Improvement Project : Parallel Gene Sequence Alignment

April 8, 2009 / CS 312 Algorithms
Duane Johnson

Objective

The Needleman-Wunsch algorithm as implemented in Project 5 is a serial solution to the gene sequence alignment problem. This paper reports on a parallel solution using Nvidia's CUDA extensions to C on a general-purpose graphics processing unit (GPGPU).

Solution

Instead of solving the problem in left-to-right and top-to-bottom fashion, this parallel algorithm works from the upper-left corner to the lower-right corner of the scoring table. It also partitions the problem into smaller units that are sized appropriately for the 240-core 8-series graphics processor from Nvidia. Once the scoring is achieved in parallel, the actual alignment backtrace proceeds as it did before--that is, in serial.

Details of the Parallel Algorithm

The 8-series graphics hardware consists of 30 blocks of processing cores, each block containing 8 cores (i.e. a total of 240 cores). In order to take advantage of these cores, it seemed reasonable to divide the scoring table up into sections of 16 x 16 and map these sections to blocks on the hardware (see Fig. 1).

The algorithm starts in the upper-left hand corner of the scoring table and assigns the first 16 x 16 section of the table to a block. The block's 8 cores then proceed in a diagonal upper-left-to-lower-right calculation step, adding up and scoring the various pathways as they go. Since each cell depends only on its three neighbors (one to the left, one above, and one diagonally to the left and above) the 8 cores are free to operate at near-optimal speed--"near optimal", that is, because they cannot all begin work at first, since each row must wait until the first cell on the row above it has been scored.

Once the first 16 x 16 section has been calculated, the section to its right and the section below it can begin to be calculated in parallel. Thus, the second iteration of 16 x 16 sections (i.e. a diagonal "row") can potentially be achieved as fast as the first iteration. This block-wise diagonal marching continues until the entire table is scored and an optimal score is entered in the lower-right hand cell of the table.

