Applied Parallel Computing

parallel-computing.pro

OpenACC introduction (part 2)

Aleksei Ivakhnenko

Contents

- Understanding PGI compiler output
  - Compiler flags and environment variables
  - Compiler limitations in dependencies tracking
- Organizing data persistence regions using *data* directives
- Data clauses
- Profiling GPU kernels in OpenACC application
  - *Time* option
  - PGI_ACC_TIME environment variable
  - Profiling with pgprof.
- Hands-on:
  - “Fill-in” exercise on implementing wave propagation stencil in OpenACC (wave13pt).
  - Adding OpenACC directives step by step

PGI accelerator info

```
aivahnenko@tesla-cmc:~$ pgaccelinfo

CUDA Driver Version:           7000
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  346.46
Tue Feb 17 17:56:08 PST 2015

Device Number:                 0
Device Name:                   Tesla K40c
Device Revision Number:        3.5
Global Memory Size:            12884705280
Number of Multiprocessors:     15
Number of SP Cores:            2880
Number of DP Cores:            960
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    745 MHz
Execution Timeout:             No
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            Yes
ECC Enabled:                   No
Memory Clock Rate:             3004 MHz
Memory Bus Width:              384 bits
L2 Cache Size:                 1572864 bytes
Max Threads Per SMP:           2048
Async Engines:                 2
Unified Addressing:            Yes
Managed Memory:                Yes
PGI Compiler Option:           -ta=tesla:cc35
```

Compiler info flags

- -Minfo[=all|accel|ccff|ftn|inline|intensity|ipa|loop|lre|mp|opt|par|pfo|stat|time|unified|vect]-Generate informational messages about optimizations
  - all -Minfo=accel,inline,ipa,loop,lre,mp,opt,par,unified,vect
  - accel Enable Accelerator information
  - ccff Append information to object file
  - ftn Enable Fortran-specific information
  - inline Enable inliner information
  - intensity Enable compute intensity information
  - ipa Enable IPA information
  - loop Enable loop optimization information
  - lre Enable LRE information
  - mp Enable OpenMP information
  - opt Enable optimizer information
  - par Enable parallelizer information
  - pfo Enable profile feedback information
  - stat Same as -Minfo=time
  - time Display time spent in compiler phases
  - unified Enable unified binary information
  - vect Enable vectorizer information

Compiler profiling flags

- -Mprof[=[no]ccff|dwarf|func|lines|time|mpich|sgimpi|mpich1|mpich2|mvapich1] Generate additional code for profiling; implies -Minfo=ccff
  - [no]ccff Enable (disable) CCFF information
  - dwarf Add limited DWARF info sufficient for performance profilers
  - func Function-level profiling
  - lines Line-level profiling
  - time Sample-based instruction-level profiling
  - mpich Profile with default MPICH v3.0; implies -Mmpi=mpich
  - sgimpi Profile with default SGI MPI; implies -Mmpi=sgimpi
  - mpich1 DEPRECATED: Use MPIDIR for MPICH1; implies -Mmpi=mpich1
  - mpich2 DEPRECATED: Use MPIDIR for MPICH2; implies -Mmpi=mpich2
  - mvapich1 DEPRECATED: Use MPIDIR for MVAPICH1; implies -Mmpi=mvapich1

Compiler target flags

- -ta= Choose target accelerator
  - tesla — Select NVIDIA Tesla accelerator target
    - ✓ cc20 — Compile for compute capability 2.0
    - ✓ cc30 — Compile for compute capability 3.0
    - ✓ cc35 — Compile for compute capability 3.5
    - ✓ cc50 — Compile for compute capability 5.0
    - ✓ cuda6.5 — Use CUDA 6.5 Toolkit compatibility (default)
    - ✓ cuda7.0 — Use CUDA 7.0 Toolkit compatibility
    - ✓ fastmath — Use fast math library
    - ✓ [no]flushz — Enable flush-to-zero mode on the GPU
    - ✓ [no]fma — Generate fused mul-add instructions (default at -O3)
    - ✓ keepbin — Keep kernel .bin files
    - ✓ keepgpu — Keep kernel source files
    - ✓ keepptx — Keep kernel .ptx files
    - ✓ [no]lineinfo — Generate GPU line information
    - ✓ [no]llvm — Use LLVM back end
    - ✓ loadcache — Choose what hardware level cache to use for global memory loads
      - – L1 — Use L1 cache
      - – L2 — Use L2 cache
    - ✓ maxregcount:<n> — Set maximum number of registers to use on the GPU
    - ✓ pin — Set default to pin host memory
    - ✓ [no]rdc — Generate relocatable device code
    - ✓ [no]unroll — Enable automatic inner loop unrolling (default at -O3)
    - ✓ beta — Enable beta code generation features
  - nvidia — nvidia is a synonym for tesla

