HanLP Handbook

version 1.0

hankcs

April 28, 2018

Contents

Welcome to CUDA Programming documentation!

Contents:

1 namespace for ceres libs

1. namespace for ceres libs

```
自动微分函数
using ceres::AutoDiffCostFunction;
损失函数
using ceres::CostFunction;
using ceres::Problem;
Solver
using ceres::Solver;
Solve
using ceres::Solve;
```

2. A templated CostFunctor

residual is : 10 - x

```
// A templated cost functor that implements the residual r = 10 -
// x. The method operator() is templated so that we can then use an
// automatic differentiation wrapper around it to generate its
// derivatives.
struct CostFunctor {
  template <typename T> bool operator()(const T* const x, T* residual) const {
    residual[0] = T(10.0) - x[0];
    return true;
  }
};
```

3. main function

```
int main(int argc, char** argv) {
  google::InitGoogleLogging(argv[0]);

  // The variable to solve for with its initial value. It will be
  // mutated in place by the solver.
  double x = 0.5;
  const double initial_x = x;

  // Build the problem.
  Problem problem;

  // Set up the only cost function (also known as residual). This uses
  // auto-differentiation to obtain the derivative (jacobian).
  CostFunction* cost_function =
      new AutoDiffCostFunction<CostFunctor, 1, 1>(new CostFunctor);
  problem.AddResidualBlock(cost_function, NULL, & x);

  // Run the solver!
  Solver::Options options;
  options.minimizer_progress_to_stdout = true;
  Solver::Summary summary;
  Solve(options, &problem, &summary);

  std::cout << summary.BriefReport() << "\n";
  std::cout << "x : " << initial_x
            << " -> " << x << "\n";
  return 0;
}
```

4. Summary

Flow

- build problem
- 构建cost function
- run the solver

2 含有多个子函数的极小值

Flow:

- 1.初始值

```
double x1 =  3.0;
double x2 = -1.0;
```

```
double x3 =  0.0;
double x4 =  1.0;
```

- 2.构造损失函数

```
struct F1 {
  template <typename T> bool operator()(const T* const x1,
                                        const T* const x2,
                                        T* residual) const {
    // f1 = x1 + 10 * x2;
    residual[0] = x1[0] + T(10.0) * x2[0];
    return true;
  }
};

struct F2 {
  template <typename T> bool operator()(const T* const x3,
                                        const T* const x4,
                                        T* residual) const {
    // f2 = sqrt(5) (x3 - x4)
    residual[0] = T(sqrt(5.0)) * (x3[0] - x4[0]);
    return true;
  }
};

struct F3 {
  template <typename T> bool operator()(const T* const x2,
                                        const T* const x4,
                                        T* residual) const {
    // f3 = (x2 - 2 x3)^2
    residual[0] = (x2[0] - T(2.0) * x4[0]) * (x2[0] - T(2.0) * x4[0]);
    return true;
  }
};

struct F4 {
  template <typename T> bool operator()(const T* const x1,
                                        const T* const x4,
                                        T* residual) const {
    // f4 = sqrt(10) (x1 - x4)^2
    residual[0] = T(sqrt(10.0)) * (x1[0] - x4[0]) * (x1[0] - x4[0]);
    return true;
  }
};
```

- 3.创建问题

```
Problem problem;
// Add residual terms to the problem using the using the autodiff
// wrapper to get the derivatives automatically. The parameters, x1 through
// x4, are modified in place.
//ceres::AutoDiffCostFunction<$`typename CostFunctor`, $`int kNumResiduals`, $`int N0`>
problem.AddResidualBlock(new AutoDiffCostFunction<F1, 1, 1, 1>(new F1),
                         NULL,
                         &x1, &x2);

problem.AddResidualBlock(new AutoDiffCostFunction<F2, 1, 1, 1>(new F2),
                         NULL,
                         &x3, &x4);
problem.AddResidualBlock(new AutoDiffCostFunction<F3, 1, 1, 1>(new F3),
                         NULL,
                         &x2, &x3);
problem.AddResidualBlock(new AutoDiffCostFunction<F4, 1, 1, 1>(new F4),
                         NULL,
                         &x1, &x4);
```

