INSTITUTE OF MICROELECTRONICS

360.252 Computational Science on Many-Core Architectures

Exercise 3

Author:
Peter Holzner, 01426733

Submission: November 10, 2020

Contents

1 Strided and Offset Memory Access

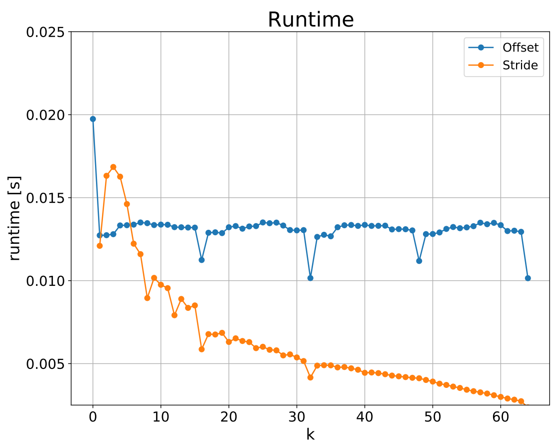
The code for this part of the exercise can be found here: 1.

In the figures below, we can see that the Offset kernel clearly outperforms the Stride kernel for any $k > 1$. This is no surprise, since the Stride kernel with $k > 1$ destroys any kind of data locality we have. Data access and transfers are usually bundled together into chunks of multiple bytes, depending on the cache size of the processing unit.

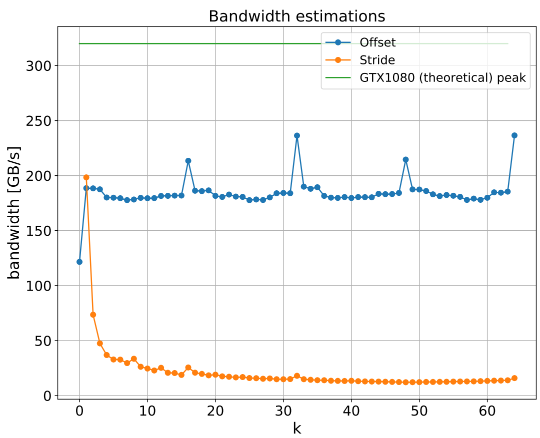
The offset kernel with $k = 0$ and the stride kernel with $k = 1$ should actually be equally fast since they would be identical. I don't quite understand why they aren't. There must be a mistake in my code or some other unaccounted problem dragging down performance.

The Offset kernel produces more interesting results. We have bandwidth peaks at multiples of 16, at $[16, 32, 48, 64]$ (there should also be a peak, not a trough at 0). Based on these results, I assume that the GTX1080 used for these tests uses a memory pipeline (cache) of size $8\ bytes * 16/3 = 128\ bytes$. Packing memory access such that they can fit precisely in multiples of the cache line size.

The peak bandwidth one can expect for these kernels is roughly $3/4 * theoretical\ peak = 3/4 * 320GB/s$. The $3/4$ stems from the fact that the GTX1080 uses a $256\ bit$[1] memory interface which fits precisely 4 double precision ($8\ byte = 64\ bit$) floats. We only use 3 out of 4 doubles per line of code in our kernel.



(a) Runtimes



(b) Bandwidth

Similar results were also shown by Nvidia themselves in an article on their website: `https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/`.

[1]https://de.wikipedia.org/wiki/Nvidia-GeForce-10-Serie

2 CG

Unfortunately, I was not able to complete this assignment. I verified that my kernels for the vector dot products and the vector additions worked, see , but when I implemented the CG method, my residuals kept increasing instead of decreasing. I simply ran out of time to fix my implementation. Nevertheless, you can find my code included in the kernel section, see .

3 CUDA Kernels

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include "timer.hpp"

#define BLOCK_SIZE 256
#define GRID_SIZE 256
#define TESTS 5
#define K_LIM 64
#define N_DEF 100'000'000
#define hRES_INIT -1.
#define dRES_INIT 0.


