TAC-HEP GPU Course Final Project - Michael Martinez

Code can be found here on Github

Part 1: CPU code to stencil and multiply two matrices

```
#include <stdlib.h>
#include <iostream>
#include <time.h>

#define DSIZE 512
#define RADIUS 2

float dotprod(int* A, int* B, int size){
    float sum = 0;
    for (int i=0; i<size; i++){
        sum += (A[i])*(B[i]);
    }
    return sum;
}

void matmult(int* A, int* B, int* result, int nrows, int ncols){
    for (int i=0; i<ncols; i++){
        int v1[nrows];
        for (int j=0; j<nrows; j++){
            //first construct column
            v1[j] = B[j*ncols + i];

        }
        for (int j=0; j<nrows; j++){
            //construct row vector
            int v2[ncols];
            for (int k=0; k<ncols; k++){
                v2[k] = A[j*ncols+k];
            }
            //now do matrix multiplication
            result[j*ncols+i] = dotprod(v1,v2,sizeof(v1)/sizeof(int));
        }
    }
}

void stencil(int* in, int* out, int radius, int size){
    for (int i=0; i<size; i++){
        for (int j=0; j<size; j++){
            int result = 0;
            if ((std::abs((i+radius)%(size-1)-radius) >= radius) &&
(std::abs((j+radius)%(size-1)-radius) >= radius)){
                for (int k=-1*radius; k<=radius; k++){
                    result += in[(j + k)*size + i]; //vertical
                    result += in[j * size + (i + k)]; //horizontal
                }
                result -= in[j*size + i];
                out[j*size + i] = result;
            }
            else {
                out[j*size + i] = in[j*size + i];
            }
        }
    }
}
```

```
/*
 A: construct 2 square matrices, A and B (size >= 512) with integer values
 B: run stencil on A and B (radius >= 2)
 C: multiply A*B
 D: check results by writing tests
     -check corners, edges, middle for stencil
     -check matrix mult
*/

int main(){

    double t0, t3, t3sum;

    t0 = clock();

    int A[DSIZE*DSIZE];
    int B[DSIZE*DSIZE];
    int Astencil[DSIZE*DSIZE];
    int Bstencil[DSIZE*DSIZE];
    int C[DSIZE*DSIZE];

    for (int i=0; i<DSIZE*DSIZE; i++){
        A[i] = 1;
        B[i] = 3;
    }

    stencil(A, Astencil, RADIUS, DSIZE);
    stencil(B, Bstencil, RADIUS, DSIZE);

    matmult(Astencil, Bstencil, C, DSIZE, DSIZE);


    //check: I generated the correct answers and chose a few spots on the edges
    //       and center etc to check the c/cuda/etc answers are correct

    if (A[0] != 1){ //check array A is constructed properly aka rand() worked
        std::cout << "Error in generating Array A, value (0,0) should be 1, returned
" << A[0];
        return 1;
    }
    if (B[0] != 3){ //check array B is constructed properly aka rand() worked
        std::cout << "Error in generating Array B, value (0,0) should be 3, value is
" << B[0];
        return 1;
    }
    if (Astencil[0] != 1){ //confirm stencil did nothing to corner values on A
        std::cout << "Stencil not applied properly on A, value (0,0) should be 1,
returned " << Astencil[0];
        return 1;
    }
    if (Astencil[1*DSIZE + 157] != 1){ //confirm stencil did nothing to edge values
on A
        std::cout << "Stencil not applied properly on A, value (1,157) should be 1,
returned " << Astencil[1*DSIZE + 150];
        return 1;
    }
    if (Astencil[342*DSIZE + 58] != 9){ //confirm stencil worked in center on A
        std::cout << "Stencil not applied properly on A, value (342,58) should be 9,
returned " << Astencil[342*DSIZE + 58];
        return 1;
    }
    if (Bstencil[0] != 3){ //confirm stencil did nothing to corner values on B
        std::cout << "Stencil not applied properly on B, value (0,0) should be 3,
returned " << Bstencil[0];
        return 1;
    }
```

```
if (Bstencil[1*DSIZE + 157] != 3){ //confirm stencil did nothing to edge values on B
        std::cout << "Stencil not applied properly on B, value (1,157) should be 3,
returned " << Bstencil[1*DSIZE + 150];
        return 1;
    }
    if (Bstencil[342*DSIZE + 58] != 27){ //confirm stencil worked in center on B
        std::cout << "Stencil not applied properly on B, value (342,58) should be
27, returned " << Bstencil[342*DSIZE + 58];
        return 1;
    }
    if (C[0] != 1536){ //confirm matrix multiplication worked on corner
        std::cout << "Matrix multiplication incorrect, value (0,0) should be 1536,
returned " << C[0];
        return 1;
    }
    if (C[1*DSIZE + 157] != 13728){ //confirm matrix multiplication worked on edge
        std::cout << "Matrix multiplication incorrect, value (157,0) should be
13728, returned " << C[1*DSIZE + 157];
        return 1;
    }
    if (C[235*DSIZE+461] != 123456){ //confirm matrix multiplication worked in
center
        std::cout << "Matrix multiplication incorrect, value (235,461) should be
123456, returned " << C[35*DSIZE+61];
        return 1;
    }

    std::cout << "Success!\n";

    // CPU timing
    t3 = clock();
    t3sum = ((double)(t3-t0))/CLOCKS_PER_SEC;
    printf("Compute took %f seconds\n", t3sum);

}
```