- 4.问题求解

```
Solver::Options options;
LOG_IF(FATAL, !ceres::StringToMinimizerType(FLAGS_minimizer,
                                            &options.minimizer_type))
    << "Invalid minimizer: " << FLAGS_minimizer
    << ", valid options are: trust_region and line_search.";

options.max_num_iterations = 100;
options.linear_solver_type = ceres::DENSE_QR;
options.minimizer_progress_to_stdout = true;

Solver::Summary summary;
```

```
solve 的输入为:  solver 的option, 以及要求解的问题problem

Solve(options, &problem, &summary);

std::cout << summary.FullReport() << "\n";
```

3 curve fitting

目的 curve fitting y = e ** (mx+c)

```
// Data generated using the following octave code.
//   randn('seed', 23497);
//   m = 0.3;
//   c = 0.1;
//   x=[0:0.075:5];
//   y = exp(m * x + c);
//   noise = randn(size(x)) * 0.2;
//   y_observed = y + noise;
//   data = [x', y_observed'];
```

flow

- 1.初始值

```
const int kNumObservations = 67;
const double data[] = {
  0.000000e+00, 1.133898e+00,
  7.500000e-02, 1.334902e+00,
  1.500000e-01, 1.213546e+00,
  2.250000e-01, 1.252016e+00,
  3.000000e-01, 1.392265e+00,
  3.750000e-01, 1.314458e+00,
  4.500000e-01, 1.472541e+00,
  5.250000e-01, 1.536218e+00,
  6.000000e-01, 1.355679e+00,
  6.750000e-01, 1.463566e+00,
  7.500000e-01, 1.490201e+00,
  8.250000e-01, 1.658699e+00,
  9.000000e-01, 1.067574e+00,
  9.750000e-01, 1.464629e+00,
  1.050000e+00, 1.402653e+00,
  1.125000e+00, 1.713141e+00,
  1.200000e+00, 1.527021e+00,
  1.275000e+00, 1.702632e+00,
  1.350000e+00, 1.423899e+00,
  1.425000e+00, 1.543078e+00,
  1.500000e+00, 1.664015e+00,
  1.575000e+00, 1.732484e+00,
  1.650000e+00, 1.543296e+00,
  1.725000e+00, 1.959523e+00,
  1.800000e+00, 1.685132e+00,
  1.875000e+00, 1.951791e+00,
  1.950000e+00, 2.095346e+00,
  2.025000e+00, 2.361460e+00,
  2.100000e+00, 2.169119e+00,
  2.175000e+00, 2.061745e+00,
  2.250000e+00, 2.178641e+00,
  2.325000e+00, 2.104346e+00,
  2.400000e+00, 2.584470e+00,
  2.475000e+00, 1.914158e+00,
  2.550000e+00, 2.368375e+00,
  2.625000e+00, 2.686125e+00,
  2.700000e+00, 2.712395e+00,
  2.775000e+00, 2.499511e+00,
  2.850000e+00, 2.558897e+00,
  2.925000e+00, 2.309154e+00,
  3.000000e+00, 2.869503e+00,
  3.075000e+00, 3.116645e+00,
  3.150000e+00, 3.094907e+00,
  3.225000e+00, 2.471759e+00,
  3.300000e+00, 3.017131e+00,
  3.375000e+00, 3.232381e+00,
  3.450000e+00, 2.944596e+00,
  3.525000e+00, 3.385343e+00,
  3.600000e+00, 3.199826e+00,
  3.675000e+00, 3.423039e+00,
  3.750000e+00, 3.621552e+00,
  3.825000e+00, 3.559255e+00,
  3.900000e+00, 3.530713e+00,
  3.975000e+00, 3.561766e+00,
  4.050000e+00, 3.544574e+00,
  4.125000e+00, 3.867945e+00,
```

```
  4.200000e+00, 4.049776e+00,
  4.275000e+00, 3.885601e+00,
  4.350000e+00, 4.110505e+00,
  4.425000e+00, 4.345320e+00,
  4.500000e+00, 4.161241e+00,
  4.575000e+00, 4.363407e+00,
  4.650000e+00, 4.161576e+00,
  4.725000e+00, 4.619728e+00,
  4.800000e+00, 4.737410e+00,
  4.875000e+00, 4.727863e+00,
  4.950000e+00, 4.669206e+00,
};
```