__global__ void initKernel(double* arr, const size_t N, const double val)
{
  const size_t stride = blockDim.x * gridDim.x;
  size_t tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
    arr[tid] = val;
}

/* Task 1 a) */
__global__ void vectorAddOffset(const double* x, const double* y, double* z, const size_t N,
     const size_t k) {
  const size_t stride = blockDim.x * gridDim.x;

  if (k){
    for (size_t tid = threadIdx.x + blockIdx.x * blockDim.x; tid < N-k; tid += stride)
    z[tid+k] = x[tid+k] + y[tid+k];
  }
  else
  {
    for (size_t tid = threadIdx.x + blockIdx.x * blockDim.x; tid < N; tid += stride)
    z[tid] = x[tid] + y[tid];
  }
}

/* Task 1 b) */
__global__ void vectorAddStride(const double* x, const double* y, double* z, const size_t N,
     const size_t k) {
  const size_t stride = blockDim.x * gridDim.x;

  if (k==1){
    for (size_t tid = threadIdx.x + blockIdx.x * blockDim.x; tid < N/k; tid += stride)
    z[tid] = x[tid] + y[tid];
  }
  else{
    for (size_t tid = threadIdx.x + blockIdx.x * blockDim.x; tid < N/k; tid += stride)
      z[tid*k] = x[tid*k] + y[tid*k];

  }

}

double median(std::vector<double>& vec)
{
  // modified taken from here: https://stackoverflow.com/questions/2114797/compute-median-of
      -values-stored-in-vector-c

  size_t size = vec.size();

  if (size == 0)
          return 0.;

  sort(vec.begin(), vec.end());
```

```
  size_t mid = size/2;

  return size % 2 == 0 ? (vec[mid] + vec[mid-1]) / 2 : vec[mid];
}

void printArray(double* x, const size_t N, const size_t num){
  std::string sep = " > ";
  for (int i=0; i < num; i++) std::cout << x[i] << sep;
  std::cout << std::endl;
  for (int i=0; i < num; i++) std::cout << x[N-1-i] << sep;
  std::cout << std::endl;
}

int main(void)
{
  size_t N = N_DEF;
  std::string mode = "csv";
  std::vector<double> time(TESTS, -1.);
  //std::vector<double> h_res(N, hRES_INIT); // init to -1
  double* h_res = new double[N];
  double* x = new double[N];
  double* y = new double[N];
  for (int i=0; i < N; ++i) h_res[i] = hRES_INIT;

  double *d_x, *d_y, *d_res;
  cudaMalloc(&d_x, N*sizeof(double));
  cudaMalloc(&d_y, N*sizeof(double));
  cudaMalloc(&d_res, N*sizeof(double));
  cudaDeviceSynchronize();
  initKernel<<<GRID_SIZE, BLOCK_SIZE>>>(d_x, N, 1.);
  initKernel<<<GRID_SIZE, BLOCK_SIZE>>>(d_y, N, 2.);
  cudaDeviceSynchronize();

  std::string sep = ";";
  std::cout << "k" << sep
            << "time" << sep
            << "N" << sep
            << "Grid Size" << sep
            << "Block Size" << std::endl;

  initKernel<<<GRID_SIZE, BLOCK_SIZE>>>(d_res, N, dRES_INIT); // init to 0
  cudaDeviceSynchronize();

  for (size_t k = 0; k < K_LIM; ++k)
  {
    Timer timer;

    for (int i = 0; i < TESTS; ++i)
    {
      timer.reset();

      vectorAddStride<<<GRID_SIZE, BLOCK_SIZE>>>(d_x, d_y, d_res, N, k);
      //vectorAddOffset<<<GRID_SIZE, BLOCK_SIZE>>>(d_x, d_y, d_res, N, k);
      cudaDeviceSynchronize();

      double rt = timer.get();

      time[i] = rt;
    }

    cudaMemcpy(h_res, d_res, N*sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(x, d_x, N*sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(y, d_y, N*sizeof(double), cudaMemcpyDeviceToHost);

    //std::string sep = " | ";

    std::cout << k << sep
      << std::scientific << median(time) << sep
      << N << sep
      << GRID_SIZE << sep
      << BLOCK_SIZE
```