Note: In all source code I included a timing utility to get an idea of runtime without the overhead generated by profiling.

Profiling c++ code with VTune

Function	CPU Time (s)	Function (Full)
dotprod	0.560041	dotprod(int*, int*, int)
matmult	0.369926	matmult(int*, int*, int*, int, int)
stencil	0.022992	stencil(int*, int*, int, int)
main	0.003008	main

This table was generated by VTune's hotspot generator. I removed some of the columns for readability and simplicity's sake but the entire csv file and its .txt version can be found in the supporting files. It's clear the matrix multiplication and dot product supporting function are the most intensive parts of the computation. Total execution time of this file varied, but was generally around 1.03 seconds.

Part 2: Porting the functions to CUDA

```
#include <stdlib.h>
#include <iostream>
#include <algorithm>
#include <time.h>

#define N 512
#define DSIZE 512
#define RADIUS 2
#define BLOCK_SIZE 32

// error checking macro
#define cudaCheckErrors(msg)                                    \
   do {                                                         \
       cudaError_t __err = cudaGetLastError();                  \
       if (__err != cudaSuccess) {                              \
           fprintf(stderr, "Fatal error: %s (%s at %s:%d)\n",   \
                   msg, cudaGetErrorString(__err),              \
                   __FILE__, __LINE__);                         \
           fprintf(stderr, "*** FAILED - ABORTING\n");          \
           exit(1);                                             \
       }                                                        \
   } while (0)

__global__ void stencil_2d(int *in, int *out, int size) {

    int gindex_x = blockIdx.x*blockDim.x + threadIdx.x;
    int gindex_y = blockIdx.y*blockDim.y + threadIdx.y;
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS][BLOCK_SIZE + 2 * RADIUS];
    int lindex_x = threadIdx.x + RADIUS;
    int lindex_y = threadIdx.y + RADIUS;

    // Read input elements into shared memory
    temp[lindex_x][lindex_y] = in[gindex_y+size*gindex_x];

    if ((threadIdx.x < RADIUS)&&(gindex_x >= RADIUS)){
        temp[lindex_x-RADIUS][lindex_y] = in[gindex_y + (size*(gindex_x - RADIUS))];
        temp[lindex_x+BLOCK_SIZE][lindex_y]=in[gindex_y+size*(gindex_x+BLOCK_SIZE)];
    }
    if ((threadIdx.y < RADIUS)&&(gindex_x >= RADIUS)){
        temp[lindex_x][lindex_y - RADIUS] = in[gindex_y - RADIUS + size*gindex_x];
        temp[lindex_x][lindex_y + BLOCK_SIZE] = in[gindex_y + BLOCK_SIZE +
size*gindex_x];
    }
    __syncthreads();

    if ((std::abs((gindex_x+RADIUS)%(size-1)-RADIUS) >= RADIUS) &&
(std::abs((gindex_y+RADIUS)%(size-1)-RADIUS) >= RADIUS)){

        // Apply the stencil
        int result = -1*temp[lindex_x][lindex_y];

        for (int offset = -1*RADIUS; offset <= RADIUS; offset++){
            result += temp[lindex_x + offset][lindex_y];
            result += temp[lindex_x][lindex_y + offset];
        }
    __syncthreads();

        // Store the result
        out[gindex_y+size*gindex_x] = result;

    }
    else{
        out[gindex_y +size*gindex_x] = in[gindex_y +size*gindex_x];
    }
   }
```

```
    // Square matrix multiplication on GPU : C = A * B
__global__ void matrix_mul_gpu(const int *A, const int *B, int *C, int size) {

        // create thread x index
