LAB 3
DUE DATE: Fri 13 Oct 5pm (upload to polylearn)
Names in Group (max 2): Liam Lefferts

---

1. Write a program in OpenMP to accelerate Matrix Multiplication
Compare timings between your Sequential, Pthreads and OpenMP implementations

code	Sequential	Pthreads (4 threads)	OpenMP (4 threads)
512*512	.741	.216	.597

2. Accelerate the following program in OpenMP (diffussion_opmvec.c) in the attached file

Sequential	OpenMP
ljcates@127x19 ~/419/Labs/Lab3 $./diffusion_baseline Running diffusion kernel 6553 times Elapsed time : 444.197 (s) FLOPS : 3217.566 (MFlops) Throughput : 2.970 (GB/s) Accuracy : 2.074976e-05	ljcates@127x19 ~/419/Labs/Lab3 $export OMP_NUM_THREADS=80 ljcates@127x19 ~/419/Labs/Lab3 $./diffusion_tiled Running diffusion kernel 6553 times Elapsed time : 25.115 (s) FLOPS : 56907.066 (MFlops) Throughput : 52.530 (GB/s) Accuracy : 2.074976e-05

Explanation of pragmas

Like previous sequential code, we begin by applying the two key factors associated with parallel programming, scaling and vectorizing.

First, we scale the code using OpenMP. Our edits apply OpenMP directives to distribute and scale our computations across all available cores and threads. The key OpenMP directive appears before our z loop:

#pragma omp for collapse(2)

This clause tells the compiler to collapse the next two loops (z and y) and then apply the OpenMP "omp for" work scheduling mechanism. Conceptually, the for loop changes to a single loop that executes as for(yz = 0; yz < ny*nx; ++yz) with the associated similar implied mapping for the use of y and z in the body of the loop. This will enable each thread to be assigned larger chunks of data to process more calculations on each loop execution.

Secondly, we manually search for vectorization opportunities. The compiler was unable to automatically vectorize our code. This occurance is common if our implementation uses or reuses multiple pointer variables and array values. The compiler believes it is unable to assure correctness as the value of enclosed expressions seem to be dependent on vector lanes to be calculated simultaneously.

The innermost loop in the diffusion() function uses two temporary array pointers heavily, f1_t and f2_t, thus the compiler cannot confirm independence. With situations like this, when a developer is sure no dependency exists, we use:

#pragma simd"

requesting the compiler to vectorize the following loops regardless of potential dependencies or other potential constraints, as we have with the x loop.

We now have both scaled and vectorized our code, and we have seen very significant performance improvement over the baseline. Reviewing the code further, the x boundary check is found to be superfluous and slowing. A stencil altering boundry is encountered rarely for volumes of significant size. Since only our starting and ending x coordinates 0 and nx - 1 will hit the boundary condition, we can create an inner loop without any boundary checks by simply ensuring we process x indices from 1 to nx - 2. Furthermore, since the stencil always traverses in single units across the x row of sub-volumes, we can update the stencil positions by simply incrementing them. Also, we can eliminate calculating the east and west locations by referencing their positions directly in the array index (e = c - 1 and w = c + 1).

Our final improvement focuses on increasing cache hits during data access. The data access patterns of stencil operations, like the kind we use, typically exploit data locality. Tiling and blocking are terms describing the technique often used for improving data reuse in cache architectures by increasing locality of a block's memory requirements. Generally “least recently used” (LRU) methods are used to

swap data from the cache as required by subsequent cache accesses. Since memory accesses are significant, reusing data in cache lines local to the current code sequence is important for efficiency.

In our diffusion stencil code, each innermost loop iteration processes the x elements of a y row sequentially, then moving to the following y row. Ignoring the work division from scheduling multiple threads for a moment, there is a high likelihood of accessing data in the L1 or L2 cache we have used before from the current and previous y rows since our access of those y data is recent. We can also noticed their is low likelyhood of z data reuse as the bottom and top row data on the adjacent z plane are used once and then not accessed again until the next full y plane is processed at the same row. If we consider processing a rectangular tile of y—actually a slab of yx values across a range of z. The top row in a given z iteration will still be in cache to serve as left, center, and right rows for the next z, and the bottom row for the z after that, avoiding unneccessary memory requests.