- 2.构造损失函数

```
struct ExponentialResidual {
  ExponentialResidual(double x, double y)
      : x_(x), y_(y) {}

  template <typename T> bool operator()(const T* const m,
                                        const T* const c,
                                        T* residual) const {
    residual[0] = T(y_) - exp(m[0] * T(x_) + c[0]);
    return true;
  }

 private:
  const double x_;
  const double y_;
};
```

- 3.创建问题

```
Problem problem;
for (int i = 0; i < kNumObservations; ++i) {
  problem.AddResidualBlock(
      new AutoDiffCostFunction<ExponentialResidual, 1, 1, 1>(
          new ExponentialResidual(data[2 * i], data[2 * i + 1])),
      NULL,
      &m, &c);
}
```

- 4. 问题求解

```
Solver::Options options;
options.max_num_iterations = 25;
options.linear_solver_type = ceres::DENSE_QR;
options.minimizer_progress_to_stdout = true;

Solver::Summary summary;
Solve(options, &problem, &summary);
std::cout << summary.BriefReport() << "\n";
std::cout << "Initial m: " << 0.0 << " c: " << 0.0 << "\n";
std::cout << "Final   m: " << m << " c: " << c << "\n";
```

flow

- 1.初始值
- 2.构造损失函数
- 3.创建问题
- 4.问题求解

4 bundle adjustment

flow

- 1.初始值
- 2.构造损失函数
- 3.创建问题
- 4.问题求解

flow

- 1.初始值
- 2.构造损失函数
- 3.创建问题
- 4.问题求解

5_CheckYourCudaResults

通过比较cpu结果与gpu结果来检查我们的gpu运算结果是否正确.

```
#include "../common/common.h"
#include <cuda_runtime.h>
#include <stdio.h>

/*
 * This example demonstrates a simple vector sum on the GPU and on the host.
 * sumArraysOnGPU splits the work of the vector sum across CUDA threads on the
 * GPU. Only a single thread block is used in this small case, for simplicity.
 * sumArraysOnHost sequentially iterates through vector elements on the host.
 */

void checkResult(float *hostRef, float *gpuRef, const int N)
{
    double epsilon = 1.0E-8;
    bool match = 1;

    for (int i = 0; i < N; i++)
    {
        if (abs(hostRef[i] - gpuRef[i]) > epsilon)
        {
            match = 0;
            printf("Arrays do not match!\n");
            printf("host %5.2f gpu %5.2f at current %d\n", hostRef[i],
                   gpuRef[i], i);
            break;
        }
    }

    if (match) printf("Arrays match.\n\n");

    return;
}


void initialData(float *ip, int size)
{
    // generate different seed for random number
    time_t t;
    srand((unsigned) time(&t));

    for (int i = 0; i < size; i++)
    {
        ip[i] = (float)(rand() & 0xFF) / 10.0f;
    }

    return;
}


void sumArraysOnHost(float *A, float *B, float *C, const int N)
{
    for (int idx = 0; idx < N; idx++)
        C[idx] = A[idx] + B[idx];
}

__global__ void sumArraysOnGPU(float *A, float *B, float *C, const int N)
{
    int i = threadIdx.x;

    if (i < N) C[i] = A[i] + B[i];
}


int main(int argc, char **argv)
{
    printf("%s Starting...\n", argv[0]);

    // set up device
    int dev = 0;
    CHECK(cudaSetDevice(dev));

    // set up data size of vectors
    int nElem = 1 << 5;
    printf("Vector size %d\n", nElem);

    // malloc host memory
    size_t nBytes = nElem * sizeof(float);

    float *h_A, *h_B, *hostRef, *gpuRef;
```

```
    h_A     = (float *)malloc(nBytes);
    h_B     = (float *)malloc(nBytes);
    hostRef = (float *)malloc(nBytes);
    gpuRef  = (float *)malloc(nBytes);

    // initialize data at host side
    initialData(h_A, nElem);
    initialData(h_B, nElem);

    memset(hostRef, 0, nBytes);
    memset(gpuRef,  0, nBytes);

    // malloc device global memory
    float *d_A, *d_B, *d_C;
    CHECK(cudaMalloc((float**)&d_A, nBytes));
    CHECK(cudaMalloc((float**)&d_B, nBytes));
    CHECK(cudaMalloc((float**)&d_C, nBytes));

    // transfer data from host to device
    CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_C, gpuRef, nBytes, cudaMemcpyHostToDevice));

    // invoke kernel at host side
    dim3 block (nElem);
    dim3 grid  (1);

    sumArraysOnGPU<<<grid, block>>>(d_A, d_B, d_C, nElem);
    printf("Execution configure <<<%d, %d>>>\n", grid.x, block.x);

    // copy kernel result back to host side
    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    // add vector at host side for result checks
    sumArraysOnHost(h_A, h_B, hostRef, nElem);

    // check device results
    checkResult(hostRef, gpuRef, nElem);

    // free device global memory
    CHECK(cudaFree(d_A));
    CHECK(cudaFree(d_B));
    CHECK(cudaFree(d_C));

    // free host memory
    free(h_A);
    free(h_B);
    free(hostRef);
    free(gpuRef);

    CHECK(cudaDeviceReset());
    return(0);
}
```