        // create thread y index
        int idx = threadIdx.x + blockDim.x * blockIdx.x;
        int idy = threadIdx.y + blockDim.y * blockIdx.y;
        // Make sure we are not out of range
        if ((idx < size) && (idy < size)) {
            float temp = 0;
            for (int i = 0; i < size; i++){
                temp += (A[size*idx + i]*B[idy + size*i]);
            }
            C[idy*size+idx] = temp;
        }

}

int main(void) {

    double t0, t3, t3sum;


    t0 = clock();

    //device copies of the intermediate stencil results arent necessary
    //but they will be good to have for debugging


    int *A, *B, *Astencil, *Bstencil, *C;
    int *dA, *dB, *dAstencil, *dBstencil, *dC;
    int size = (N)*(N) * sizeof(int);

    A = (int *)malloc(size);
    B = (int *)malloc(size);
    Astencil = (int *)malloc(size);
    Bstencil = (int *)malloc(size);
    C = (int *)malloc(size);

    cudaMalloc((void **)&dA, size);
    cudaMalloc((void **)&dB, size);
    cudaMalloc((void **)&dAstencil, size);
    cudaMalloc((void **)&dBstencil, size);
    cudaMalloc((void **)&dC, size);

    //construct matrices
    for (int i=0; i<DSIZE*DSIZE; i++){
        A[i] = 1;ROR at malloc");
        B[i] = 3;
    }

    cudaMemcpy(dA, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dB, B, size, cudaMemcpyHostToDevice);

    cudaCheckErrors("ERROR at copy to device");
```

```
      // Launch stencil_2d() kernel on GPU
      int gridSize = (N + BLOCK_SIZE-1)/BLOCK_SIZE;
      dim3 grid(gridSize, gridSize);
      dim3 block(BLOCK_SIZE, BLOCK_SIZE);
      // Launch the kernel
      // Properly set memory address for first element on which the stencil will be
applied

    stencil_2d<<<grid,block>>>(dA, dAstencil, DSIZE);
    stencil_2d<<<grid,block>>>(dB, dBstencil, DSIZE);

    matrix_mul_gpu<<<grid, block>>>(dAstencil, dBstencil, dC, DSIZE);

    cudaCheckErrors("ERROR at kernel");

    // Copy result back to host
    cudaMemcpy(Astencil, dAstencil, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(Bstencil, dBstencil, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(C, dC, size, cudaMemcpyDeviceToHost);

    cudaCheckErrors("ERROR at copy back");

/*
In the actual code, at this point I reproduced the same error-checking code as in
the CPU version. However, for brevity's sake I have omitted those lines here.
*/

// Cleanup
    free(A);
    free(B);
    free(Astencil);
    free(Bstencil);
    free(C);
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dAstencil);
    cudaFree(dBstencil);
    cudaFree(dC);
      printf("Success!\n");

    // CPU timing
    t3 = clock();
    t3sum = ((double)(t3-t0))/CLOCKS_PER_SEC;
    printf("Compute took %f seconds\n", t3sum);

    return 0;
}
```

As the two homeworks prior to the project included kernels for matrix multiplication and a 2D stencil, I essentially reproduced my code from those assignments for this project. This also means my stencil code started by using shared memory.

Profiling with nsys

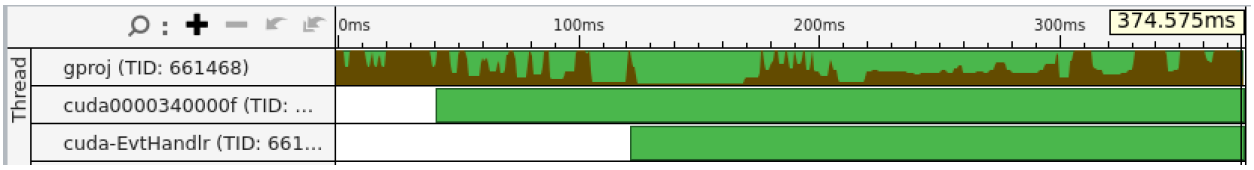
Using the "nsys stats" command after a default "nsys profile" run, I generated the following tables on device usage. As above some columns (mainly the statistical columns, which are a bit dubious anyway given how small the numbers are here) are omitted for simplicity's sake but the entire csv files (and text versions of the same) are attached.