Environment variables

- ACC_NOTIFY
- PGI_ACC_TIME
- ACC_DEVICE_TYPE
- ACC_DEVICE_NUM

Dependency tracking

- PGI compiler tracks data dependencies
  - Generates output info for dependent data preventing parallel execution
    - ✓ Helps to find code regions which have to be rewritten
  - Possible misinterpreting of code leads to sequential execution
    - ✓ Separate compilation
    - ✓ C/C++ pointers do not provide array length info

Dependency example

dependency.cpp:

```
int n=1000;
float* a=(float*)malloc(n*sizeof(float));
float* b=(float*)malloc(n*sizeof(float));
float* c=(float*)malloc(n*sizeof(float));

for (int i=0; i<n; i++)
{
    a[i]=(float)rand()/RAND_MAX;
    b[i]=(float)rand()/RAND_MAX;
}

vecadd(a, b, c, n);

free (a);
free (b);
free (c);
```

Dependency example

dependency_vecadd.hpp:

```
void vecadd (float* a, float *b, float *c, int n);
```

dependency_vecadd.cpp:

```
void vecadd (float* a, float *b, float *c, int n)
{
        #pragma acc parallel
        for (int i=0; i<n; i++)
        {
                c[i]=a[i]+b[i];
        }
}
```

Compiler output

```
aivahnenko@tesla-cmc:~/scratch/openacc/dependency$ make
pgc++ dependency_vecadd.cpp -c -acc -Minfo=accel -ta=nvidia,time -o
dependency_vecadd.o
dependency_vecadd.cpp:
vecadd(float *, float *, float *, int):
      2, Accelerator kernel generated
         Generating Tesla code
         Generating copyout(c[:n])
         Generating copyin(a[:n],b[:n])
      4, Complex loop carried dependence of b->,a-> prevents parallelization
         Loop carried dependence of c-> prevents parallelization
         Loop carried backward dependence of c-> prevents vectorization
```

Resolved dependency example1

dependency_vecadd.hpp:

```
void vecadd (float* a, float *b, float *c, int n);
```

dependency_vecadd.cpp:

```
void vecadd (float* a, float *b, float *c, int n)
{
        #pragma acc parallel loop independent
        for (int i=0; i<n; i++)
        {
                c[i]=a[i]+b[i];
        }
}
```

Resolved dependency example2

dependency_vecadd.hpp:

```
void vecadd (float * restrict a, float * restrict b, float * restrict c, int n);
```

dependency_vecadd.cpp:

```
void vecadd (float * restrict a, float * restrict b, float * restrict c, int n)
{
        #pragma acc parallel
        for (int i=0; i<n; i++)
        {
                c[i]=a[i]+b[i];
        }
}
```

Resolved dependency example3

```
pgc++ dependency_vecadd.cpp -c -acc -
Minfo=accel -ta=nvidia,time -o
dependency_vecadd.o -Msafeptr
```

Compiler output

```
aivahnenko@tesla-cmc:~/scratch/openacc/dependency$ make
pgc++ dependency_vecadd.cpp -c -acc -Minfo=accel -ta=nvidia,time  -o
dependency_vecadd.o
resolved_dependency_vecadd1.cpp:
vecadd(float *, float *, float *, int):
      2, Accelerator kernel generated
         Generating Tesla code
          4, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
      2, Generating copyout(c[:n])
         Generating copyin(a[:n],b[:n])
```

Combined directives

Fortran

```
!$acc parallel loop [clause…]
       do loop
[!$acc end parallel loop]

!$acc kernels loop [clause…]
       do loop
[!$acc end kernels loop]
```

C

```
#pragma acc parallel loop
[clause…]
       for loop

#pragma acc kernels loop
[clause…]
       for loop
```

Arrays

Fortran
Syntax: array (begin: end)
Examples: a(:, :), a(1:100, 2:n)

C
Syntax: array [begin : length]
Examples: a[2:n] // a[2], a[3], …, a[2+n-1]

Remember that C and Fortran arrays indices are calculated differently.