6_TimingYourKernel测试你的kernel性能

- 1. 用sys/time.h的函数

CPU timer can be created by using the gettimeofday system call to get the system’s wall-clock time, which returns the number of seconds since the epoch. You need to include the sys/time.h header file

```
include <sys/time.h>
inline double seconds()
{
    struct timeval tp;
    struct timezone tzp;
    int i = gettimeofday(&tp, &tzp);
    return ((double)tp.tv_sec + (double)tp.tv_usec * 1.e-6);
}
```

- 2. 用nvprof 命令行来测试cuda API所好用的时间

```
nvprof ./可执行文件
```

```
nvprof o./oumArraysOnGPU-timer

NVPROF is profiling process 12644, command: ./sumArraysOnGPU-timer
Using Device 0: GeForce GTX 1080
Vector size 16777216
```

6_TimingYourKernel测试你的kernel性能

```
initialData Time elapsed 0.604351 sec
sumArraysOnHost Time elapsed 0.012240 sec
sumArraysOnGPU <<<  32768, 512  >>>  Time elapsed 0.000959 sec
Arrays match.

==12644== Profiling application: ./sumArraysOnGPU-timer
==12644== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
73.05%  23.592ms         3  7.8640ms  7.7391ms  7.9720ms  [CUDA memcpy HtoD]
24.38%  7.8736ms         1  7.8736ms  7.8736ms  7.8736ms  [CUDA memcpy DtoH]
2.56%  828.31us         1  828.31us  828.31us  828.31us  sumArraysOnGPU(float*, float*, float*, int)

==12644== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
89.62%  323.27ms         3  107.76ms  252.03us  322.75ms  cudaMalloc
8.83%  31.856ms         4  7.9639ms  7.8661ms  8.0520ms  cudaMemcpy
1.07%  3.8497ms         3  1.2832ms  182.49us  1.8400ms  cudaFree
0.26%  923.12us         1  923.12us  923.12us  923.12us  cudaDeviceSynchronize
0.09%  312.99us        91  3.4390us     105ns  138.29us  cuDeviceGetAttribute
0.08%  305.22us         1  305.22us  305.22us  305.22us  cudaGetDeviceProperties
0.03%  110.51us         1  110.51us  110.51us  110.51us  cuDeviceTotalMem
0.01%  34.371us         1  34.371us  34.371us  34.371us  cuDeviceGetName
0.01%  28.581us         1  28.581us  28.581us  28.581us  cudaLaunch
0.00%  5.7500us         1  5.7500us  5.7500us  5.7500us  cudaSetDevice
0.00%  1.7300us         3     576ns     133ns  1.3610us  cuDeviceGetCount
0.00%  1.5580us         1  1.5580us  1.5580us  1.5580us  cudaConfigureCall
0.00%  1.2100us         4     302ns     149ns     521ns  cudaSetupArgument
0.00%  1.1950us         3     398ns     109ns     885ns  cuDeviceGet
0.00%     360ns         1     360ns     360ns     360ns  cudaGetLastError
```