Time (%)	Name	Total Time (ns)	Num Calls	Avg Time per Call (ns)
92.5	cudaMalloc	276439258	5	55287851.6
5.6	cudaMemcpy	16743910	5	3348782.0
1.6	cudaLaunchKernel	4794488	3	1598162.7
0.3	cudaFree	769888	5	153977.6
0.0	cuModuleGetLoadingMode	1770	1	1770.0

It's clear that the biggest hotspot here is memory allocation. However, this only lists the Device CUDA API calls, and ignores the CPU, where I used normal c++ malloc(). The stats output also gives a list of CPU calls, the top few lines of which I've reproduced here:

Time (%)	Name	Total Time (ns)	Num Calls	Avg Time per Call (ns)
78.6	poll	1122669482	41	27382182.5
20.1	ioctl	287565725	663	433734.1
0.6	mmap64	8725987	29	300896.1
0.2	fopen	3498457	45	77743.5

The third function on this list, "mmap64" is related to memory allocation and could be a proxy for tracking its time - if we go by that metric GPU memory allocation is much slower than CPU. It's difficult to say how much time was spent on the GPU vs CPU. To find out more about CPU performance, I profiled the CUDA application in VTune, and got that the total CPU usage time was 0.193 seconds. However, total application runtime also varied wildly, usually sticking to around 0.33 seconds, so I'm not convinced this is a good number either.



VTune also generated this plot, which shows the CPU working for either 0.04 or 0.12 seconds before starting CUDA tasks.
Looking more at the nsys output, we can examine how time was spent by the non-cudaMalloc functions. Let's start with cudaMemcpy:
Nothing too surprising, and we can move to kernels:

Time (%)	Operation	Total Time (ns)	Count	Avg (ns)
59.4	[CUDA memcpy Device-to-Host]	239420	3	79806.7
40.6	[CUDA memcpy Host-to-Device]	163838	2	81919.0

As in the CPU case, the matrix multiplication took the vast majority of the time. This is at least

Time (%)	Name	Total Time (ns)	Instances	Avg (ns)
98.3	matrix_mul_gpu(const int *, const int *, int *, int)	12037132	1	12037132.0
1.7	stencil_2d(int *, int *, int)	209533	2	104766.5

in part due to my matrix multiplication code not taking advantage of all the functionality CUDA has to offer. However, the time used to multiply matrices is still an order of magnitude less than the time for cudaMalloc, so the latter is my first focus when optimizing.

