Basic C/C++ Profiling and Debugging Tools

ECE 6397

Electrical and Computer Engineering
University of Houston
Dr. David Mayerich
Optimization Tactics

• **profiling** – measuring the time complexity and memory use of a program and its component parts (instructions, loops, functions, etc.)

• **optimization** – modifying a program to make it work more efficiently or use fewer resources

• Optimization pipeline:
  1. Rigorously profile your code
  2. Find the primary bottleneck (usually a loop)
     • can the bottleneck be modified to significantly increase performance?
     • if not, move to the next bottleneck
  3. Goto 1
C/C++ Timers

• The simplest timer available in C++ is the time() function:

\[
\text{time_t time(time_t* timer);} \\
\]

#include <time.h>
int main(){
    unsigned int start = time(NULL);  // # of seconds since Jan 1, 1970
    ...
    funcCall( p0, p1, ...);          // call any CPU function
    ...
    unsigned int end = time(NULL);   // get the time after the call
    std::cout<<"funcCall(): "<<end - start<<"s"<<std::endl;
}
C++ High-Resolution Timers

• The `time()` function has a resolution of 1 second
  • helpful for finding bottleneck loop or function
  • not very useful for profiling loops

• C++ `std::chrono::high_resolution_clock` has a resolution up to nanoseconds:

```cpp
#include <chrono>

int main()
{
    std::chrono::high_resolution_clock::time_point start, stop;
    start = std::chrono::high_resolution_clock::now();
    ...
    funcCall( p0, p1, ...);
    ...
    stop = std::chrono::high_resolution_clock::now();
    std::chrono::milliseconds d;
    d = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    std::cout << “funcCall(): ” << d.count() << “ms” << std::endl;
}
```
C++ High-Resolution Timers

• Timer resolutions:

  `std::chrono::hours`
  `std::chrono::minutes`
  `std::chrono::seconds`
  `std::chrono::milliseconds`
  `std::chrono::microseconds`
  `std::chrono::nanoseconds`

• Most interesting operations occur at the millisecond time-scale
Timing CUDA Using Device Synchronization

• CPU continues operations while the GPU executes
  • CPU timing calls will return immediately (effectively taking no time)
• Use cudaDeviceSynchronize() to set a synchronization barrier

```c
#include <time.h>

int main(){
    unsigned int start = time(NULL);    // # of seconds since Jan 1, 1970
    ...
    kernelCall<<<blocks, threads>>>( p0, p1, ...);     // call a kernel
    cudaDeviceSynchronize();             // set a device sync barrier
    unsigned int end = time(NULL);      // get the time after the call
    ...
    std::cout << "funcCall(): " << end - start << "s" << std::endl;
}
```
CUDA Streams and Events

• Synchronization barriers can stall the GPU pipeline
  • GPUs **can** do some things simultaneously (asynchronous copies, etc.)
  • Asynchronous calls can’t run at the same time as a kernel

• CUDA organizes events and API calls into a *stream* for efficiency
  • CUDA automatically optimizes events (ex. double up asynchronous calls)
  • Use the CUDA event API to insert timing events into the stream
CUDA Event API

cudaError_t cudaEventCreate(cudaEvent_t* event, unsigned int flags = cudaEventDefault);
  • create a generic CUDA event (contains timing data)

cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream = 0);
  • place an event into the CUDA stream

cudaError_t cudaEventSynchronize(cudaEvent_t event);
  • blocks CPU execution until an event is recorded

cudaError_t cudaEventElapsedTime(float* ms, cudaEvent_t start, cudaEvent_t end);
  • returns the time between the recording of two events
CUDA Timers Using Events

```c
int main()
{
    cudaEvent_t start, stop; // declare a start and stop event
    cudaEventCreate(&start); // create both events
    cudaEventCreate(&stop);
    ... // copy data to the GPU and configure
    cudaEventRecord(start); // insert the start event into the stream
    kernelCall<<<blocks, threads>>>(...); // run the kernel to be profiled
    cudaEventRecord(stop); // insert the stop event into the stream
    ... // add any other GPU activity to stream
    // all GPU events are in the stream, so stalls don't matter
    cudaEventSynchronize(stop); // wait for the stop event, if it isn't done
    float milliseconds = 0; // declare a variable to store runtime
    cudaEventElapsedTime(&milliseconds, start, stop); // get the elapsed time
}
```
Accurately calculate throughput – kernel

```c
__global__ void kernelMatrixMult(float* C, float* A, float* B, size_t M, size_t N)
{
    size_t i = blockIdx.y * blockDim.y + threadIdx.y;  //calculate the i index
    size_t j = blockIdx.x * blockDim.x + threadIdx.x;  //calculate the j index
    if(i >= M || j >= M) return;  //return if (i,j) is outside the matrix
    float c = 0;  //initialize a register to store the result
    for(size_t n = 0; n < N; n++)  //for each element in the dot product
        c += A[n*M+i] * B[j*N+n];  //perform a multiply-add
    C[i*M + j] = c;  //send the register value to global memory
}
```
Accurately calculate throughput – host

```c
int main(){
    cudaEvent_t start, stop;  //declare a start and stop event
    cudaEventCreate(&start);  //create both events
    cudaEventCreate(&stop);
    ...  //copy data to the GPU and configure
    cudaEventRecord(start);   //insert the start event into the stream
    kernelMatrixMult<<<blocks, threads>>>(...);  //multiply matrices
    cudaEventRecord(stop);    //insert the stop event into the stream
    ...
    //all GPU events are in the stream, so stalls don’t matter
    cudaEventSynchronize(stop);  //wait for the stop event, if it isn’t done
    float milliseconds = 0;     //declare a variable to store runtime
    cudaEventElapsedTime(&milliseconds, start, stop); //get the elapsed time
    std::cout<<"FLOPS: "<<(2*N*M*M / (milliseconds / 1000))<<std::endl;
    std::cout<<"B/s:   "<<(2*4*N*M*M / (milliseconds / 1000))<<std::endl;
}
```
CUDA output

• CUDA can provide console output (since compute capability 2.0)
• Use printf(⋯) to print from a kernel

```c
__global__ void kernelHello()
{
    int block = blockIdx.x;        //get the current block index
    int thread = threadIdx.x;      //get the current thread index
    printf("Hello from block %d and thread %d\n", block, thread);
}

int main()
{
    kernelHello<<<32, 32>>>();    //execute the kernel
    cudaDeviceSynchronize();       //barrier for the current CUDA device
}
Demos

- CUDA output
- Visual Studio debugging