We tile the diffusion() function by selecting a blocking factor value "YBF" for the number of y rows to process in each slab; the optimal number will maintain the sufficient amounts of y and z data in the cache long enough to be reused during computation. Since we will be processing a portion or tile of y rows across the full z dimension, we add an outer y loop to control stepping to the start of each tile. The inner y loop is then adjusted to perform the per-row processing within the current tile. The x processing with the peeled out boundary management is maintained so we keep that optimization intact.

These are a few examples of “parallel thinking” applied to real-world serial implementations to produce high-speed parallel code. Code that remains general and portable while scaling across cores and threads, as well as taking advantage of vectorization and cache architecture. We have found that, when the investment is made in optimizing code for parallel techniques such as these performance increases hundreds or even thousands of times.

```
gcc -fopenmp -std=c99 -O3 diffusion_tiled.c -o diffusion_tiled -lm
export OMP_NUM_THREADS=80
export KMP_AFFINITY=scatter
./diffusion_tiled
```

Code Appendix

```
//matrix.c
//_____________________________________________________________________________
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>
#include <pthread.h>


/*global variable accesible to all threads*/
long threads_count;
int SIZE, NTHREADS;
int **A, **B, **C, **D, **E;

/* dtime -
 *  * utility routine to return the current wall clock time
 *   */
double dtime()
{
  double tseconds = 0.0;
  struct timeval mytime;
  gettimeofday(&mytime,(struct timezone*)0);
  tseconds = (double)(mytime.tv_sec + mytime.tv_usec*1.0e-6);
  return( tseconds );
}


void init()
{
  int i, j;
  time_t t;

  A = (int**)malloc(SIZE * sizeof(int *));
  for(i = 0; i < SIZE; i++)
    A[i] = malloc(SIZE * sizeof(int));

  B = (int**)malloc(SIZE * sizeof(int *));
  for(i = 0; i < SIZE; i++)
    B[i] = malloc(SIZE * sizeof(int));

  C = (int**)malloc(SIZE * sizeof(int *));
  for(i = 0; i < SIZE; i++)
    C[i] = malloc(SIZE * sizeof(int));

  D = (int**)malloc(SIZE * sizeof(int *));
  for(i = 0; i < SIZE; i++)
```

```
//matrix.c
//_____________________________________________________________________________
D[i] = malloc(SIZE * sizeof(int));

  E = (int**)malloc(SIZE * sizeof(int *));
  for(i = 0; i < SIZE; i++)
    E[i] = malloc(SIZE * sizeof(int));
  srand(time(&t));

  for(i = 0; i < SIZE; i++) {
    for(j = 0; j < SIZE; j++) {
      A[i][j] = rand()%100;
      B[i][j] = rand()%100;
    }
  }
}

/* Matrix Multiplication non threaded*/
void matrixmult(){
  int c, d, k, sum;
  for (c = 0; c < SIZE; c++) {
    for (d = 0; d < SIZE; d++) {
      sum = 0;
      for (k = 0; k < SIZE; k++)
        sum += A[c][k]*B[k][d];

      C[c][d] = sum;
    }
  }
}

/*Matrix multiplication threaded */
void* matrixmultT(void* id){

  int i, j, k, sum;
  int tid = (int)id;
  int start = tid * SIZE/NTHREADS;
  int end = (tid+1) * (SIZE/NTHREADS) - 1;

  for(i = start; i <= end; i++) {
    for(j = 0; j < SIZE; j++) {
      sum = 0;
      for(k = 0; k < SIZE; k++)
        sum += A[i][k] * B[k][j];

      D[i][j] = sum;
    }
  }}
```

```
//matrix.c
//_____________________________________________________________________________

int matrixmultMP()
{
  int i,j,k,sum;

#pragma omp parallel
  {
#pragma omp for schedule(static)

    for (i = 0; i < SIZE; i++) {
      for (j = 0; j < SIZE; j++) {
        sum = 0;
        for (k = 0; k < SIZE; k++)
          sum += A[i][k] * B[k][j];
        E[i][j] = sum;
      }
    }
  }
  return 0;
}

int main(int argc, char *argv[] )
{
  int i,j;
  double tstart, tstop, ttime;
  pthread_t* threads;

  if(argc != 3)
  {
    printf("Usage: %s <size_of_square_matrix> <number_of_threads>\n", argv[0]);
    exit(1);
  }

  SIZE = atoi(argv[1]);
  init();
  printf("Starting Compute\r\n");

  tstart = dtime();
  matrixmult();
  tstop = dtime();
  ttime = tstop - tstart;

  if ((ttime) > 0.0)
    printf("Secs Serial = %10.3lf\n",ttime);
```

```
//matrix.c
//_____________________________________________________________________________

  /*threaded part*/

  NTHREADS = atoi(argv[2]);
  threads = (pthread_t*)malloc(NTHREADS * sizeof(pthread_t));

  tstart = dtime();

  for(i = 0; i < NTHREADS; i++)
    pthread_create(&threads[i], NULL, matrixmultT, (void *)i);
  for(i = 0; i < NTHREADS; i++)
    pthread_join(threads[i], NULL);

  free(threads);

  tstop=dtime();
  ttime=tstop-tstart;
  if ((ttime) > 0.0)
    printf("Secs Threaded = %10.3lf\n", ttime);

  /*OpenMP part*/

  tstart = dtime();

  matrixmultMP();

  tstop=dtime();
  ttime=tstop-tstart;
  if ((ttime) > 0.0)
    printf("Secs OpenMP = %10.3lf\n", ttime);

  /*print sanity check /
   for (i=0; i<SIZE; i++) {
   for(j=0; j<SIZE; j++)
   printf("%d, %d ", C[i][j], D[i][j]);
   printf("\n");
   }

  /*check solutions*/
  float dif, accum=0;
  for (i=0; i<SIZE; i++){
    for(j=0; j<SIZE; j++) {
      dif=abs(C[i][j]-D[i][j]);
      if(dif!=0) accum+=dif;
    }
  }
```

```
//matrix.c
//_____________________________________________________________________________

  if(accum < 0.1) printf("SUCESS\n");
  else printf("FAIL\n");

  /*check solutions*/
  accum=0;
  for (i=0; i<SIZE; i++){
    for(j=0; j<SIZE; j++) {
      dif=abs(C[i][j]-E[i][j]);
      if(dif!=0){
        accum+=dif;
         printf("%d ",dif);
      }
    }
  }
  if(accum < 0.1) printf("SUCESS\n");
  else printf("FAIL\n");

  /*CleanUp*/
  for(i = 0; i < SIZE; i++)
    free((void *)A[i]);
  free((void *)A);

  for(i = 0; i < SIZE; i++)
    free((void *)B[i]);
  free((void *)B);

  for(i = 0; i < SIZE; i++)
    free((void *)C[i]);
  free((void *)C);

  for(i = 0; i < SIZE; i++)
    free((void *)D[i]);
  free((void *)D);

  for(i = 0; i < SIZE; i++)
    free((void *)E[i]);
  free((void *)E);


  return( 0 );
}
```