Switching to Managed Memory

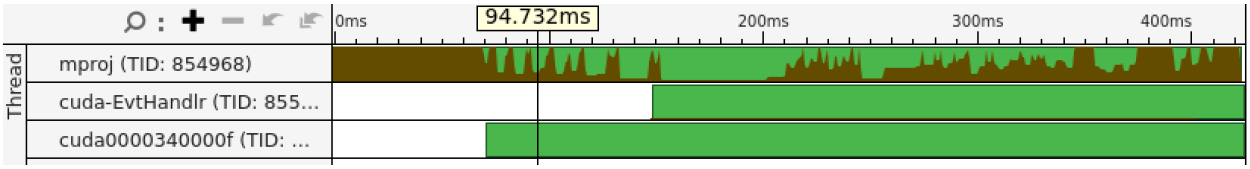
As the kernels here are essentially the same, I'm going to leave out function definitions in the copy of the code below, as well as the check functions, which are the same in every version of my code. No omissions are made in the attached code files.

```
#include <stdlib.h>
#include <iostream>
#include <algorithm>
#include <time.h>

#define N 512
#define DSIZE 512
#define RADIUS 2
#define BLOCK_SIZE 32

// error checking macro
#define cudaCheckErrors(msg)                                    \
(the same as in the above code, omitted for brevity             \

__global__ void stencil_2d(int *in, int *out, int size) {
    // same as above
 }

    // Square matrix multiplication on GPU : C = A * B
__global__ void matrix_mul_gpu(const int *A, const int *B, int *C, int size) {
   // same as above
 }

int main(void) {

    double t0, t3, t3sum;
    t0 = clock();

    int *dA, *dB, *dAstencil, *dBstencil, *dC;
    int size = (DSIZE)*(DSIZE) * sizeof(int);

    cudaMallocManaged((void **)&dA, size);
    cudaMallocManaged((void **)&dB, size);
    cudaMallocManaged((void **)&dAstencil, size);
    cudaMallocManaged((void **)&dBstencil, size);
    cudaMallocManaged((void **)&dC, size);

    cudaCheckErrors("ERROR at mallocmanaged");
```

```
    //construct matrices
    for (int i=0; i<DSIZE*DSIZE; i++){
        dA[i] = 1;
        dB[i] = 3;
    }

    cudaDeviceSynchronize();

    cudaCheckErrors("ERROR at synchronize");

      // Launch stencil_2d() kernel on GPU
      int gridSize = (N + BLOCK_SIZE-1)/BLOCK_SIZE;
      dim3 grid(gridSize, gridSize);
      dim3 block(BLOCK_SIZE, BLOCK_SIZE);
      // Launch the kernel

    stencil_2d<<<grid,block,(BLOCK_SIZE + 2 * RADIUS)*(BLOCK_SIZE + 2 *
RADIUS)*sizeof(int)>>>(dA, dAstencil, DSIZE);
    stencil_2d<<<grid,block,(BLOCK_SIZE + 2 * RADIUS)*(BLOCK_SIZE + 2 *
RADIUS)*sizeof(int)>>>(dB, dBstencil, DSIZE);

    matrix_mul_gpu<<<grid, block>>>(dAstencil, dBstencil, dC, DSIZE);

    cudaCheckErrors("ERROR at kernel");

    cudaDeviceSynchronize();

    cudaCheckErrors("ERROR at synchronize (post kernel execution)");

    // Error checking, as above

// Cleanup
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dAstencil);
    cudaFree(dBstencil);
    cudaFree(dC);
      printf("Success!\n ");
    // CPU timing
    t3 = clock();
    t3sum = ((double)(t3-t0))/CLOCKS_PER_SEC;
    printf("Compute took %f seconds\n", t3sum);

      return 0;
}
```

Profiling the managed memory code

Most of the profiling here was the same as before. I'll start with the CPU vs GPU time stuff -



the VTune run had CPU time at 0.241 seconds - more than in the non-managed case. I'm not

sure if this is a real difference, and it could be just due to the timing variance my code experiences. This version varied as much as the original CUDA code, but seemed to average slightly faster, around 0.32 seconds in total.
Consulting the same graph from before in VTune shows somewhere between 0.07 and 0.15 seconds before CUDA execution starts, but I'm unsure if this actually means anything.

Time (%)	Name	Total Time (ns)	Num Calls	Average (ns)
94.3	cudaMallocManaged	290521786	5	58104357.2
4.9	cudaDeviceSynchronize	15090332	2	7545166.0
0.6	cudaLaunchKernel	1947284	3	649094.7
0.2	cudaFree	679375	5	135875.0
0.0	cuModuleGetLoadingMode	2140	1	2140.0

Looking at the nsys stats output, we can see that cudaMallocManaged is taking more time than normal cudaMalloc was previously. However, this is understandable as the function (when combined with cudaDeviceSynchronize) performs the same functions as malloc, cudaMalloc, and cudaMemcpy.

We can examine the difference further by adding these operations. In the original code, cudaMalloc and cudaMemcpy together took 98.1% of device time, for a total of ~293 ms, while in the new code cudaMallocManaged and cudaDeviceSynchronize took 99.2% of time and ~306 ms. It's worth noting this doesn't account for regular c++ malloc(), as well as that cudaFree seems to be faster in the optimized case, however these two factors I assume are very minor.

Time (%)	Operation	Total Time (ns)	Count	Avg (ns)
76.2	[CUDA memcpy Unified Host-to-Device]	257556	42	6132.3
23.8	[CUDA memcpy Unified Device-to-Host]	80533	19	4238.6

Memory copying time seems to be around the same as without cudaMallocManaged, but the unified memory system does many more operations than explicit copying. Additionally, the device-to-host transfer seems to be 3 times faster than vice versa (as well as 3 times faster than the cudaMemcpy previously), and I'm not sure why this is, perhaps on this run I just got lucky and the device-to-host had fewer page faults.