Data clauses

Data clauses

- copy*(list)
- create(list)
- present(list)
- present_or_copy*(list)
- present_or_create(list)
- deviceptr(list)
- private(list)
- firstprivate(list)

*<blank>|in|out

Data region example

```
int a [1000];
int b [1000];

#pragma acc parallel
    for (int i=0; i<1000; i++)
    {
        a[i] = i – 100 + 23;
    }

#pragma acc parallel
    for (int j=0; j<1000; j++)
    {
        b[j] = a[j] – j – 10 + 213;
    }
```

Parallel directives are separated and this would require individual data copies for all kernels

Data region example

```
pgcc no_region.c -acc -Minfo=accel -ta=nvidia,time -o no_region
no_region.c:
main:
     10, Accelerator kernel generated
         Generating Tesla code
         11, #pragma acc loop vector(128) /* threadIdx.x */
     10, Generating copyout(a[:])
     11, Loop is parallelizable
     16, Accelerator kernel generated
         Generating Tesla code
         17, #pragma acc loop vector(128) /* threadIdx.x */
     16, Generating copyout(b[:])
         Generating copyin(a[:])
     17, Loop is parallelizable
```

Data region example

```
Accelerator Kernel Timing data
/scratch/aivahnenko/openacc/examples/data_regions/no_region.c
  main  NVIDIA  devicenum=0
    time(us): 97
    10: compute region reached 1 time
        10: kernel launched 1 time
            grid: [1]  block: [128]
             device time(us): total=5 max=5 min=5 avg=5
            elapsed time(us): total=594 max=594 min=594 avg=594
    10: data region reached 1 time
    16: compute region reached 1 time
        16: kernel launched 1 time
            grid: [1]  block: [128]
             device time(us): total=7 max=7 min=7 avg=7
            elapsed time(us): total=58 max=58 min=58 avg=58
    16: data region reached 2 times
        16: data copyin transfers: 1
             device time(us): total=19 max=19 min=19 avg=19
        16: data copyout transfers: 1
             device time(us): total=51 max=51 min=51 avg=51
    21: data region reached 1 time
        21: data copyout transfers: 1
             device time(us): total=15 max=15 min=15 avg=15
```

Copying time 19+51+15
85

Data region example

```
int a [1000];
int b [1000];
#pragma acc data copyout (a[0:1000],b[0:1000])
    {
    #pragma acc parallel
        for (int i=0; i<1000; i++)
        {
            a[i]=i-100+23;
        }

    #pragma acc parallel
        for (int j=0; j<1000; j++)
        {
            b[j]=a[j]-j-10+213;
        }
}
```

Parallel directives are inside the data region. This would require only one data transfer

Data region example

```
pgcc region.c -acc -Minfo=accel -ta=nvidia,time -o region
region.c:
main:
      9, Generating copyout(a[:],b[:])
     11, Accelerator kernel generated
         Generating Tesla code
         12, #pragma acc loop vector(128) /* threadIdx.x */
         Loop is parallelizable
     17, Accelerator kernel generated
         Generating Tesla code
         18, #pragma acc loop vector(128) /* threadIdx.x */
         Loop is parallelizable
```

Data region example

```
Accelerator Kernel Timing data
/scratch/aivahnenko/openacc/examples/data_regions/region.c
  main  NVIDIA  devicenum=0
    time(us): 76
    9: data region reached 1 time
    11: compute region reached 1 time
        11: kernel launched 1 time
            grid: [1]  block: [128]
             device time(us): total=6 max=6 min=6 avg=6
            elapsed time(us): total=547 max=547 min=547 avg=547
    17: compute region reached 1 time
        17: kernel launched 1 time
            grid: [1]  block: [128]
             device time(us): total=7 max=7 min=7 avg=7
            elapsed time(us): total=35 max=35 min=35 avg=35
    23: data region reached 1 time
        23: data copyout transfers: 2
             device time(us): total=63 max=52 min=11 avg=31
```