diffusion_baseline.c

```
#include <string.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <omp.h>
#include <assert.h>
#include <sys/mman.h>

#define REAL float
#define NX (256)
#define NXP nx

#ifndef M_PI
#define M_PI (3.14159265358979323846)
#endif


void init(REAL *buff, const int nx, const int ny, const int nz,
     const REAL kx, const REAL ky, const REAL kz,
     const REAL dx, const REAL dy, const REAL dz,
     const REAL kappa, const REAL time) {
  REAL ax, ay, az;
  int jz, jy, jx;
  ax = exp(-kappa*time*(kx*kx));
  ay = exp(-kappa*time*(ky*ky));
  az = exp(-kappa*time*(kz*kz));
  for (jz = 0; jz < nz; jz++) {
    for (jy = 0; jy < ny; jy++) {
      for (jx = 0; jx < nx; jx++) {
        int j = jz*NXP*ny + jy*NXP + jx;
        REAL x = dx*((REAL)(jx + 0.5));
        REAL y = dy*((REAL)(jy + 0.5));
        REAL z = dz*((REAL)(jz + 0.5));
        REAL f0 = (REAL)0.125
          *(1.0 - ax*cos(kx*x))
          *(1.0 - ay*cos(ky*y))
          *(1.0 - az*cos(kz*z));
        buff[j] = f0;
      }
    }
  }
}
```