Time (%)	Name	Total Time (ns)	Instances	Avg (ns)
86.0	matrix_mul_gpu(const int *, const int *, int *, int)	12994142	1	12994142.0
14.0	stencil_2d(int *, int *, int)	2109056	2	1054528.0

Matrix multiplication also seems to be an order of magnitude faster this time, which is odd as I don't expect any of the changes I made to have any effect on that kernel's operation. I'm chalking this up as a fluke as well.

Part 3: More Optimization

For this file, I kept managed memory and split the two stencil calls into their own streams. Additionally, I replaced my original matrix filling operation with a new CUDA kernel that could run in the two streams as well. The matrix multiplication was kept to the default stream so it could be done after the stenciling. I also attached the memory to each stream with cudaStreamAttachMemAsync, which I believe sped things up a bit. I also had to pay attention to what streams could use what memory, leading me to use the cudaMemAttachGlobal flag on the intermediate stencil matrices so the default stream could use them.

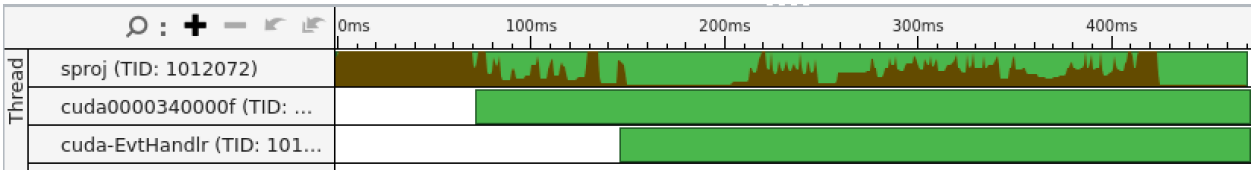
As before, I'll omit my matrix and stencil code, as well as the CUDA error check macro and my own error checking.

```
#include <stdlib.h>
#include <iostream>
#include <algorithm>
#include <time.h>
#define N 512
#define DSIZE 512
#define RADIUS 2
#define BLOCK_SIZE 32

// error checking macro
#define cudaCheckErrors(msg)                                    \
(the same as in the above code, omitted for brevity             \

__global__ void stencil_2d(int *in, int *out, int size) {
    // same as above
 }
    // Square matrix multiplication on GPU : C = A * B
__global__ void matrix_mul_gpu(const int *A, const int *B, int *C, int size) {
   // same as above
 }
__global__ void fill_gpu(int *A, int val, int size){
        // fills given array A with constant int value "val"
        int idx = threadIdx.x + blockDim.x * blockIdx.x;
        int idy = threadIdx.y + blockDim.y * blockIdx.y;
        A[idy*size+idx] = val;
}

int main(void) {

    double t0, t3, t3sum;
    t0 = clock();
    int *dA, *dB, *dAstencil, *dBstencil, *dC;
    int size = (N)*(N) * sizeof(int);
    //create streams and malloc
    cudaStream_t streamA, streamB;
    cudaStreamCreate(&streamA);
    cudaStreamCreate(&streamB);
    cudaMallocManaged((void **)&dA, size);
    cudaStreamAttachMemAsync(streamA, dA, size);
    cudaMallocManaged((void **)&dB, size);
    cudaStreamAttachMemAsync(streamB, dB, size);
    cudaMallocManaged((void **)&dAstencil, size);
    cudaStreamAttachMemAsync(streamA, dA, size, cudaMemAttachGlobal);
    cudaMallocManaged((void **)&dBstencil, size);
    cudaStreamAttachMemAsync(streamB, dB, size, cudaMemAttachGlobal);
    cudaMallocManaged((void **)&dC, size);

    cudaCheckErrors("ERROR at malloc");
```

```
      // Launch stencil_2d() kernel on GPU
      int gridSize = (N + BLOCK_SIZE-1)/BLOCK_SIZE;
      dim3 grid(gridSize, gridSize);
      dim3 block(BLOCK_SIZE, BLOCK_SIZE);
      // Launch the kernel

    //fill values
    fill_gpu<<<grid, block, 0, streamA>>>(dA, 1, DSIZE);
    fill_gpu<<<grid, block, 0, streamB>>>(dB, 3, DSIZE);

    stencil_2d<<<grid,block,0,streamA>>>(dA, dAstencil, DSIZE);
    stencil_2d<<<grid,block,0,streamB>>>(dB, dBstencil, DSIZE);

    cudaStreamSynchronize(streamA);
    cudaStreamSynchronize(streamB);

    matrix_mul_gpu<<<grid, block>>>(dAstencil, dBstencil, dC, DSIZE);

    cudaCheckErrors("ERROR at kernel”);

    // Copy result back to host

    cudaDeviceSynchronize();
    cudaCheckErrors("ERROR at copy back");

    // Error checking, as above

// Cleanup

    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dAstencil);
    cudaFree(dBstencil);
    cudaFree(dC);

    cudaStreamDestroy(streamA);
    cudaStreamDestroy(streamB);

    printf("Success!\n");

    // CPU timing
    t3 = clock();
    t3sum = ((double)(t3-t0))/CLOCKS_PER_SEC;
    printf("Compute took %f seconds\n", t3sum);

    return 0;
}
```