Copying time 63
Lower by 26%

Profiling

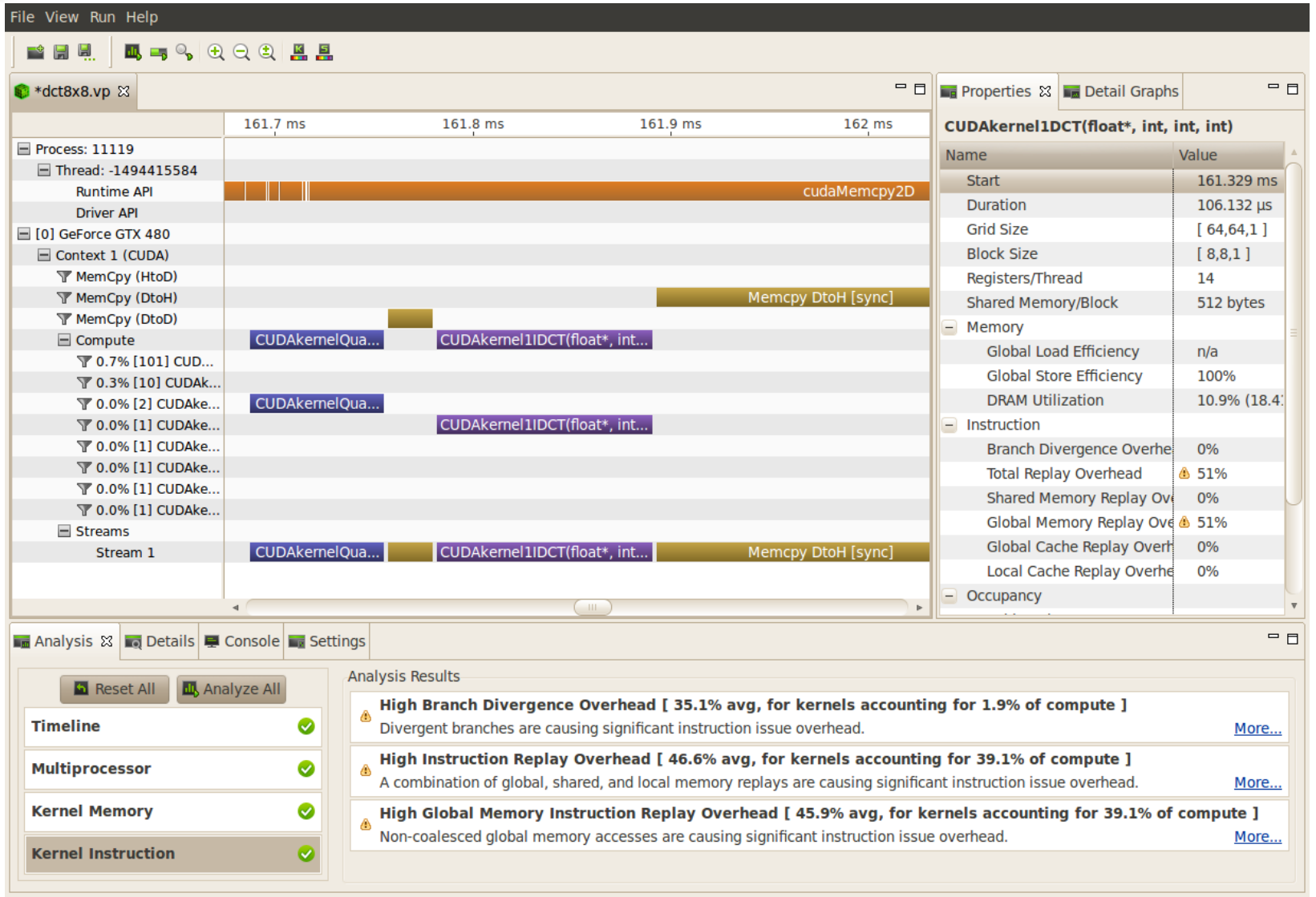
- Profiling code gives you a solid understanding of what is going on
  - Most heavy regions, bottlenecks
- Different profilers to use with CUDA
  - pgprof
  - nvprof
  - nvvp
  - Nvidia command-line profiler
  - *time* compiler option
  - PGI_ACC_TIME environment variable

