

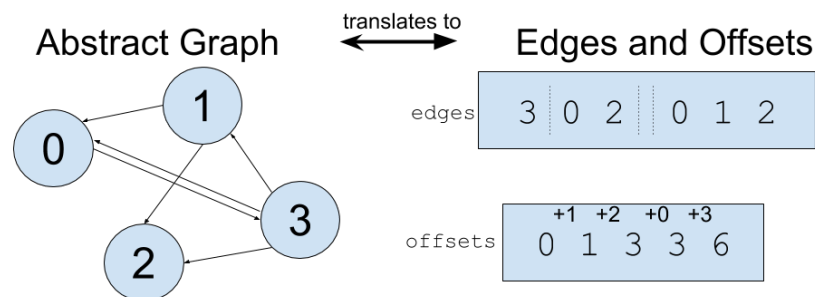
CS205 Homework 2, Group Problem

Andrew Ross, Shawn Pan, and Nathaniel Burbank

Breadth-First Search (BFS)

To implement BFS as a *sparse* matrix vector computation, we initially represented the graph as an array of linked lists, where each array element corresponds to a node, and each linked list corresponds to its edges. However, this approach ended up being incompatible with OpenAcc, which only copied the array over to the GPU (and left the linked lists on the host device).

Instead, we devised a representation of the graph as two 1D arrays of edge endpoints and edge counts (actually offsets) for each node, which schematically looks like:



The offsets array has length $N + 1$, where N is the number of nodes, and is structured such that the i th and $i + 1$ th entries give the range of indices in the edges array that contain the edges for node i . If we needed to also store weights, we could include a second array of the same size as edges that contained them. The edge array's length is given by the last element of offsets.

This representation allowed us to implement sparse matrix-vector multiplication using only two relatively small 1D arrays, both of which are accessed sequentially. This already made our sequential implementation quite fast:

```
int edgelist_BFS(int* edges, int* offsets, int start, int goal, int N) {
    int distance = 0;
    int reached_goal = 0;
    int* x_old = zeros(N); x_old[start] = 1;
    int* x_new = zeros(N); x_new[start] = 1;

    while (!reached_goal && distance < N) {
        edgelist_matrix_vector_multiply(edges, offsets, x_old, x_new, N);
        reached_goal = x_new[goal];
        swap(x_old, x_new);
        distance += 1;
    }

    free(x_old);
    free(x_new);
}
```

```

    return reached_goal ? distance : -1;
}

void edgelist_matrix_vector_multiply(int* edges, int* offsets, int* vector, int* result)
{
    for (int i = 0; i < N; i++)
        if (vector[i] == 1)
            for (int j = offsets[i]; j < offsets[i+1]; j++)
                result[edges[j]] = 1;
}

```

We implemented the parallelized version with the OpenAcc parallel loop pragma.

```

void edgelist_matrix_vector_multiply_parallel(const int* edges, const int* offsets, int* vector, int* result)
// loop over elements in our vector
#pragma acc parallel loop copyin(vector[:N], offsets[:N+1], edges[:offsets[N]]) copyout(result[:offsets[N]])
for (int i = 0; i < N; i++) {
    if (vector[i] == 1) {
        for (int j = offsets[i]; j < offsets[i+1]; j++) {
            result[edges[j]] = 1;
        }
    }
}
}

```

We also attempted to make both loops parallel, but the performance was approximately the same as parallelizing just the outer loop.

```

#pragma acc data copyin(x_old[:N], offsets[:N+1], edges[:offsets[N]]) copyout(x_new[:N])
{
    while (!reached_goal && distance < N) {
        #pragma acc parallel loop gang
        for (int i = 0; i < N; i++) {
            #pragma acc loop independent vector
            for (int j = offsets[i]; j < offsets[i+1]; j++) {
                x_new[edges[j]] |= x_old[i];
            }
        }
        reached_goal = x_new[goal];
        swap(x_old, x_new);
        distance += 1;
    }
}

```

We suspect our parallel version was not faster than the serial version for several reasons. All the graph sizes took about 3 seconds on the GPU accelerated code, which suggests most of that is overhead in starting up the GPU. Bigger problem sizes may be needed to observe speedup. The inner loop has a variable number of iterations equal to the number of edges flowing out of each node. Therefore, work may not have been distributed equally among the parallel threads. The problem is not very cache friendly in memory access when writing the result of the matrix-vector multiplication.