Profiling the optimized code

Once again I’ll start with guessing at what the CPU is doing via VTune.



From the graph it seems the GPU started between 0.06 and 0.15 seconds after the program started, maybe. We get a “CPU time” of 0.221 seconds, which is between the two other CUDA programs. However, of the three this version was the most variable in timing, though it seemed to usually take around 0.31 seconds.

Time (%)	Name	Total Time (ns)	Num Calls	Avgerage (ns)
94.1	cudaStreamCreate	285797412	2	142898706.0
4.2	cudaDeviceSynchronize	12699551	1	12699551.0
0.9	cudaLaunchKernel	2605830	5	521166.0
0.5	cudaStreamSynchronize	1585047	2	792523.5
0.2	cudaFree	725726	5	145145.2
0.1	cudaMallocManaged	287257	5	57451.4
0.0	cudaStreamAttachMemAsync	47581	4	11895.3
0.0	cudaStreamDestroy	25150	2	12575.0
0.0	cuModuleGetLoadingMode	1420	1	1420.0

It now seems that the most time is being taken creating the streams, although I suspect this is also counting memory allocation time. Taking cudaStreamCreate, cudaStreamSynchronize, cudaDeviceSyncronize, and cudaMallocManaged together, we get 98.7% of the time and 300ms, in between the other two programs but once again I don’t trust the variance.

Time (%)	Name	Total Time (ns)	Instances	Avg (ns)
86.4	matrix_mul_gpu(const int *, const int *, int *, int)	12677285	1	12677285.0
7.0	fill_gpu(int *, int, int)	1034161	2	517080.5
6.5	stencil_2d(int *, int *, int)	960818	2	480409.0

We also see the matrix multiplication taking the same amount of time, and percentage of usage, as in the non-stream case. This makes sense, as it is on the default stream in both cases. However, the stencil time seems to have been cut in half, with the remainder filled by the matrix filling kernel. I think the speedup on the stencil is due to the streams, as well as the data already being on the device from being filled in - it may have even been in the same SMs.

Due to filling in the memory on the device, there is only one memory copy operation, so I’m omitting that table in this write-up.

Part 5: Alpaka

Due in part to Thanksgiving break, and in part to the poor online documentation of the library, as well as my unfamiliarity with some of the more advanced c++ concepts, such as “auto” and

templates, which it relies on, I was unable to get Alpaka working for this project in time to turn it in. I will happily accept many points off the final not doing this part entails.

Setting up the code to run, and environment details

As I skipped the Alpaka section, my work should be easy to reproduce. The c++ code I compiled with “g++ project.cpp -o cproj”. In order, the three CUDA programs (initial, managed memory, and optimization/adding streams) were compiled as follows:

-“nvcc project.cu -o gproj”
-“nvcc managed.cu -o mproj”
-“nvcc streams.cu -o sproj”

The names in the above commands also correspond to the nsys reports attached. The VTune hotspot reports for the CPU code is simply called “hotspot.txt”/“hotspot.csv”

Github access

Code files, nsys/VTune outputs, and a copy of this document can be found at https://github.com/michael7198/tachep2024finalproject

Final Notes

Thanks for a great class! I really enjoyed finally learning c++, at least to some extent, and while my current research may not involve GPUs too much, I can already think of several future projects that would!