.

3. 例子

```
#include "../common/common.h"
#include <cuda_runtime.h>
#include <stdio.h>

/*
 * This example demonstrates a simple vector sum on the GPU and on the host.
 * sumArraysOnGPU splits the work of the vector sum across CUDA threads on the
 * GPU. Only a single thread block is used in this small case, for simplicity.
 * sumArraysOnHost sequentially iterates through vector elements on the host.
 * This version of sumArrays adds host timers to measure GPU and CPU
 * performance.
 */

 void checkResult(float *hostRef, float *gpuRef, const int N)
 {
   double epsilon = 1.0E-8;
   bool match = 1;

   for (int i = 0; i < N; i++)
   {
     if (abs(hostRef[i] - gpuRef[i]) > epsilon)
     {
       match = 0;
       printf("Arrays do not match!\n");
       printf("host %5.2f gpu %5.2f at current %d\n", hostRef[i],
           gpuRef[i], i);
       break;
     }
   }

   if (match) printf("Arrays match.\n\n");

   return;
}

void initialData(float *ip, int size)
{
   // generate different seed for random number
   time_t t;
   srand((unsigned) time(&t));

   for (int i = 0; i < size; i++)
   {
     ip[i] = (float)( rand() & 0xFF ) / 10.0f;
   }

   return;
}
```

7

6_TimingYourKernel测试你的kernel性能

```
void sumArraysOnHost(float *A, float *B, float *C, const int N)
{
  for (int idx = 0; idx < N; idx++)
  {
    C[idx] = A[idx] + B[idx];
  }
}
__global__ void sumArraysOnGPU(float *A, float *B, float *C, const int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < N) C[i] = A[i] + B[i];
}

int main(int argc, char **argv)
{
  printf("%s Starting...\n", argv[0]);

  // set up device
  int dev = 0;
  cudaDeviceProp deviceProp;
  CHECK(cudaGetDeviceProperties(&deviceProp, dev));
  printf("Using Device %d: %s\n", dev, deviceProp.name);
  CHECK(cudaSetDevice(dev));

  // set up data size of vectors
  int nElem = 1 << 24;
  printf("Vector size %d\n", nElem);

  // malloc host memory
  size_t nBytes = nElem * sizeof(float);

  float *h_A, *h_B, *hostRef, *gpuRef;
  h_A     = (float *)malloc(nBytes);
  h_B     = (float *)malloc(nBytes);
  hostRef = (float *)malloc(nBytes);
  gpuRef  = (float *)malloc(nBytes);

  double iStart, iElaps;

  // initialize data at host side
  iStart = seconds();
  initialData(h_A, nElem);
  initialData(h_B, nElem);
  iElaps = seconds() - iStart;
  printf("initialData Time elapsed %f sec\n", iElaps);
  memset(hostRef, 0, nBytes);
  memset(gpuRef,  0, nBytes);

  // add vector at host side for result checks
  iStart = seconds();
  sumArraysOnHost(h_A, h_B, hostRef, nElem);
  iElaps = seconds() - iStart;
  printf("sumArraysOnHost Time elapsed %f sec\n", iElaps);

  // malloc device global memory
  float *d_A, *d_B, *d_C;
  CHECK(cudaMalloc((float**)&d_A, nBytes));
  CHECK(cudaMalloc((float**)&d_B, nBytes));
  CHECK(cudaMalloc((float**)&d_C, nBytes));

  // transfer data from host to device
  CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice));
  CHECK(cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice));
  CHECK(cudaMemcpy(d_C, gpuRef, nBytes, cudaMemcpyHostToDevice));

  // invoke kernel at host side
  int iLen = 512;
  dim3 block (iLen);
  dim3 grid  ((nElem + block.x - 1) / block.x);

  iStart = seconds();
  sumArraysOnGPU<<<grid, block>>>(d_A, d_B, d_C, nElem);
  CHECK(cudaDeviceSynchronize());
  iElaps = seconds() - iStart;
  printf("sumArraysOnGPU <<<  %d, %d  >>>  Time elapsed %f sec\n", grid.x,
      block.x, iElaps);

  // check kernel error
  CHECK(cudaGetLastError()) ;

  // copy kernel result back to host side
  CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));
```

```
  // check device results
  checkResult(hostRef, gpuRef, nElem);

  // free device global memory
  CHECK(cudaFree(d_A));
  CHECK(cudaFree(d_B));
  CHECK(cudaFree(d_C));

  // free host memory
  free(h_A);
  free(h_B);
  free(hostRef);
  free(gpuRef);

  return(0);
}
```