```
        << std::endl;
  }
  std::cout << "-------------------------------------------------" << std::endl;
  std::cout << "res" << std::endl;
  printArray(h_res, N, 4);
  std::cout << "x" << std::endl;
  printArray(x, N, 4);
  std::cout << "y" << std::endl;
  printArray(y, N, 4);
  cudaFree(d_x);
  cudaFree(d_y);
  cudaFree(d_res);
  cudaDeviceSynchronize();
  delete[] h_res;

  return EXIT_SUCCESS;
}
```

Listing 1: Ex3.1 Strided and Offset Memory Access

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
#include <string>
#include "poisson2d.hpp"
#include "timer.hpp"

#define BLOCK_SIZE 256
#define GRID_SIZE 256

/** Computes y = A*x for a sparse matrix A in CSR format and vector x,y  */
__global__ void csr_Ax(const size_t N,
                       int *csr_rowoffsets, int *csr_colindices, double *csr_values,
                       double *x, double *y)
{
  const size_t stride = gridDim.x * blockDim.x;
  for (int i = threadIdx.x + blockDim.x * blockIdx.x;
    i < N;
    i += stride)
  {
    double tmp = 0.0;
    for (int j = csr_rowoffsets[i]; j < csr_rowoffsets[i+1]; ++j)
      tmp += csr_values[j] * x[csr_colindices[j]];
    y[i] = tmp;
  }
}

__global__ void xADDay(const size_t N, double *x, double *y, double *z, const double alpha)
{
  const size_t stride = blockDim.x * gridDim.x;
  for(size_t i = threadIdx.x + blockIdx.x * blockDim.x; i < N; i += stride)
      z[i] = x[i] + alpha * y[i];
}

__global__ void xDOTy(const size_t N, double* x, double* y, double* z)
{
  size_t tid = threadIdx.x + blockDim.x* blockIdx.x;
  size_t stride = blockDim.x* gridDim.x;

  __shared__ double cache[BLOCK_SIZE];

  double tid_sum = 0.0;
  for (; tid < N; tid += stride)
  {
    tid_sum += x[tid] * y[tid];
  }
  tid = threadIdx.x;
  cache[tid] = tid_sum;

  __syncthreads();
  for (size_t i = blockDim.x/2; i != 0; i /=2)
  {
```

```
    __syncthreads();
    if (tid < i) //lower half does smth, rest idles
      cache[tid] += cache[tid + i]; //lower looks up by stride and sums up
  }

  if(tid == 0) // cache[0] now contains block_sum
  {
    atomicAdd(z, cache[0]);
  }
}

/** Implementation of the conjugate gradient algorithm.
 *
 *  The control flow is handled by the CPU.
 *  Only the individual operations (vector updates, dot products, sparse matrix-vector
     product) are transferred to CUDA kernels.
 *
 *  The temporary arrays p, r, and Ap need to be allocated on the GPU for use with CUDA.
 *  Modify as you see fit.
 */
void conjugate_gradient(const size_t N,  // number of unknows
                        int *csr_rowoffsets, int *csr_colindices, double *csr_values,
                        double *h_rhs,
                        double *h_solution,
                        const double conv_factor)
                        //, double *init_guess)   // feel free to add a nonzero initial
                            guess as needed
{

  // clear solution vector (it may contain garbage values):
  std::fill(h_solution, h_solution + N, 0.0);