diffusion_baseline.c

```
REAL accuracy(const REAL *b1, REAL *b2, const int len) {
  REAL err = 0.0;
  int i;
  for (i = 0; i < len; i++) {
    err += (b1[i] - b2[i]) * (b1[i] - b2[i]);
  }
  return (REAL)sqrt(err/len);
}

void
diffusion_openmpv(REAL *restrict f1, REAL *restrict f2, int nx, int ny, int nz,
    REAL ce, REAL cw, REAL cn, REAL cs, REAL ct,
    REAL cb, REAL cc, REAL dt, int count) {
  {
    REAL *f1_t = f1;
    REAL *f2_t = f2;

    for (int i = 0; i < count; ++i) {
      for (int z = 0; z < nz; z++) {
        for (int y = 0; y < ny; y++) {
          for (int x = 0; x < nx; x++) {
            int c, w, e, n, s, b, t;
            c =  x + y * NXP + z * NXP * ny;
            w = (x == 0)    ? c : c - 1;
            e = (x == NXP-1) ? c : c + 1;
            n = (y == 0)    ? c : c - NXP;
            s = (y == ny-1) ? c : c + NXP;
            b = (z == 0)    ? c : c - NXP * ny;
            t = (z == nz-1) ? c : c + NXP * ny;
            f2_t[c] = cc * f1_t[c] + cw * f1_t[w] + ce * f1_t[e]
              + cs * f1_t[s] + cn * f1_t[n] + cb * f1_t[b] + ct * f1_t[t];
          }
        }
      }
      REAL *t = f1_t;
      f1_t = f2_t;
      f2_t = t;
    }
  }
  return;
}
```

diffusion_baseline.c

```
static double cur_second(void) {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return (double)tv.tv_sec + (double)tv.tv_usec / 1000000.0;
}


void dump_result(REAL *f, int nx, int ny, int nz, char *out_path) {
  FILE *out = fopen(out_path, "w");
  assert(out);
  size_t nitems = nx * ny * nz;
  fwrite(f, sizeof(REAL), nitems, out);
  fclose(out);
}

int main(int argc, char *argv[])
{

  struct timeval time_begin, time_end;

  int    nx   = NX;
  int    ny   = NX;
  int    nz   = NX;

  REAL *f1 = (REAL *)malloc(sizeof(REAL)*NX*NX*NX);
  REAL *f2 = (REAL *)malloc(sizeof(REAL)*NX*NX*NX);
  assert(f1 != MAP_FAILED);
  assert(f2 != MAP_FAILED);
  REAL *answer = (REAL *)malloc(sizeof(REAL) * NXP*ny*nz);
  REAL *f_final = NULL;

  REAL   time  = 0.0;
  int    count = 0;
  int    nthreads;

  REAL l, dx, dy, dz, kx, ky, kz, kappa, dt;
  REAL ce, cw, cn, cs, ct, cb, cc;

  nthreads = omp_get_num_threads();

  l = 1.0;
  kappa = 0.1;
  dx = dy = dz = l / nx;
  kx = ky = kz = 2.0 * M_PI;
  dt = 0.1*dx*dx / kappa;
```

```
  count = 0.1 / dt;
  f_final = (count % 2)? f2 : f1;

  init(f1, nx, ny, nz, kx, ky, kz, dx, dy, dz, kappa, time);

  ce = cw = kappa*dt/(dx*dx);
  cn = cs = kappa*dt/(dy*dy);
  ct = cb = kappa*dt/(dz*dz);
  cc = 1.0 - (ce + cw + cn + cs + ct + cb);

  printf("Running diffusion kernel %d times\n", count); fflush(stdout);
  gettimeofday(&time_begin, NULL);
  diffusion_openmpv(f1, f2, nx, ny, nz, ce, cw, cn, cs, ct, cb, cc,
      dt, count);
  gettimeofday(&time_end, NULL);
  time = count * dt;
  dump_result(f_final, nx, ny, nz, "diffusion_result_baseline.dat");

  init(answer, nx, ny, nz, kx, ky, kz, dx, dy, dz, kappa, time);
  REAL err = accuracy(f_final, answer, nx*ny*nz);
  double elapsed_time = (time_end.tv_sec - time_begin.tv_sec)
    + (time_end.tv_usec - time_begin.tv_usec)*1.0e-6;
  REAL mflops = (nx*ny*nz)*13.0*count/elapsed_time * 1.0e-06;
  double thput = (nx * ny * nz) * sizeof(REAL) * 3.0 * count
    / elapsed_time * 1.0e-09;

  fprintf(stderr, "Elapsed time : %.3f (s)\n", elapsed_time);
  fprintf(stderr, "FLOPS        : %.3f (MFlops)\n", mflops);
  fprintf(stderr, "Throughput   : %.3f (GB/s)\n", thput);
  fprintf(stderr, "Accuracy     : %e\n", err);

  free(f1);
  free(f2);
  return 0;
}
```

