A Tool for Performance Analysis of GPU-Accelerated Applications

Keren Zhou, John Mellor-Crummey
Department of Computer Science
Rice University
Problem

- OpenMP Target, Kokkos, and RAJA generate sophisticated GPU code with many small procedures
  - Complex calling contexts on both CPU and GPU
- Existing performance tools are ill-suited for analyzing such complex kernels because they lack a comprehensive profile view
- At best existing tools only attribute runtime cost to a flat profile view of functions executed on GPUs
Key contribution

- A novel measurement system builds a complete profile view to show performance metrics for GPU-accelerated code for multiple CPU threads
  - Construct calling context trees for GPU programs by analyzing control flow and call graphs
  - Employ \textit{wait-free} data structures to attribute GPU samples back to heterogeneous calling contexts
  - Apportion GPU samples to calling contexts using instruction samples of GPU function calls
Start from a simple application

Two OpenMP threads launch \textit{vecAdd} kernels concurrently

```
#omp parallel num_threads(2)
    cuLaunchKernel(vecAdd, ...)

int __noinline__ add(int a, int b) {
    return a + b;
}

void vecAdd(int *l, int *r, int *p, size_t iter1, size_t iter2) {
    size_t idx = blockDim.x * blockIdx.x + threadIdx.x;
    for (size_t i = 0; i < iter1; ++i) {
        p[idx] = add(l[idx], r[idx]);
    }
    for (size_t i = 0; i < iter2; ++i) {
        p[idx] = add(l[idx], r[idx]);
    }
}
```
nvvp lacks of calling context

A tool should attribute latencies back to call sites at line 12 and line 15
nvvp lacks of control flow analysis

A tool should attribute performance to loops
A complete profile view

```
__device__
int  __attribute__((noinline))  add(int a, int b) {
    return a + b;
}

extern "C"
__global__
void vecAdd(int *l, int *r, int *p, size_t N, size_t iter1, size_t iter2) {
    size_t idx = blockDim.x * blockIdx.x + threadIdx.x;
    for (size_t i = 0; i < iter1; ++i) {
        p[idx] = add(l[idx], r[idx]);
    }
```

Samples

- `loop at vecAdd.cu: 14`  1.07e+07  60.3%
- `loop at vecAdd.cu: 11`  5.26e+06  29.6%
- `vecAdd.cu: 12`  8.71e+05  4.9%
- `vecAdd.cu: 12: $vecAdd$.Z3addii`  6.95e+05  3.9%
- `vecAdd.cu: 3`  6.17e+05  3.5%
- `vecAdd.cu: 17`  7.78e+04  0.4%
Step 1: Build calling context tree on CPU

- Use HPCToolkit’s CCT-tree

```c
#pragma omp parallel num_threads(2)
  cuLaunchKernel(vecAdd, ...)

int __noinline__ add(int a, int b) {
  return a + b;
}

void vecAdd(int *l, int *r, int *p, size_t iter1, size_t iter2) {
  size_t idx = blockDim.x * blockIdx.x + threadIdx.x;
  for (size_t i = 0; i < iter1; ++i) {
    p[idx] = add(l[idx], r[idx]);
  }
  for (size_t i = 0; i < iter2; ++i) {
    p[idx] = add(l[idx], r[idx]);
  }
}
```

main

```c
#pragma omp parallel spawn
  cuLaunchKernel
cuLaunchKernel
```
Step 2: Apply static control flow analysis

- Identify loops

```c
#pragma omp parallel num_threads(2)
cuLaunchKernel(vecAdd, ...)

int __noinline__ add(int a, int b) {
    return a + b;
}

void vecAdd(int *l, int *r, int *p, size_t iter1, size_t iter2) {
    size_t idx = blockDim.x * blockIdx.x + threadIdx.x;
    for (size_t i = 0; i < iter1; ++i) {
        p[idx] = add(l[idx], r[idx]);
    }
    for (size_t i = 0; i < iter2; ++i) {
        p[idx] = add(l[idx], r[idx]);
    }
}
```

main

#pragma omp parallel spawn

cuLaunchKernel

cuLaunchKernel

vecAdd

LOOP0 LOOP1
Step 3: Collect GPU samples

- Two categories of threads
  - Worker threads
    - Launch kernels, move and allocate data, synchronize GPU calls
  - CUPTI thread
    - Collect GPU samples