  // initialize work vectors:
  double* h_pAp = (double*)malloc(sizeof(double));
  double* h_r2 = (double*)malloc(sizeof(double));
  double* h_r22 = (double*)malloc(sizeof(double));
  double* zero = (double*)malloc(sizeof(double));
  *zero = 0.00;
  *h_pAp = 0.00;
  *h_r2 = 0.00;
  *h_r22 = 0.00;
  double* x;
  double* p;
  double* r;
  double* Ap;
  double* pAp;
  double* r2;

  // arrays
  const size_t arr_size = N*sizeof(double);
  cudaMalloc(&x, arr_size);
  cudaMalloc(&p, arr_size);
  cudaMalloc(&r, arr_size);
  cudaMalloc(&Ap, arr_size);
  // scalars
  cudaMalloc(&pAp, sizeof(double));
  cudaMalloc(&r2, sizeof(double));

  // line 2: initialize r and p:
  //std::copy(h_rhs, h_rhs+N, h_p);
  //std::copy(h_rhs, h_rhs+N, h_r);
  cudaMemcpy(x, h_solution, arr_size, cudaMemcpyHostToDevice);
  cudaMemcpy(r, h_rhs, arr_size, cudaMemcpyHostToDevice);
  cudaMemcpy(Ap, h_rhs, arr_size, cudaMemcpyHostToDevice);
  cudaMemcpy(p, h_rhs, arr_size, cudaMemcpyHostToDevice);

  double alpha, beta;
  int iters = 0;
  //while (1) {
  while (iters < 10000) { // will end with iter == 10'000 or earlier
    cudaMemcpy(r2, zero, sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(pAp, zero, sizeof(double), cudaMemcpyHostToDevice);
```

```
    // 4: Ap = A * p
    csr_Ax<<<GRID_SIZE, BLOCK_SIZE>>>(N, csr_rowoffsets, csr_colindices, csr_values, p, Ap);
    // 5: pAp = <p,Ap>
    xDOTy<<<GRID_SIZE, BLOCK_SIZE>>>(N, p, Ap, pAp);
    // r2 = <r,r>
    xDOTy<<<GRID_SIZE, BLOCK_SIZE>>>(N, r, r, r2);
    cudaDeviceSynchronize();
    cudaMemcpy(h_pAp, pAp, sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(h_r2, r2, sizeof(double), cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();
    // 6: alpha = <r,r>/<p,Ap>
    alpha = (*h_r2) / (*h_pAp);
    // 7: x = x_i+1 = ...
    xADDay<<<GRID_SIZE, BLOCK_SIZE>>>(N, x, p, x, alpha);
    // 8: r = r_i+1 = ...
    xADDay<<<GRID_SIZE, BLOCK_SIZE>>>(N, r, Ap, r, -alpha);

    // 9: r2 = <r,r>
    xDOTy<<<GRID_SIZE, BLOCK_SIZE>>>(N, r, r, r2);
    cudaDeviceSynchronize();
    cudaMemcpy(h_r22, r2, sizeof(double), cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();

    // // 10: check
    if (iters < 10 or iters > 10000 - 10)
      std::cout << "r2[" << iters << "] = " << *h_r2 << " vs " << conv_factor << std::endl;
    if (*h_r22 < conv_factor) {
      break;
    }

    // beta = beta_i = ...
    beta = (*h_r22) / (*h_r2);
    // 10: check
    // if (iters < 10 or iters > 10000 - 10)
    //   std::cout << "r2[" << iters << "] = " << beta << " vs " << conv_factor << std::endl
        ;
    // if (beta < conv_factor or beta > 10) {
    //   break;
    // }
    // 12: p = p_i+1 = ...
    xADDay<<<GRID_SIZE, BLOCK_SIZE>>>(N, r, p, p, beta);
    cudaDeviceSynchronize();

    ++iters;
  }

  if (iters >= 10000)
    std::cout << "Conjugate Gradient did NOT converge within 10000 iterations with r^2 = "
        << *h_r2 << std::endl;
  else
    std::cout << "Conjugate Gradient converged in " << iters << " iterations with r^2 = " <<