nvprof

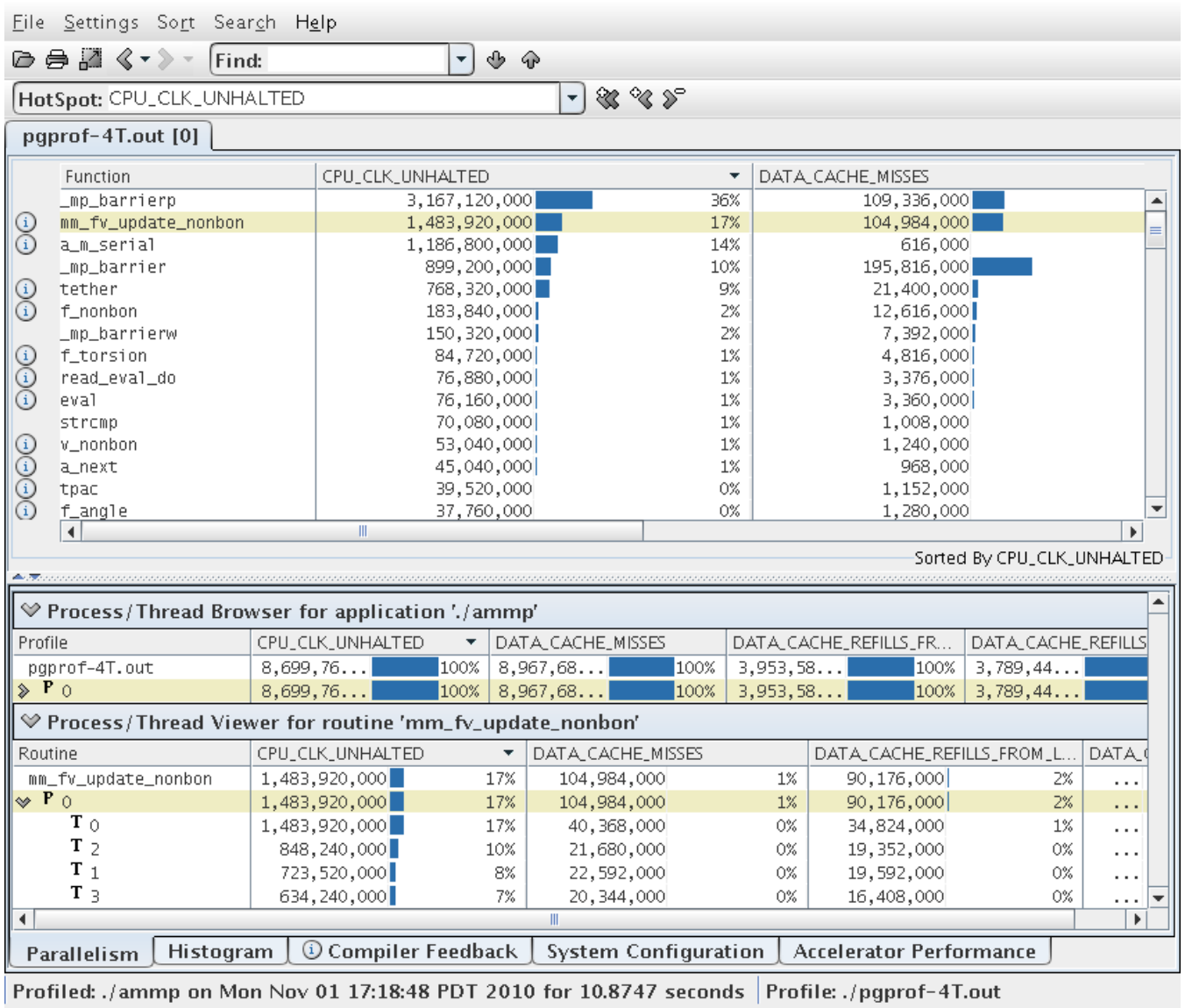
```
==8827== Profiling application: ./resolved_dependency1
==8827== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 62.70%  11.778us         2  5.8890us  5.5690us  6.2090us  [CUDA memcpy HtoD]
 23.51%  4.4160us         1  4.4160us  4.4160us  4.4160us  [CUDA memcpy DtoH]
 13.80%  2.5920us         1  2.5920us  2.5920us  2.5920us  _Z6vecaddPfS_S_i_2_gpu

==8827== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 85.84%  386.32ms         1  386.32ms  386.32ms  386.32ms  cuCtxCreate
 13.56%  61.040ms         1  61.040ms  61.040ms  61.040ms  cuMemHostAlloc
  0.24%  1.0889ms         1  1.0889ms  1.0889ms  1.0889ms  cuMemAllocHost
  0.21%  941.06us         4  235.26us  9.8680us  479.28us  cuMemAlloc
  0.05%  239.55us         1  239.55us  239.55us  239.55us  cuModuleLoadData
  0.02%  78.488us         9  8.7200us  3.0380us  30.140us  cuEventRecord
  0.02%  72.419us         2  36.209us  20.628us  51.791us  cuMemcpyHtoDAsync
  0.01%  67.319us         1  67.319us  67.319us  67.319us  cuStreamCreate
  0.01%  65.466us         1  65.466us  65.466us  65.466us  cuLaunchKernel
  0.01%  42.055us         5  8.4110us  1.5540us  17.727us  cuEventSynchronize
  0.00%  21.269us         1  21.269us  21.269us  21.269us  cuMemcpyDtoHAsync
  0.00%  14.038us         4  3.5090us  1.0750us  8.6130us  cuEventCreate
  0.00%  11.670us         8  1.4580us     526ns  4.5340us  cuDeviceGet
  0.00%  10.901us         4  2.7250us  2.1980us  3.8630us  cuEventElapsedTime
  0.00%  10.766us         2  5.3830us  1.0050us  9.7610us  cuDeviceGetCount
  0.00%  10.668us         1  10.668us  10.668us  10.668us  cuModuleGetFunction
  0.00%  10.495us         3  3.4980us  2.0350us  6.0010us  cuStreamSynchronize
  0.00%  5.1980us         4  1.2990us     367ns  3.3090us  cuDeviceComputeCapability
  0.00%  2.4950us         1  2.4950us  2.4950us  2.4950us  cuDriverGetVersion
  0.00%  2.1270us         2  1.0630us     587ns  1.5400us  cuCtxGetCurrent
  0.00%  1.1050us         1  1.1050us  1.1050us  1.1050us  cuCtxSetCurrent
  0.00%     931ns         1     931ns     931ns     931ns  cuCtxAttach
```