7_理解基本的CUDA执行模型-CUDA Execution Model

通过以前的知识我们知道如何去通过配置grid和block来lauch kernel，本章我们会有一些指导如何来配置grid和block来提高性能.

You might want to know if there are some guidelines for selecting grid and block configurations.

This chapter will answer those questions and provide you with deeper insight into kernel launch configurations and performance profile information,

but from a different angle: the hardware perspective.

GPU architecuture Overview

GPU 是由一组流处理器的阵列来构成(Streaming Multiprocessors).

CUDA 组件

- CUDA cores
- Shared Memory/L1 Cache
- Register File
- Load/Store Units
- Special Function Units
- Warp Scheduler
- SIMT VS SIMD

CUDA 是单指令多线程架构(SIMT - single Instruction Multiple Thread)，如此管理和执行以32个线程组成一组的warp.

在同一个warp中的所有线程在同一时间执行相同的指令,每个线程有自己的指令地址计数器和寄存器状态,有各自的数据.每个SM把block分成以32个线程的warp为一组来调度并执行.

SIMT有别与AMD的SIMD，SIMT的好处就是无需开发者费力把数据凑成合适的矢量长度，并且SIMT允许每个线程有不同的分支。 纯粹使用SIMD不能并行的执行有条件跳转的函数，很显然条件跳转会根据输入数据不同在不同的线程中有不同表现，这个只有利用SIMT才能做到。

一个重要不同是SIMD中的向量中的元素相互之间可以自由通信，因为它们存在于相同的地址空间（例如，都在CPU的同一寄存器中），而SIMT中的每个线程的寄存器都是私有的，线程之间只能通过shared memory和同步机制进行通信。

总结：SIMT与SIMD本质相同：都是单指令多数据。 SIMT比SIMD更灵活，允许一条指令的多数据分开寻址；SIMD是必须连续在一起的片段。 SIMT形式上是多线程，本质上还是一个线程，只不过数据可以零散的分散开。但是如果你真的将数据分散开的话，执行效率上又会大打折扣，因为不满足并行访问的要求。 总之SIMT是SIMD的一种推广，更灵活而已。

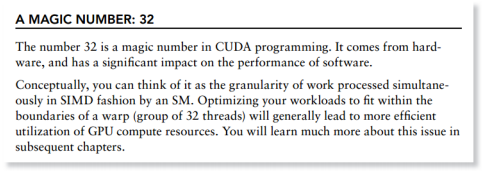
- SIMT model include three key features that SIMD does not.

-1. Each thread has its own instruction address counter.