All-Pairs Shortest Path (APSP)

First, we implemented the sequential version of the Floyd-Warshall algorithm:

```
void floyd_apsp(float* D, int n) {
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                float detour = D[i*n + k] + D[k*n + j];
                if (detour < D[i*n + j])
                    D[i*n + j] = detour;
            }
        }
    }
}
```

However, we encountered difficulties trying to parallelize this due to the fact that the outer loop over k must be executed sequentially. Inserting `#pragmas` into the inner loops is possible, but leads to slower performance due to the repeated spin-up and spin-down of threads and unnecessary data movement. We tried to specify that data should be copied once at the beginning of the block and reused in the parallel inner loops, but this did not appear to improve performance.

We did succeed in parallelizing the algorithms based on the tropical semiring, however, mostly because they spend the bulk of their time in matrix multiplication operations that we have already learned to code in parallel. We implemented two distinct versions of the parallel semiring-based APSP with separate indexing schemes. In the first, we stored arrays in row-major order, with partitioning and recombining steps at each recursive iteration:

```
void tropical_apsp(float* A, int n) {
    if (n > 1) {
        int l = n/2;
        size_t size = l*l*sizeof(float);
        float* temp = (float*)malloc(size);

        // manually partition A into quadrants
        float* A11 = (float*)malloc(size);
        float* A12 = (float*)malloc(size);
        float* A21 = (float*)malloc(size);
        float* A22 = (float*)malloc(size);
        float* quadrants[4] = { A11, A12, A21, A22 };
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                quadrants[2*(i>=l)+(j>=l)][(i%l)*l+(j%l)] = A[i*n + j];

        tropical_apsp(A11, l);
        tropical_gemm(A11, A12, temp, l); memcpy(A12, temp, size);
        tropical_gemm(A21, A11, temp, l); memcpy(A21, temp, size);
        tropical_gemm(A21, A12, temp, l); minplace(A22, temp, l);

        tropical_apsp(A22, l);
    }
}
```

```

    tropical_gemm(A22, A21, temp, 1); memcpy(A21, temp, size);
    tropical_gemm(A12, A22, temp, 1); memcpy(A12, temp, size);
    tropical_gemm(A12, A21, temp, 1); minplace(A11, temp, 1);

    for (int i = 0; i < 1; i++) {
        for (int j = 0; j < 1; j++) {
            A[(i)*n+(j)] = A11[i*1+j];
            A[(i)*n+(j+1)] = A12[i*1+j];
            A[(i+1)*n+(j)] = A21[i*1+j];
            A[(i+1)*n+(j+1)] = A22[i*1+j];
        }
    }

    free(A11);
    free(A12);
    free(A21);
    free(A22);
    free(temp);
}

}

void tropical_gemm(float* A, float* B, float* C, int n) {
    #pragma acc parallel loop copyin(A[:n*n], B[:n*n]) copyout(C[:n*n])
    for (int i = 0; i < n; i++) {
        #pragma acc loop independent
        for (int j = 0; j < n; j++) {
            float c = inf;
            for (int k = 0; k < n; k++)
                c = fmin(c, A[i*n+k] + B[k*n+j]);
            C[i*n+j] = c;
        }
    }
}

```

To reduce the number of copy operations required, we realized that for square matrices whose dimensions n are powers of 4, it is possible to index them in a 1D array not in row-major order but recursively such that quadrants map to quarters. Just to enumerate this briefly, for an 8×8 matrix, the 1D indexes would be as follows:

$$\begin{bmatrix} 0 & 1 & 4 & 5 \\ 2 & 3 & 6 & 7 \\ 8 & 9 & 12 & 13 \\ 10 & 11 & 14 & 15 \end{bmatrix}$$

This allows us to implement a slightly simpler APSP routine:

```

void tropical_quadtree_apsp(float* A, unsigned int n) {
    if (n > 1) {
        int l = n/2;
        size_t size = l*l*sizeof(float);
        float* temp = (float*)malloc(size);
    }
}