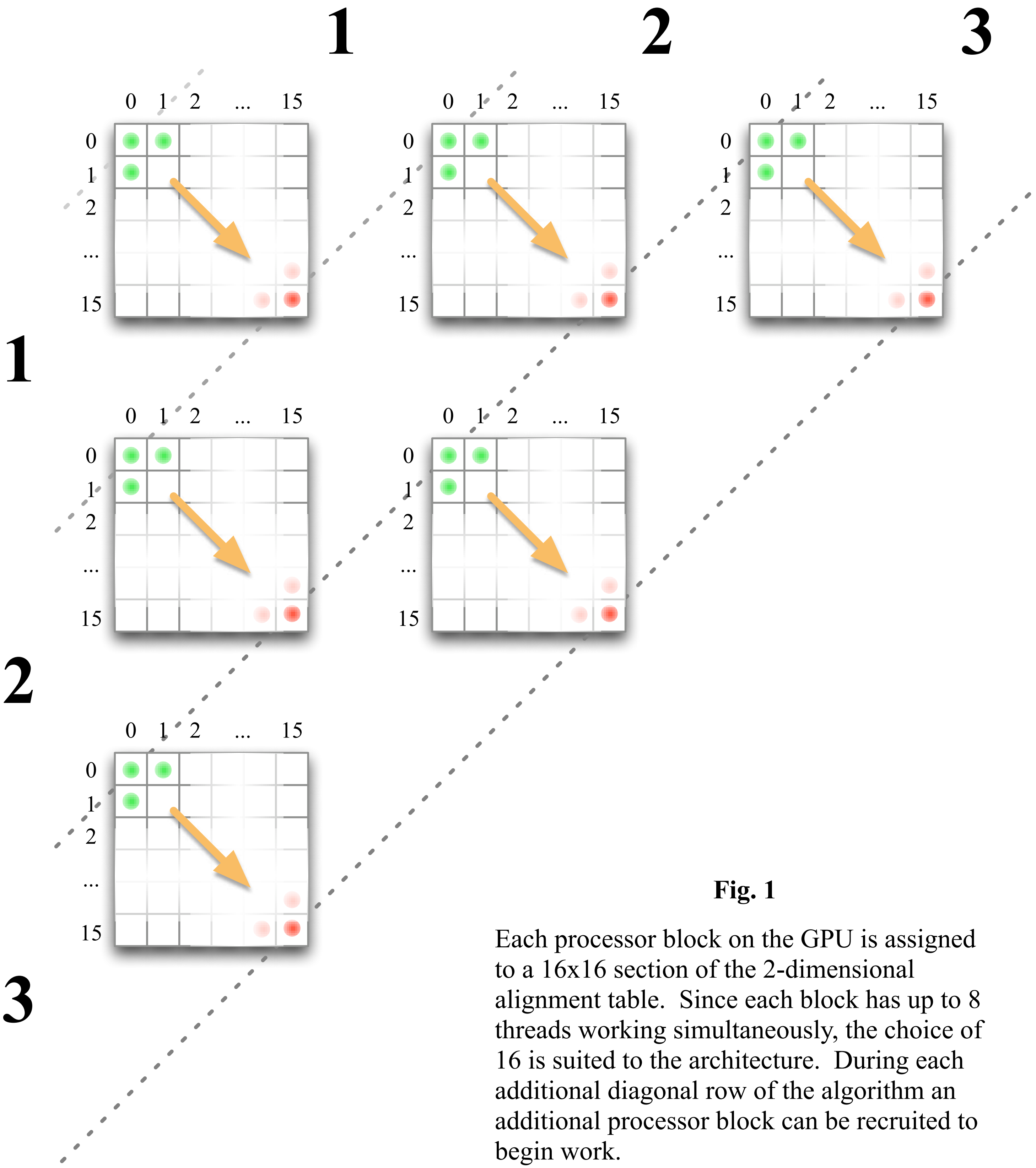

Fig. 1

Each processor block on the GPU is assigned to a 16x16 section of the 2-dimensional alignment table. Since each block has up to 8 threads working simultaneously, the choice of 16 is suited to the architecture. During each additional diagonal row of the algorithm an additional processor block can be recruited to begin work.