-2. Each thread has its own register state.

-3. Each thread can have an independent execution path.

CUDA GPU magic number



A MAGIC NUMBER: 32

The number 32 is a magic number in CUDA programming. It comes from hardware, and has a significant impact on the performance of software.

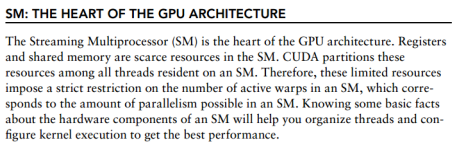
Conceptually, you can think of it as the granularity of work processed simultaneously in SIMD fashion by an SM. Optimizing your workloads to fit within the boundaries of a warp (group of 32 threads) will generally lead to more efficient utilization of GPU compute resources. You will learn much more about this issue in subsequent chapters.

threads的并行与同步

需要注意的是，大部分threads只是逻辑上并行，并不是所有的thread可以在物理上同时执行。例如，遇到分支语句（if else，while，for等）时，各个thread的执行条件不一样必然产生分支执行，这就导致同一个block中的线程可能会有不同步调。另外，并行thread之间的共享数据会导致竞态：多个线程请求同一个数据会导致未定义行为。CUDA提供了cudaThreadSynchronize()来同步同一个block的thread以保证在进行下一步处理之前，所有thread都到达某个时间点。 同一个warp中的thread可以以任意顺序执行，active warps被sm资源限制。当一个warp空闲时，SM就可以调度驻留在该SM中另一个可用warp。在并发的warp之间切换是没什么消耗的，因为硬件资源早就被分配到所有thread和block，所以该新调度的warp的状态已经存储在SM中了。不同于CPU，CPU切换线程需要保存/读取线程上下文（register内容），这是非常耗时的，而GPU为每个threads提供物理register，无需保存/读取上下文。

8_Understanding_the_nature_of_WARP

THE Heart of THE GPU Architecture



SM: THE HEART OF THE GPU ARCHITECTURE

The Streaming Multiprocessor (SM) is the heart of the GPU architecture. Registers and shared memory are scarce resources in the SM. CUDA partitions these resources among all threads resident on an SM. Therefore, these limited resources impose a strict restriction on the number of active warps in an SM, which corresponds to the amount of parallelism possible in an SM. Knowing some basic facts about the hardware components of an SM will help you organize threads and configure kernel execution to get the best performance.

The Fermi Architecture

The Kepler Architecture

Kepler架构有一个特性,就是动态并行(Dynamic Parallelism) 它允许通过GPU端lauch kernel,而不是必须由host端launch kernel,这样可以省去与cpu的交互.

Profile-Driven Optimizaiton性能驱动的优化

Profiling 可以分析程序性能通过测量:

- 代码的空间复杂度与时间复杂度.
- 特殊指令的使用
- 函数调用的频率与使用周期

Profiling tools provide deep insight into kernel performance and help you identify bottlenecks in kernels.

CUDA 提供了两种性能分析工具

CUDA provides two primary profiling tools:

- nvvp, a standalone visual profiler;
- nvprof, a command-line profiler.
-
    1. nvvp

nvvp is a Visual Profiler, which helps you to visualize and optimize the performance of your CUDA program. This tool displays a timeline of program activity on both the CPU and GPU, helping you to identify opportunities for performance improvement. In addition, nvvp analyzes your application for potential performance bottlenecks and suggests actions to take to eliminate or reduce those bottlenecks. The tool is available as both a standalone application and as part of the Nsight Eclipse Edition (nsight).

-
    2. nvprof

nvprof collects and displays profiling data on the command line. nvprof was introduced with CUDA 5 and evolved from an older command-line CUDA profiling tool. Like nvvp, it enables the collection of a timeline of CUDA-related activities on both the CPU and GPU, including kernel execution, memory transfers, and CUDA API calls. It also enables you to collect hardware counters and performance metrics for CUDA kernels.