diffusion_tiled.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <omp.h>
#include <assert.h>
#include <sys/mman.h>

#define REAL float
#define NX (256)
#define NXP nx

#ifndef M_PI
#define M_PI (3.141592653589793238626)
#endif


void init(REAL *buff, const int nx, const int ny, const int nz,
     const REAL kx, const REAL ky, const REAL kz,
     const REAL dx, const REAL dy, const REAL dz,
     const REAL kappa, const REAL time) {
  REAL ax, ay, az;
  int jz, jy, jx;
  ax = exp(-kappa*time*(kx*kx));
  ay = exp(-kappa*time*(ky*ky));
  az = exp(-kappa*time*(kz*kz));
  for (jz = 0; jz < nz; jz++) {
    for (jy = 0; jy < ny; jy++) {
      for (jx = 0; jx < nx; jx++) {
        int j = jz*NXP*ny + jy*NXP + jx;
        REAL x = dx*((REAL)(jx + 0.5));
        REAL y = dy*((REAL)(jy + 0.5));
        REAL z = dz*((REAL)(jz + 0.5));
        REAL f0 = (REAL)0.125
          *(1.0 - ax*cos(kx*x))
          *(1.0 - ay*cos(ky*y))
          *(1.0 - az*cos(kz*z));
        buff[j] = f0;
      }
    }
  }
}

REAL accuracy(const REAL *b1, REAL *b2, const int len) {
```

```
  REAL err = 0.0;
  int i;
  for (i = 0; i < len; i++) {
    err += (b1[i] - b2[i]) * (b1[i] - b2[i]);
  }
  return (REAL)sqrt(err/len);
}

void diffusion_tiled(REAL *restrict f1, REAL *restrict f2,
    int nx, int ny, int nz,
    REAL ce, REAL cw, REAL cn, REAL cs, REAL ct,
    REAL cb, REAL cc, REAL dt, int count) {

  unsigned long tsc;
  int nthreads;

#pragma omp parallel
  {
    REAL *f1_t = f1;
    REAL *f2_t = f2;
    int mythread;

    for (int i = 0; i < count; ++i) {
#define YBF 16
#pragma omp for collapse(2)
      for (int yy = 0; yy < ny; yy += YBF) {
        for (int z = 0; z < nz; z++) {
          int ymax = yy + YBF;
          if (ymax >= ny) ymax = ny;
          for (int y = yy; y < ymax; y++) {
            int x;
            int c, n, s, b, t;
            x = 0;
            c =  x + y * NXP + z * NXP * ny;
            n = (y == 0)    ? c : c - NXP;
            s = (y == ny-1) ? c : c + NXP;
            b = (z == 0)    ? c : c - NXP * ny;
            t = (z == nz-1) ? c : c + NXP * ny;
            f2_t[c] = cc * f1_t[c] + cw * f1_t[c] + ce * f1_t[c+1]
              + cs * f1_t[s] + cn * f1_t[n] + cb * f1_t[b] + ct * f1_t[t];
#pragma simd
            for (x = 1; x < nx-1; x++) {
              ++c;
              ++n;
              ++s;
              ++b;
              ++t;
```