```

```

float* A11 = A;
float* A12 = A + n*n/4;
float* A21 = A + n*n/2;
float* A22 = A + 3*n*n/4;

tropical_quadtrees_apsp(A11, 1);
tropical_quadtrees_gemm(A11, A12, temp, 1); memcpy(A12, temp, size);
tropical_quadtrees_gemm(A21, A11, temp, 1); memcpy(A21, temp, size);
tropical_quadtrees_gemm(A21, A12, temp, 1); minplace(A22, temp, 1);

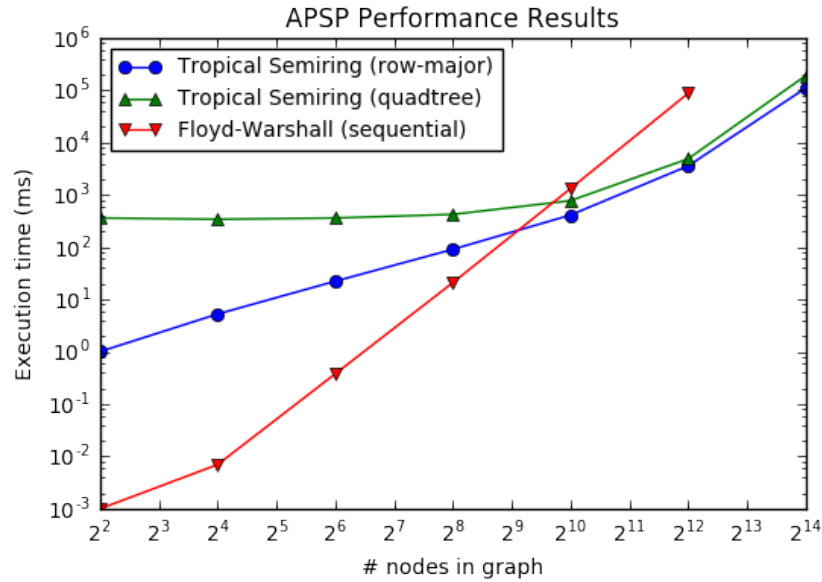
tropical_quadtrees_apsp(A22, 1);
tropical_quadtrees_gemm(A22, A21, temp, 1); memcpy(A21, temp, size);
tropical_quadtrees_gemm(A12, A22, temp, 1); memcpy(A12, temp, size);
tropical_quadtrees_gemm(A12, A21, temp, 1); minplace(A11, temp, 1);

free(temp);
}
}

void tropical_quadtrees_gemm(float* A, float* B, float* C, unsigned int n) {
    #pragma acc parallel loop copyin(A[:n*n], B[:n*n]) copyout(C[:n*n])
    for (unsigned int i = 0; i < n; i++) {
        #pragma acc loop independent
        for (unsigned int j = 0; j < n; j++) {
            float c = inf;
            for (unsigned int k = 0; k < n; k++)
                c = fmin(c, A[qx(i,k)] + B[qx(k,j)]);
            C[qx(i,j)] = c;
        }
    }
}

```

where the `qx` function maps 2D indexes to 1D quadtree indexes, which can be done in practice with **interleaving bits**. However, in practice, this performed slightly slower than our normal tropical semiring APSP implementation, perhaps because for the actual matrix multiplications, we sacrifice the sequential access patterns that make parallelization memory-efficient (since logically adjacent matrix elements are no longer physically adjacent). Another possible reason is the additional computational cost required to translate 1D indexes into quadtree form. Performance results for both versions of the tropical algorithm as compared to the sequential Floyd-Warshall implementation are shown below, benchmarked against RMAT graphs of varying scale factor (edge factor did not appear to have a significant effect on performance):



Note that the sequential Floyd-Warshall algorithm is initially faster because there is no need to copy any data to the GPU, but for large problem sizes, the parallel implementations are over an order of magnitude faster.

We tested our algorithms to ensure they agreed with each other and with the example from the lecture notes. [Unit tests](#) and the [APSP code](#) are available online.