</td>
<td>Profiling within your code</td>
</tr>
</tbody>
</table>
Enabling Performance Potential

- Use powerful tools easily
- Retrieve useful data
- Turn "a lot of" data into meaningful information
- Turn information into better code
Demonstration Examples

• The following examples are available through qwiklab

https://spl-nvlabs.qwiklab.com/focuses/preview/261?locale=en
Goals

• Generate and analyze a performance profile of CPU code
• Use debugger to track down and fix fatal GPU bug
• Use debugger to track down and fix nonfatal GPU bug
Preparing to Migrate from CPU to GPU

- Identify bottlenecks that may prevent migration from CPU to GPU
- Identify areas that are suitable for use on GPU
Matrix Multiplication Example

\[ C = A \times B + C \]

Master process

Slave process 1

Slave process n-1
Generating a MAP profile

- Run MAP from command line or from the GUI
Compute Analysis

```
void mmult(int size, int nslices, double *A, double *B, double *C)
{
    for(int i=0; i<size/nslices; i++)
        for(int j=0; j<size; j++)
            double res = 0.0;
            for(int k=0; k<size; k++)
                res += A[i*size+k]*B[k*size+j];
            C[i*size+j] += res;
}
```

Showing data from 17,256 samples taken over 24 processes (719 per process)
MPI Analysis

if(myrank == 0)
{
    printf("%d: Receiving result matrix...\n", myrank);
    for(int i=1; i<=proc; i++)
    {
        MPI_Recv( &mat_c[slice+i], slice, MPI_DOUBLE, i, 500+i, MPI_COMM_WORLD, &st);
    }
}
else
{
    printf("%d: Sending result matrix...\n", myrank);
    MPI_Send( &mat_c[0], slice, MPI_DOUBLE, 0, 500+myrank, MPI_COMM_WORLD);
}

if(myrank == 0) {...}

free(mat_a);
free(mat_b);
free(mat_c);

MPI_Finalize();

return 0;

Showing data from 18,024 samples taken over 24 processes (751 per process)
Next Steps

• The next example attempts to write a GPU kernel to perform the matrix multiplication, but introduces a fatal bug
• Allinea DDT can be used to track what is going wrong in this GPU kernel
Fatal Bug

Let’s smash this bug using Allinea DDT
A More Useful Error Message
Where Did Array A (in GPU Kernel) Come From?

- Using the Stacks view, we can see that array A comes from the array d_A in the mmult_cuda function.
How is $d_A$ Allocated?

The `mmult_cuda` function is run on the host.

$d_A$ is allocated on the GPU using `cudaMallocPitch`.

$d_A$ gets values from host array $A$ using `cudaMemcpy2D`.
What Does cudaMallocPitch do?

- **cudaMallocPitch** is the preferred method for allocating 2D arrays as it pads the data and aligns it for better performance.

  ```
  void mmult_cuda(int size, int nslices, double *A, double *B, double *C) {
    double *d_A, *d_B, *d_C;
    size_t pitch_A, pitch_B, pitch_C;
    int widthinbytes = size*sizeof(double);
    int height = size/nslices;
    // Allocate on device
    CUDA_CHECK(cudaMallocPitch(&d_A, &pitch_A, widthinbytes, height));
  }
  ```

- From the NVIDIA documentation, **pitch_A** is the length (in bytes) of the padded row for d_A.
- The allocation looks fine, we must be indexing it improperly.
Improper Indexing

- We learned from the previous slide that \texttt{pitch\_A} and \texttt{pitch\_B} are length in bytes.
- If we want the number elements for indexing purposes, we need to divide by the \texttt{sizeof(double)}.
Edit Within DDT

```c
mcpy2D(
   es,
   size, cudaMemcpyHostToDevice
mcpy2D(
   es,
   height, cudaMemcpyHostToDevice

// Set CUDA grid dimensions
    dim3 block(BLOCK_X, 1);
    dim3 grid((size+Block.x-1)/block.x,(height+block.y-1)/block.y); // allocate one more block on x or y if size%BLOCK_X!=0

    // Call Kernel
    mmult_kernel<<<grid,block>>>(size, nslices, d_A, pitch А/sizeof(double),
                                 d_B, pitch B/sizeof(double),
                                 d_C, pitch C/sizeof(double));
```
Smash that Bug

010001000100100

Note: Allinea DDT can only send input to the mpirun process with this MPI implementation
Further Optimization

- The next example attempts to improve performance further by moving data into shared GPU memory.

- This time a nonfatal bug is introduced where the solution is incorrect.

- Allinea DDT can help track this bug down.
Track Data Before and After Calculation Loop

- Click **Run to here** on the line right before the calculation is stored.
Set Parameters for Multi-Dimensional Array Viewer

- Modify subscripts \( i \) and \( j \) and place \( \$ \) in front of them
- Set the range from 0 to 63
- Click Evaluate
Select Block 1

- Select Thread 0 of Block 1 and Click Go
- Since $i=2$, we expect row 2 of the array to be updated
- Click Step Over to execute line 52
Multidimensional Array Viewer Shows Exact Changes

- Click Evaluate to update the array viewer
- Row 2 updated as expected
- Click Step Over again and update the array viewer
Wrong Row Updated

- It appears that we forgot a pair of parentheses at line 53
Correct the Instruction Used to Update the Array

- The behavior is now correct
- Let’s compare the performance of the optimized versions
Differences in Runtime

Timings were generated on a problem size of 7680 on Dual Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
Single Tesla K80
Great Things to Try with Allinea MAP

- Find the peak memory use
- Remove I/O bottleneck
- Make sure threads are well utilized
- Improve memory access
- Restructure for vectorization
- Add your own metrics to the MAP time based sampler
Great things to try with Allinea DDT

- The scalable print alternative
- Stop on variable change
- Static analysis warnings on code errors
- Detect read/write beyond array bounds
- Detect stale memory allocations
This session will be gathering major CUDA Developer Tools vendors, including NVIDIA and PGI to share their latest feature development.

David Lecomber - Senior Director, HPC Tools, ARM – will be taking part in this event

Tuesday, May 9, 2:00 PM - 4:00 PM – Hilton Market
Q&A and Wrap-up
Thank you!

Any questions, feel free to ask.