```
//diffusion_tiled.c
//_____________________________________________________________________________

                f2_t[c] = cc * f1_t[c] + cw * f1_t[c-1] + ce * f1_t[c+1]
                  + cs * f1_t[s] + cn * f1_t[n] + cb * f1_t[b] + ct * f1_t[t];
              }
              ++c;
              ++n;
              ++s;
              ++b;
              ++t;
              f2_t[c] = cc * f1_t[c] + cw * f1_t[c-1] + ce * f1_t[c]
                + cs * f1_t[s] + cn * f1_t[n] + cb * f1_t[b] + ct * f1_t[t];
            }
          }
        }
        REAL *t = f1_t;
        f1_t = f2_t;
        f2_t = t;
      }
    }
    return;
}

static double cur_second(void) {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return (double)tv.tv_sec + (double)tv.tv_usec / 1000000.0;
}


void dump_result(REAL *f, int nx, int ny, int nz, char *out_path) {
  FILE *out = fopen(out_path, "w");
  assert(out);
  size_t nitems = nx * ny * nz;
  fwrite(f, sizeof(REAL), nitems, out);
  fclose(out);
}

int main(int argc, char *argv[])
{

  struct timeval time_begin, time_end;

  int    nx   = NX;
  int    ny   = NX;
  int    nz   = NX;
```

```
// diffusion_tiled.c
//_____________________________________________________________________________

 REAL *f1 = (REAL *)malloc(sizeof(REAL)*NX*NX*NX);
  REAL *f2 = (REAL *)malloc(sizeof(REAL)*NX*NX*NX);
  assert(f1 != MAP_FAILED);
  assert(f2 != MAP_FAILED);
  REAL *answer = (REAL *)malloc(sizeof(REAL) * NXP*ny*nz);
  REAL *f_final = NULL;

  REAL   time  = 0.0;
  int    count = 0;
  int    nthreads;

  REAL l, dx, dy, dz, kx, ky, kz, kappa, dt;
  REAL ce, cw, cn, cs, ct, cb, cc;

#pragma omp parallel
#pragma omp master
  nthreads = omp_get_num_threads();


  l = 1.0;
  kappa = 0.1;
  dx = dy = dz = l / nx;
  kx = ky = kz = 2.0 * M_PI;
  dt = 0.1*dx*dx / kappa;
  count = 0.1 / dt;

  f_final = (count % 2)? f2 : f1;

  init(f1, nx, ny, nz, kx, ky, kz, dx, dy, dz, kappa, time);

  ce = cw = kappa*dt/(dx*dx);
  cn = cs = kappa*dt/(dy*dy);
  ct = cb = kappa*dt/(dz*dz);
  cc = 1.0 - (ce + cw + cn + cs + ct + cb);

  printf("Running diffusion kernel %d times\n", count); fflush(stdout);
  gettimeofday(&time_begin, NULL);
  diffusion_tiled(f1, f2, nx, ny, nz, ce, cw, cn, cs, ct, cb, cc,
      dt, count);
  gettimeofday(&time_end, NULL);
  time = count * dt;
  dump_result(f_final, nx, ny, nz, "diffusion_result.dat");

  init(answer, nx, ny, nz, kx, ky, kz, dx, dy, dz, kappa, time);
  REAL err = accuracy(f_final, answer, nx*ny*nz);
```

```
//diffusion_tiled.c
//_____________________________________________________________________________

 double elapsed_time = (time_end.tv_sec - time_begin.tv_sec)
     + (time_end.tv_usec - time_begin.tv_usec)*1.0e-6;
  REAL mflops = (nx*ny*nz)*13.0*count/elapsed_time * 1.0e-06;
  double thput = (nx * ny * nz) * sizeof(REAL) * 3.0 * count
    / elapsed_time * 1.0e-09;

  fprintf(stderr, "Elapsed time : %.3f (s)\n", elapsed_time);
  fprintf(stderr, "FLOPS        : %.3f (MFlops)\n", mflops);
  fprintf(stderr, "Throughput   : %.3f (GB/s)\n", thput);
  fprintf(stderr, "Accuracy     : %e\n", err);

  free(f1);
  free(f2);
  return 0;
}
```