Universidade Estadual de Feira de Santana

Cuda Runtime API

Feira de Santana - BA

Junho, 2016

1 Introdução

Este tutorial é um resultado do projeto de pesquisa do estudante bolsista (Fapesb) do Lacad, Cássio Silva de Sá Santos e tem como objetivo apresentar um manual para a medição de tempo de execução dos kernels CUDA utilizando eventos bem como a maneira correta de fazer o controle do perfil dentro da aplicação, dizendo assim, qual parte do código deverá ter seus métodos analisados.

Este tutorial tem como principal fonte a documentação oficial da Nvidia [1] e como complemento os testes e estudos feitos pelo bolsista.

2 Cuda Runtime API

O CUDA Runtime API disponibiliza algumas funções para controle do perfil gerado das aplicações as quais tem o papel de Inicializar, ativar e desativar a geração do perfil. Neste tutorial serão tratadas as formas de utilização destas funções bem como a utilização da API de eventos para medir o tempo das aplicações CUDA.

2.1 Executando Profiler através da chamada a API

Biblioteca: cuda_profiler_api.h
O Cuda RunTime API disponibiliza um método para iniciar a geração de perfil dentro do código de uma aplicação CUDA. Esta API geralmente é utilizada para gerar o perfil de diferentes conjuntos de contadores repetindo na execução do kernel [2].

O método em questão é:

```
cudaError_t cudaProfilerInitialize (const char* configFile,
                                    const char* outputFile,
                                    cudaOutputMode_t outputMode)
```

Obs.: Configurações previamente definidas por variáveis de ambiente (Ver tutorial sobre Command-Line-Profiler) são sobrescritas pelo cudaProfilerInitialize()
Obs2.: O Cuda Profiler não pode ser iniciado pela API se outra ferramenta de Profiler estiver ativa.
Os parâmetros deste métodos indicam respectivamente:

1. nome do arquivo de configurações que lista os contadores e opções para o perfil
2. Nome do arquivo de saida do perfil no qual os resultados serão guardados
3. O modo de saída dos resultados (Chave-Valor ou separado por virgula)

2.2 Limitando a região do Profiler

Por padrão, as ferramentas de profiling coletam dados de toda a execução da aplicação. Porém é possível limitar a região de profiler para reduzir a quantidade de dados de perfil que a ferramenta deve processar e, portanto, focar a atenção na parte do código no qual o resultado de uma otimização vai ser mais significativo no ganho de performance [3].

Para limitar a região que deseja-se fazer o profiler na aplicação, o CUDA provem funções para iniciar e finalizar a coleta de dados.

Obs.: Quando estiver usando as funções listadas a seguir, será necessário induzir a ferramenta de profiling para desativar o profiling no inicio da aplicação.

As funções que podem ser utilizadas são:

1. cudaProfilerStart(): para iniciar o profiling

2. cudaProfilerStop(): para finalizar o profiling

Para utilizar estas funções é necessário incluir a biblioteca cuda_profiler_api.h ou quando se tratar de uma API de driver usar cudaProfiler.h

2.2.1 Passos:

1.1 Adicione a Inicialização do perfil dentro de seu código.

```
cudaProfilerInitialize();
```

1.2 Adicione cudaProfilerStart() / cudaProfilerStop()

```
for (i = 0; i < N; i++) {
    if (i == 12) cudaProfilerStart();
    if (i == 15) cudaProfilerStop();
}
```

2.3 Compilando

Para compilar arquivos .cu basta utilizar o compilador nvcc disponibilizado na instalação do Cuda ToolKit.

```
$ nvcc x.cu
```

2.4 Parâmetros utilizados em compilação

Alguns parâmetros de compilação podem ser utilizados juntamente com o nvcc para auxiliar no profiler ou debug de aplicações cuda em tempo de compilação [4], entre eles está:

Uso de recursos (–resource-usage)
Um resumo sobre a quantidade de registros usados e a quantidade de memória necessária por função do dispositivo compilada pode ser visualizado passando a opção –resource-usage para o compilador *nvcc*.
Exemplo:

```
$ nvcc --resource-usage seu_cod.cu
```

ou

```
$ nvcc -res-usage seu_cod.cu
```

ou

```
$ nvcc -arch=sm_20 -ptxas-options=-v seu_cod.cu
```

```
[root@enxu testes]# nvcc --resource-usage
a.out             asyncAPI.cu       .asyncAPI.cu.swp  syncAPI.cu        testeAsyn.cu
[root@enxu testes]# nvcc --resource-usage asyncAPI.cu -o async
asyncAPI.cu(49): warning: variable "devID" was declared but never referenced

asyncAPI.cu(49): warning: variable "devID" was declared but never referenced

ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z16increment_kernelPii' for 'sm_20'
ptxas info    : Function properties for _Z16increment_kernelPii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 4 registers, 44 bytes cmem[0]
[root@enxu testes]#
```

Figura 1: Execução do nvcc com a opção –resource-usage

Na ultima linha do exemplo da **Figura 1**, temos a informação de que cada Thread CUDA está usando 4 registradores. [5]

Além disso, podemos ver o tamanho total de memória constante sendo utilizada (Entre variáveis declaradas como _constant_ e argumentos de kernel), no exemplo da Figura X temos um total de 44 bytes de memória constante.

Bytes stack frame representa a soma total da memória para os quadros de pilha das funções _global_ e _device_

2.5 Usando Eventos CUDA para medir tempo

Temporização de funções em GPU com o CUDA Event API:

1. cudaEventCreate()
2. cudaEventRecord()
3. cudaEventSynchronize()
4. cudaEventDelete()
5. cudaEventElapsedTime()

A temporização no CUDA pode ser feita a partir do registro de eventos [6]. Ou seja, você declara um evento, utilizando a função CUDA: cudaEventCreate(). Então você registra o evento utilizando o cudaEventRecord() antes e depois do kernel, assim estes eventos registrarão o tempo antes e depois da execução do kernel então basicamente o que deve ser feito é efetuar a diferença entre os tempos dos eventos usando o cudaEventElapsedTime().

A função cudaEventSynchronize() é requerida pois cudaEventRecord() é uma função assíncrona.

Exemplo de uso.:

```
cudaEvent_t start, stop;
float elapsed;
```

```
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(&start, 0);
fool_kernel<<<grid,block>>>();
cudaEventRecord(&stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsed,start,stop);
printf("Elapsed time %f (seconds) \n", elapsed/1000);
```

Com os resultados obtidos podem ser tomadas algumas providencias:

1. Se o tempo da CPU é maior comparado com o tempo da GPU:
   (a) Determine se mais trabalhos da CPU podem ser transferidos para a GPU
2. Se uma determina função Kernel só representa uma pequena fração do tempo de execução:
   (a) Otimize os kernels mais dominantes primeiro, e em seguida re-avalie a performance.

Referências

[1] http://docs.nvidia.com/cuda/cuda-runtime-api/index.htm

[2] http://datamining.xmu.edu.cn/documentation/cuda4.1/group_CUDART_PROFILER_g30df1f0afc74fb91f098f817ce87726b.html

[3] http://docs.nvidia.com/cuda/cuda-runtime-api/group_CUDART_PROFILER.html#group_CUDART_PROFILER

[4] http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

[5] http://stackoverflow.com/questions/12388207/interpreting-output-of-ptxas-options-v

[6] http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#using-cuda-gpu-timers