你将学到如何写高效的kernel通过使用profile-driven approach

Throughout the examples and exercises in this book, you will learn about the proper metrics for analyzing kernels with the command-line profiler and master the skill of writing an efficient kernel using the profile-driven approach.

In this book, you will mainly use nvprof to dissect your kernel with the goal of improving performance. You will learn how to select appropriate counters and metrics, and use nvprof from the command line to collect profiling data, which can then be used to plan an optimization strategy. You will learn how to analyze your kernel from multiple angles using different counters and metrics.

影响性能的可能的原因

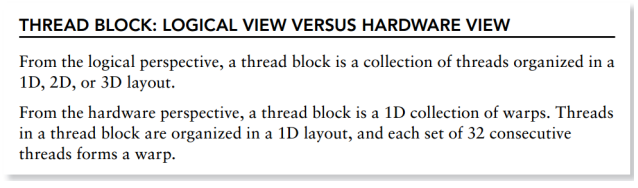
1. 存储带宽 Memory Bandwidth
2. 计算资源
3. 指令与存储延迟 instruction and memory latency

8_Understanding_the_nature_of_WARP

Launching kernel时,从软件角度看,所有的线程是并行的.

但从硬件角度来讲,并不是所有的线程同时执行.

cuda中,将32个线程分到一个单元,该单元称为warp.



THREAD BLOCK: LOGICAL VIEW VERSUS HARDWARE VIEW

From the logical perspective, a thread block is a collection of threads organized in a 1D, 2D, or 3D layout.

From the hardware perspective, a thread block is a 1D collection of warps. Threads in a thread block are organized in a 1D layout, and each set of 32 consecutive threads forms a warp.

warp数计算

分配的warp数 = 1个Block的threads总数 / warpSize , 然后向上取整,

即 分配的warp数 = ceil(1个Block的threads总数 / warpSize)

如果现在有80个线程,则会分配3个warp,即96个线程来支持80个线程,尽管这些线程没有用,但是仍然会消耗SM资源,比如说寄存器.

warp Divergence

一个warp中的32个threads需要执行相同的指令,但是如果32个线程中遇到控制流语句时,如果进入不同的分支,那么同一时刻除了正在执行的分支外,其余分支被阻塞了,会影响性能.这类问题就是warp divergence. 比如在一个warp中程序如下:

```
if(cond) {} else {}
```

如果有16个thread的cond为True,则这16个线程执行,同时其余16个线程被暂停.等这16个线程执行完之后,再执行其余16个线程.因此就会导致性能减半.

为了获得最好的性能,需要避免同一个warp存在不同的执行路径.

如何避免warp divergence

将条件改为以warp大小为步调,然后取奇偶数,

if((cond/warpSize) % 2) {} else {}

用nvprof来分析branch的效率,可以用来检查是否有可以提高的必要性.

```
nvprof --metrics branch_efficiency ./simpleDivergence
```

nvcc -g -G 编译时不做branch 的优化 nvcc -O3 编译时会做branch的优化

资源分配

计算资源限制了有效的warp数,因此,必须要注意硬件的限制,为最大化gpu的利用率,你需要最大化有效warps的数量.

compute resource partitioning requires special attention in CUDA programming: The compute resources limit the number of active warps. Therefore, you must be aware of the restrictions imposed by the hardware, and the resources used by your kernel. In order to maximize GPU utilization, you need to maximize the number of active warps.

一个thread block被成为active block当计算资源如寄存器,共享内存分配给该block.它所包含的warp就被成为active warps.active warps进一步会被分为三类:

- selected warp
- stalled warp
- eligible warp

分配grid,block size 指南

GUIDELINES FOR GRID AND BLOCK SIZE

Using these guidelines will help your application scale on current and future devices:

- Keep the number of threads per block a multiple of warp size (32).
- Avoid small block sizes: Start with at least 128 or 256 threads per block.
- Adjust block size up or down according to kernel resource requirements.
- Keep the number of blocks much greater than the number of SMs to expose sufficient parallelism to your device.
- Conduct experiments to discover the best execution configuration and resource usage.