NVVP


File View Run Help
*dct8x8.vp
161.7 ms
161.8 ms
161.9 ms
162 ms
Process: 11119
Thread: -1494415584
Runtime API
cudaMemcpy2D
Driver API
[0] GeForce GTX 480
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Memcpy DtoH [sync]
MemCpy (DtoD)
Compute
CUDAkernelQua...
CUDAkernel1IDCT(float*, int...
0.7% [101] CUD...
0.3% [10] CUDAk...
0.0% [2] CUDAke...
0.0% [1] CUDAke...
Streams
Stream 1
Properties
Detail Graphs
CUDAkernel1DCT(float*, int, int, int)
Name
Value
Start
161.329 ms
Duration
106.132 µs
Grid Size
[64,64,1]
Block Size
[8,8,1]
Registers/Thread
14
Shared Memory/Block
512 bytes
Memory
Global Load Efficiency
n/a
Global Store Efficiency
100%
DRAM Utilization
Instruction
Branch Divergence Overhe
0%
Total Replay Overhead
51%
Shared Memory Replay Ov
Global Memory Replay Ove
Global Cache Replay Overh
Local Cache Replay Overhe
Occupancy
Analysis
Details
Console
Settings
Reset All
Analyze All
Timeline
Multiprocessor
Kernel Memory
Kernel Instruction
Analysis Results
High Branch Divergence Overhead [35.1% avg, for kernels accounting for 1.9% of compute]
Divergent branches are causing significant instruction issue overhead.
More...
High Instruction Replay Overhead [46.6% avg, for kernels accounting for 39.1% of compute]
A combination of global, shared, and local memory replays are causing significant instruction issue overhead.
More...
High Global Memory Instruction Replay Overhead [45.9% avg, for kernels accounting for 39.1% of compute]
Non-coalesced global memory accesses are causing significant instruction issue overhead.
More...