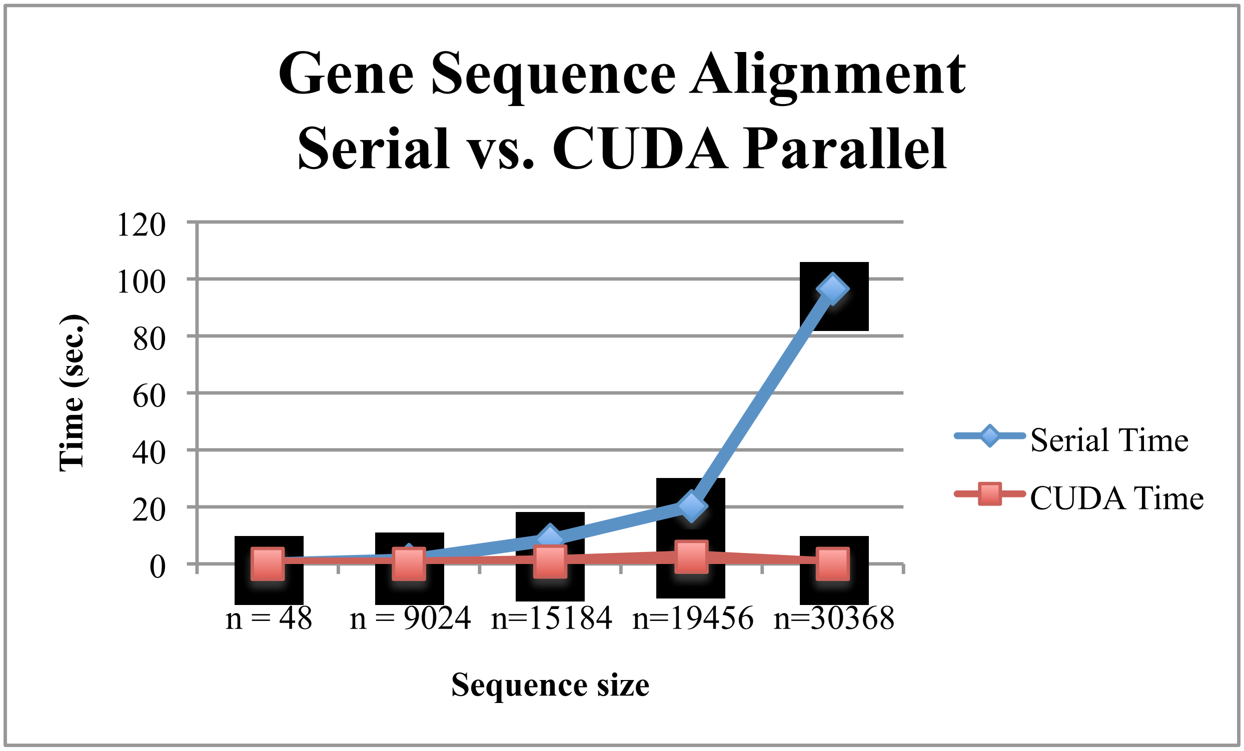

Fig. 2 Parallel alignment yields a speedup of about 10 X

	n = 48	**n = 9024**	**n=15184**	**n=19456**	**n = 30368**
Serial Time	0.000018	1.392797	8.435221	20.276045	96.541793
CUDA Time	0.000043	0.005645	0.813134	2.487627	failed
Space	2 kB	81 MB	231 MB	379 MB	922 MB

Table 1 Measurements of scoring times from n = 48 to n = 30368

Analysis

Ignoring setup time for the GPU, the absolute speedup is nearly 10 X for values of n in the range $15,000 < n < 20,000$. Part of this speedup is due to the fact that the serial implementation is not optimized for serial processing; nevertheless, the performance improvement is significant for problems whose scoring table can be fit in the GPU's memory. As can be seen in the case of n = 30368, the CUDA score failed since the table could not be transferred to GPU memory.