         *h_r2 << std::endl;

  cudaMemcpy(h_solution, x, arr_size, cudaMemcpyDeviceToHost);
  cudaDeviceSynchronize();

  cudaFree(x);
  cudaFree(p);
  cudaFree(r);
  cudaFree(Ap);
  cudaFree(pAp);
  cudaFree(r2);
  free(h_pAp);
  free(h_r2);
  free(h_r22);
}


/** Solve a system with `points_per_direction * points_per_direction' unknowns */
void solve_system(size_t points_per_direction) {
```

```cpp
size_t N = points_per_direction * points_per_direction; // number of unknows to solve for

std::cout << "Solving Ax=b with " << N << " unknowns." << std::endl;

//
// Allocate CSR arrays.
//
// Note: Usually one does not know the number of nonzeros in the system matrix a-priori.
//       For this exercise, however, we know that there are at most 5 nonzeros per row in
    the system matrix, so we can allocate accordingly.
//
const size_t size_row = sizeof(int) * (N+1);
const size_t size_col = sizeof(int) * 5 * N;
const size_t size_val = sizeof(double) * 5 * N;
int *h_csr_rowoffsets =    (int*)malloc(size_row);
int *h_csr_colindices =    (int*)malloc(size_col);
double *h_csr_values  = (double*)malloc(size_val);

int* csr_rowoffsets;
int* csr_colindices;
double* csr_values;
cudaMalloc(&csr_rowoffsets, size_row);
cudaMalloc(&csr_colindices, size_col);
cudaMalloc(&csr_values, size_val);

//
// fill CSR matrix with values
//
generate_fdm_laplace(points_per_direction, h_csr_rowoffsets, h_csr_colindices,
    h_csr_values);

cudaMemcpy(csr_rowoffsets, h_csr_rowoffsets, size_row, cudaMemcpyHostToDevice);
cudaMemcpy(csr_colindices, h_csr_colindices, size_col, cudaMemcpyHostToDevice);
cudaMemcpy(csr_values, h_csr_values, size_val, cudaMemcpyHostToDevice);

//
// Allocate solution vector and right hand side:
//
double *solution = (double*)malloc(sizeof(double) * N);
double *rhs      = (double*)malloc(sizeof(double) * N);
std::fill(rhs, rhs + N, 1);

//
// Call Conjugate Gradient implementation (CPU arrays passed here; modify to use GPU
    arrays)
// CSR Matrix is passed as GPU arrays already.
// rhs and solution are CPU arrays.
// This isn't a nice setup obviously...but it's little more than a 1 file "script", so I
    think that's fine for now.
//
double conv_factor = 1e-6; //1e-6
conjugate_gradient(N, csr_rowoffsets, csr_colindices, csr_values, rhs, solution,
    conv_factor);

//
// Check for convergence:
//
double residual_norm = relative_residual(N, h_csr_rowoffsets, h_csr_colindices,
    h_csr_values, rhs, solution);
std::string check = "OK";
if (residual_norm > conv_factor) check = "FAIL";
std::cout << "Relative residual norm: " << residual_norm << " (should be smaller than 1e
    -6): " << check << std::endl;

cudaFree(csr_rowoffsets);
cudaFree(csr_colindices);
cudaFree(csr_values);
free(solution);
free(rhs);
free(h_csr_rowoffsets);
free(h_csr_colindices);
```

```
  free(h_csr_values);
}


int main() {

  solve_system(10); // solves a system with 100*100 unknowns

  return EXIT_SUCCESS;
}
```