- Interaction
  - **Notification**: A worker thread T creates a notification record when it launches a kernel and tags the kernel with a correlation ID C, notifying the CUPTI thread that C belongs to T
  - **Sample attribution**: The CUPTI thread collects samples associated with C and communicates sample attribution records back to thread T
Sample attribution as an example

- The CUPTI thread adds samples to sample attribution queues using a **push (CAS)** operation. Each worker thread **steals (XCHG)** the head of its sample queue with NULL to steal all its records.

- **Wait-free progress** is guaranteed because a CUPTI thread’s CAS fails at most once when tries to add samples.

- **Memory reclamation** occurs when a worker thread’s samples have been attributed to its calling context tree. The worker puts records into a free queue which can be swapped by the CUPTI thread.
Step 4: Attribute GPU samples

• Attribute samples to function calls

```c
#pragma omp parallel num_threads(2)
    cuLaunchKernel(vecAdd, ...)

int __noinline__ add(int a, int b) {
    return a + b;
}

void vecAdd(int *l, int *r, int *p, size_t iter1, size_t iter2) {
    size_t idx = blockDim.x * blockIdx.x + threadIdx.x;
    for (size_t i = 0; i < iter1; ++i) {
        p[idx] = add(l[idx], r[idx]);
    }
    for (size_t i = 0; i < iter2; ++i) {
        p[idx] = add(l[idx], r[idx]);
    }
}
```

Approximate a calling context tree

- **Problem**
  - High cost to unwind call stacks on GPU

- **Solution**
  - Construct a call graph by parsing call instructions and linking corresponding procedures
  - Create “supernode” for recursive procedures
  - Split the call graph into a calling context tree
  - Apportion samples of procedures that have multiple call sites
Apportion samples of a procedure based on its call sites

**GPU Static Call Graph**

- Procedure P1
  - 10 Samples
- Procedure P2
  - 10 Samples
  - 4 Calls
- Procedure P3
  - 10 Samples
  - 6 Calls
- Procedure P4
  - 4 Calls
  - 6 Calls
  - \( \text{Ratio}(P_2, P_4) = \frac{4}{4+6} = 0.4 \)
  - 10 Samples
  - \( \text{Ratio}(P_3, P_4) = \frac{6}{4+6} = 0.6 \)

**GPU Calling Context Tree**

- Procedure P1
  - 10 Samples
- Procedure P2
  - 4 Calls
  - 10 Samples
- Procedure P3
  - 6 Calls
  - 10 Samples
- Procedure P4'
  - 4 Samples
- Procedure P4''
  - 6 Samples

3/19/2019
RAJA

• Template-based programming model based on C++
• Loop template can map a C++ lambda function for an iteration onto GPUs using CUDA
• RAJA performance suite
  • Explores performance of 30 loop-based computational kernels
  • https://github.com/LLNL/RAJAPerf
Profile rajaperf

Inline function
Loop
Hotspot
Status and ongoing work

• We extended HPCToolkit to build a complete profile view for analyzing the runtime characteristics of GPU-accelerated applications

• Work in progress
  • Collect all the performance information, including kernel performance, data movement, compute utilization, and PC sampling information in a single phase
  • Study MPI-based GPU-accelerated applications
vhalf = Real_t(1.) / (Real_t(1.) + compHalfStep);

if ( delvc > Real_t(0.) ) {
    q_new /* = qq_old[i] = qL_old[i] */ = Real_t(0.);
} else {
    ssc = ( pbvc * e_new + vhalf * vhalf * bvc * pHalfStep ) / rho0;

    if ( ssc <= Real_t(.1111111e-36) ) {
        ssc = Real_t(.3333333e-18);
    } else {
        ssc = SQRT(ssc);
    }

    q_new = (sssc*qL_old + qq_old);
}

e_new = e_new + Real_t(0.5) * delvc
    *(Real_t(3.0)*(p_old + q_old)
    + Real_t(1.0)*(pbvc + pHalfStep + p_old));
Template GPU

Procedures above

Actual Kernel Code