Since many alignment problems are in the range of thousand or tens of thousands of base pairs, parallel-accelerated speedups using GPGPUs is a practical way to reduce alignment times for sizes in this range by an order of ten.

```
#include <stdio.h>
#include <sys/time.h>
#include <string.h>
#include <assert.h>
#include <cuda.h>

#include "misc.h"

#define BLOCK_SIZE 16
// Costs
#define INDEL 3
#define SUBST 1
#define MATCH 0


typedef struct GRID
{
    int w; // Width of grid
    int h; // Height of grid
    int* box;
    int success;
} Grid;

Grid* grid_new();
Grid* grid_init( Grid* g, int w, int h );
Grid* grid_init_file( Grid* g, char* filename );
Grid* grid_free( Grid* g );
Grid* grid_clear( Grid* g );
Grid* grid_copy( Grid* a, Grid* b );
Grid* grid_copy_to_device( Grid* g );
Grid* grid_copy_from_device( Grid* g );
Grid* grid_set_seq_row( Grid* g, char* seq, int w );
Grid* grid_set_seq_col( Grid* g, char* seq, int h );
Grid* grid_show( Grid* g );
Grid* grid_save( FILE* f, Grid* g );
Grid* grid_alignment_serial( Grid* g );
Grid* grid_alignment_parallel( Grid* g );

Grid* grid_new()
{
    Grid* g = (Grid*)malloc( sizeof( Grid ) );

    g->w = 0;
    g->h = 0;
    g->box = NULL;
    g->success = true;

    return g;
}

Grid* grid_init( Grid* g, int w, int h )
{
    assert( g != NULL );
    assert( g->success == true );

    // Free memory if necessary
    grid_free( g );

    g->w = w + 2;
    g->h = h + 2;
    g->box = (int *)malloc( g->w * g->h * sizeof( int ) );
```

```c

    assert( g->box != NULL);

    return g;
}

Grid* grid_init_file( Grid* g, char* filename )
{
    assert( g != NULL );
    assert( g->success == true );
    assert( filename != NULL );
    assert( strlen(filename) > 0 );

    FILE* input = fopen( filename, "r" );
    if( input != NULL )
    {
        int i = -1;
        // int cur = 0;
        char* line;
        char* seq[2];
        int seq_size[2] = {0, 0};

        seq[0] = (char *)malloc( 1 );
        seq[1] = (char *)malloc( 1 );
        while( (line = readline( input )) )
        {
            int length = strlen( line );
            if( length == 0 )
            {
                // Skip blank lines
            }
            else if( line[0] == '>' )
            {
                // Begin sequence after this line
                i++;
            }
            else if( i >= 0 && i <= 1 )
            {
                // Grab contents and add it to our seq
                int j;
                for( j = 0; j < length; j++ )
                {
                    if( line[j] != ' ' && line[j] != '\t' && line[j] != '\n' )
                    {
                        seq[i][ seq_size[i]++ ] = line[j];
                        seq[i] = (char*)realloc( seq[i], seq_size[i] + 1 );
                    }
                }
            }
            free( line );
        }
        // Be nice and add a trailing null char so sequences can be printed
        seq[0][ seq_size[0] ] = '\0';
        seq[1][ seq_size[1] ] = '\0';

        // Grid sequences point to newly loaded sequences
        grid_init( g, seq_size[0], seq_size[1] );
        grid_set_seq_row( g, seq[0], seq_size[0] );
        grid_set_seq_col( g, seq[1], seq_size[1] );

        fclose( input );
```

```
    }
    else
    {
        g->success = false;
    }

    return g;
}

Grid* grid_free( Grid* g )
{
    assert( g->success == true );

    if( g->box != NULL )
    {
        free( g->box );
        g->box = NULL;
    }

    return g;
}

Grid* grid_clear( Grid* g )
{
    assert( g->success == true );
    assert( g->box != NULL );

    int size = g->w * g->h * sizeof(int);
    memset( g->box, 0, size );

    return g;
}

// Copy grid 'a' to grid 'b'
Grid* grid_copy( Grid* a, Grid* b )
{
    assert( a->success == true && b->success == true );
    assert( a->w == b->w && a->h == b->h );
    assert( b->box != NULL );

    int size = a->w * a->h * sizeof(int);
    memcpy( b->box, a->box, size );

    return a;
}

// Copies a Grid object to the device and returns a DEVICE pointer to the copy
Grid* grid_copy_to_device( Grid* g )
{
    assert( g->success == true );
    assert( g->box != NULL );

    // Create a temp Grid object where we will setup a Device pointer to the box data
    Grid tmp;
    tmp.w = g->w;
    tmp.h = g->h;
    tmp.success = true;

    // Allocate room for the object AND the object's box data
    Grid* grid_d;
    int size = sizeof( int ) * tmp.w * tmp.h;
```

```
    cudaMalloc( (void**)& grid_d, sizeof( Grid ) );
    cudaMalloc( (void**)& tmp.box, size );

    // Copy the object and the box data to the device
    cudaMemcpy( grid_d, &tmp, sizeof( Grid ), cudaMemcpyHostToDevice);
    cudaMemcpy( tmp.box, g->box, size, cudaMemcpyHostToDevice);

    // Return the DEVICE pointer
    return grid_d;
}

Grid* grid_copy_from_device( Grid* g )
{
    // Copy the object from the device
    Grid* grid_h = (Grid*)malloc( sizeof( Grid ) );
    cudaMemcpy( grid_h, g, sizeof( Grid ), cudaMemcpyDeviceToHost);

    // assert( grid_h->success == true );

    // Copy the box data from the device
    int size = sizeof( int ) * grid_h->w * grid_h->h;
    int* box = (int*)malloc( size );
    cudaMemcpy( box, grid_h->box, size, cudaMemcpyDeviceToHost);
    grid_h->box = box;