Listing 2: Ex3.2 Congugate Gradient, CG.cu

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
#include <string>
#include "poisson2d.hpp"
#include "timer.hpp"

#define BLOCK_SIZE 256
#define GRID_SIZE 256

__global__ void dot_product(double *x, double *y, double *dot, unsigned int n)
{
  unsigned int index = threadIdx.x + blockDim.x*blockIdx.x;
  unsigned int stride = blockDim.x*gridDim.x;

  __shared__ double cache[256];

  double temp = 0.0;
  while(index < n){
    temp += x[index]*y[index];

    index += stride;
  }

  cache[threadIdx.x] = temp;

  __syncthreads();

    for(int i = blockDim.x/2; i>0; i/=2)
    {
        __syncthreads();
        if(threadIdx.x < i)
            cache[threadIdx.x] += cache[threadIdx.x + i];
    }

  if(threadIdx.x == 0){
    atomicAdd(dot, cache[0]);
  }
}

__global__ void xADDay(const size_t N, double *x, double *y, double *z, const double alpha)
{
  const size_t stride = blockDim.x * gridDim.x;
  for(size_t i = threadIdx.x + blockIdx.x * blockDim.x; i < N; i += stride)
      z[i] = x[i] + alpha * y[i];
}

__global__ void xDOTy(const size_t N, double* x, double* y, double* z)
{
  size_t tid = threadIdx.x + blockDim.x* blockIdx.x;
  size_t stride = blockDim.x* gridDim.x;

  __shared__ double cache[BLOCK_SIZE];

  double tid_sum = 0.0;
  for (; tid < N; tid += stride)
  {
    tid_sum += x[tid] * y[tid];
```

```
  }
  tid = threadIdx.x;
  cache[tid] = tid_sum;

  __syncthreads();
  for (size_t i = blockDim.x/2; i != 0; i /=2)
  {
    __syncthreads();
    if (tid < i) //lower half does smth, rest idles
      cache[tid] += cache[tid + i]; //lower looks up by stride and sums up
  }

  if(tid == 0) // cache[0] now contains block_sum
  {
    atomicAdd(z, cache[0]);
  }
}

int main() {

  int N = 256;

  double xInit = 1.;
  double alpha = 2.;
  double yInit = 2.5;

  double *x = (double*)malloc(sizeof(double) * N);
  double *y = (double*)malloc(sizeof(double) * N);
  double *z = (double*)malloc(sizeof(double) * N);
  double *Dot = (double*)malloc(sizeof(double));
  *Dot = -1.;
  std::fill(x, x + N, xInit);
  std::fill(y, y + N, yInit);
  std::fill(z, z + N, 0.0);

  double *px, *py, *pz, *pDot;
  cudaMalloc(&px, N*sizeof(double));
  cudaMalloc(&py, N*sizeof(double));
  cudaMalloc(&pz, N*sizeof(double));
  cudaMalloc(&pDot, sizeof(double));
  cudaMemcpy(px, x, N*sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(py, y, N*sizeof(double), cudaMemcpyHostToDevice);
  //cudaMemcpy(pz, z, N*sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(pDot, Dot, sizeof(double), cudaMemcpyHostToDevice);

  xADDay<<<GRID_SIZE, BLOCK_SIZE>>>(N, px, py, pz, alpha);
  cudaDeviceSynchronize();
  xDOTy<<<GRID_SIZE, BLOCK_SIZE>>>(N, px, py, pDot);
  //dot_product<<<GRID_SIZE, BLOCK_SIZE>>>(px, py, pDot, N);
  cudaDeviceSynchronize();

  cudaMemcpy(z, pz, N*sizeof(double), cudaMemcpyDeviceToHost);
  cudaMemcpy(Dot, pDot, sizeof(double), cudaMemcpyDeviceToHost);
  cudaDeviceSynchronize();

  std::cout << "Checking xADDay..." << std::endl;
  int cnt = 0;
  for (int i = 0; i < N; ++i)
    if (z[i] != xInit + alpha*yInit) ++cnt;

  if (cnt)
    std::cout << "Something went wrong...let's see:" << std::endl;
  else
    std::cout << "Everything ok, see:" << std::endl;

  for (int i = 0; i < 5; ++i)
    std::cout << "z[" << i << "] = " << z[i] << std::endl;
  std::cout << "..." << std::endl;
  for (int i = N-1-5; i < N; ++i)
    std::cout << "z[" << i << "] = " << z[i] << std::endl;

  std::cout << "------------------------------------" << std::endl;
```

```
  std::cout << "Checking xDOTy..." << std::endl;
  if (*Dot != xInit*yInit*N)
    std::cout << "NOPE: " << *Dot << " != " << xInit*yInit*N << std::endl;
  else
    std::cout << "OK: " << *Dot << " == " << xInit*yInit*N << std::endl;

  free(x);
  free(y);
  free(z);
  free(Dot);
  cudaFree(px);
  cudaFree(py);
  cudaFree(pz);
  cudaFree(pDot);


  return EXIT_SUCCESS;
}
```

Listing 3: Ex3.2 Kernel tests, kernelTests.cu