pgprof


File Settings Sort Search Help
Find:
HotSpot: CPU_CLK_UNHALTED
pgprof-4T.out [0]
Function | CPU_CLK_UNHALTED | DATA_CACHE_MISSES
_mp_barrierp 3,167,120,000 36% 109,336,000
mm_fv_update_nonbon 1,483,920,000 17% 104,984,000
a_m_serial 1,186,800,000 14% 616,000
_mp_barrier 899,200,000 10% 195,816,000
tether 768,320,000 9% 21,400,000
f_nonbon 183,840,000 2% 12,616,000
_mp_barrierw 150,320,000 2% 7,392,000
f_torsion 84,720,000 1% 4,816,000
read_eval_do 76,880,000 1% 3,376,000
eval 76,160,000 1% 3,360,000
strcmp 70,080,000 1% 1,008,000
v_nonbon 53,040,000 1% 1,240,000
a_next 45,040,000 1% 968,000
tpac 39,520,000 0% 1,152,000
f_angle 37,760,000 0% 1,280,000
Sorted By CPU_CLK_UNHALTED
Process/Thread Browser for application './ammp'
Profile | CPU_CLK_UNHALTED | DATA_CACHE_MISSES | DATA_CACHE_REFILLS_FR... | DATA_CACHE_REFILLS
pgprof-4T.out 8,699,76... 100% 8,967,68... 100% 3,953,58... 100% 3,789,44...
P 0 8,699,76... 100% 8,967,68... 100% 3,953,58... 100% 3,789,44...
Process/Thread Viewer for routine 'mm_fv_update_nonbon'
Routine | CPU_CLK_UNHALTED | DATA_CACHE_MISSES | DATA_CACHE_REFILLS_FROM_L... | DATA_
mm_fv_update_nonbon 1,483,920,000 17% 104,984,000 1% 90,176,000 2%
P 0 1,483,920,000 17% 104,984,000 1% 90,176,000 2%
T 0 1,483,920,000 17% 40,368,000 0% 34,824,000 1%
T 2 848,240,000 10% 21,680,000 0% 19,352,000 0%
T 1 723,520,000 8% 22,592,000 0% 19,592,000 0%
T 3 634,240,000 7% 20,344,000 0% 16,408,000 0%
Parallelism
Histogram
Compiler Feedback
System Configuration
Accelerator Performance
Profiled: ./ammp on Mon Nov 01 17:18:48 PDT 2010 for 10.8747 seconds
Profile: ./pgprof-4T.out

Mapping to CUDA blocks and threads

- Inlying loops generate multi-dimensional blocks and grids
  - In general: gang -> blocks, vector -> threads
- Example: 2D loop -> grid[100x200], block[16x32]

```
#pragma acc kernels loop gang(100), vector(16)
   for( … )
      #pragma acc loop gang(200), vector(32)
         for( … )
```

100 blocks in a row (Y direction)

16 threads in a row

200 blocks in a column (X direction)

32 threads in a column

Mapping to CUDA blocks and threads

```
#pragma acc kernels
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];

#pragma acc kernels loop gang(100) vector(128)
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];

#pragma acc parallel num_gangs(100) vector_length(128)
{
    #pragma acc loop gang vector
        for (int i = 0; i < n; i++)
            y[i] += a*x[i];
}
```

n/32 blocks, 32 threads each by default

100 blocks of 128 threads, each thread executes one iteration of a loop with kernels directive

100 blocks of 128 threads, each thread executes one iteration of a loop with parallel directive

Mapping to CUDA blocks and threads

```
#pragma acc parallel num_gangs(100)
{
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
}
#pragma acc parallel num_gangs(100)
{
    #pragma acc loop gang
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
}
```

100 blocks, 32 threads each by default

100 blocks, 32 threads each by default

Mapping to CUDA blocks and threads

Each thread can execute one or more iterations of the loop. This depends on the mapping parameters. Executing several iterations can increase the performance by lowering the initialization time

```
#pragma acc kernels loop gang(100) vector(128)
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
```

100 blocks, 128 threads each

```
#pragma acc kernels loop gang(50) vector(128)
    for (int i = 0; i < n; i++)
        y[i] += a*x[i];
```

50 blocks, 128 threads each