    assert( grid_h->box != NULL );

    // Return the HOST pointer
    return grid_h;
}

Grid* grid_set_seq_row( Grid* g, char* seq, int w )
{
    assert( g != NULL );
    assert( g->box != NULL );
    assert( g->w >= w );

    for( int i = 2; i < g->w; i++ )
    {
        g->box[i] = seq[i-2];
    }

    return g;
}

Grid* grid_set_seq_col( Grid* g, char* seq, int h )
{
    assert( g->h >= h );

    int i = 2 * g->w;
    int j;
    for( j = 0; j < h; j++, i+=g->w )
        g->box[i] = seq[j];

    return g;
}

// Show small grids as text output.  NOTE: Will not work for values > 48
Grid* grid_show( Grid* g )
{
    return grid_save( stdout, g );
```

```
}

Grid* grid_save( FILE* f, Grid* g )
{
    fprintf( f, "Show Grid: %d x %d\n", g->w - 2, g->h - 2 );
    int i, j;
    for( i = 0; i < g->h; i++ )
    {
        for( j = 0; j < g->w; j++ )
        {
            int c = g->box[ i * g->w + j ];
            if( i == 0 || j == 0 )
            {
                if( c == 0) fprintf( f, "    " );
                else fprintf(f, "  %c ", c);
            }
            else fprintf(f, "%3d ", c);
        }
        fprintf( f, "\n");
    }
    return g;
}

__host__ __device__ int min3( int a, int b, int c )
{
    return (a < b ? (a < c ? a : (b < c ? b : c)) : (b < c ? b : c));
}

__host__ __device__ Grid* grid_align_setup( Grid* g )
{
    // Initialize corner
    g->box[1 * g->w + 1] = 0;

    // Prepare first horizontal line
    for( int i = 2; i < g->w; i++ )
        g->box[1 * g->w + i] = (i - 1) * INDEL;

    // Prepare first vertical line
    for( int i = 2; i < g->h; i++ )
        g->box[i * g->w + 1] = (i - 1) * INDEL;

    return g;
}

// Aligns a BLOCK_SIZE x BLOCK_SIZE segment of a grid.  'g' is a Grid in DEVICE memory.
__global__ void cuda_grid_align_block( Grid* g, int k_major )
{
    int t = threadIdx.x;
    int row = g->w;
    int x_init, y_init;
    int x_block = g->w / BLOCK_SIZE - 1;

    if( k_major <= x_block)
    {
        x_init = (k_major - blockIdx.x) * BLOCK_SIZE + 2;
        y_init = (blockIdx.x) * BLOCK_SIZE + 2;
    }
    else
    {
        x_init = (x_block - blockIdx.x) * BLOCK_SIZE + 2;
        y_init = (k_major - x_block + blockIdx.x) * BLOCK_SIZE + 2;
```

```
    }

    // Increasing Breadth
    for( int k = 0; k < BLOCK_SIZE * 2; k++ )
    {
        if( t <= k && k - t < BLOCK_SIZE)
        {
            int x = x_init + (k - t);
            int y = (y_init + t) * row;

            int diag = g->box[(y - row) + (x - 1)];
            int vert = g->box[(y - row) + (x)];
            int horz = g->box[(y) + (x - 1)];

            int c1 = diag + (g->box[x] == g->box[y] ? MATCH : SUBST);
            int c2 = vert + INDEL;
            int c3 = horz + INDEL;

            g->box[x + y] = min3(c1, c2, c3);
        }
        __syncthreads();
    }
}

// Single-processor Alignment
Grid* grid_alignment_serial( Grid* g )
{
    grid_align_setup( g );

    // Setup for diagonal alignment solution
    int width = g->w - 2;
    int col = 1, row = g->w;

    // Increase diagonally
    for( int k = 0; k < 2 * width; k++ )
    {
        int i_max = (k < width ? k : 2 * width - k - 2);
        for( int i = 0; i <= i_max; i++ )
        {
            int x, y;
            if( k < width )
            {   // Increasing breadth
                x = (2 + k - i);
                y = (2 + i) * row;
            }
            else
            {   // Decreasing breadth
                x = (1 + width - i);
                y = (3 + k - width + i) * row;
            }
            int diag = g->box[(y - row) + (x - col)];
            int vert = g->box[(y - row) + (x)];
            int horz = g->box[(y) + (x - col)];
            int c1 = diag + (g->box[x] == g->box[y] ? MATCH : SUBST);
            int c2 = vert + INDEL;
            int c3 = horz + INDEL;
            g->box[x + y] = min3(c1, c2, c3);
        }
    }

    return g;
```

```
}

// Aligns a grid.  'g' is a Grid in DEVICE memory.
Grid* grid_alignment_parallel( Grid* g, int width, int debug )
{
    int blocks = width / BLOCK_SIZE;

    int k = 0;
    for( int i = 1; i <= blocks; i++ )
    {
        if( debug ) printf("iteration %d (>)\n", k);
        cuda_grid_align_block<<< i, BLOCK_SIZE >>>( g, k++ );
    }
    for( int i = blocks - 1; i > 0; i--)
    {
        if( debug ) printf("iteration %d (<)\n", k);
        cuda_grid_align_block<<< i, BLOCK_SIZE >>>( g, k++ );
    }

    return g;
}

int main( int argc, char** argv )
{
    char* input = shell_arg_string( argc, argv, "-f", "default.fasta" );
    char* output = shell_arg_string( argc, argv, "-o", "" );
    int show_alignment = shell_arg_present( argc, argv, "--show" );
    int align_serial = shell_arg_present( argc, argv, "--serial" );
    int show_debug = shell_arg_present( argc, argv, "--debug" );
    int show_help = shell_arg_present( argc, argv, "-h" ) ||
                    shell_arg_present( argc, argv, "--help" );

    printf( "CUDA Needleman-Wunsch\n(c) 2009 Duane Johnson\n\n" );
    printf( "Input FASTA: %s\n", input );
    if( strlen( output ) > 0 )
        printf( "Output Alignment: %s\n", output );
    printf( "Show Alignment: %d\n", show_alignment );
    printf( "Align in %s\n", (align_serial ? "Serial" : "Parallel") );
    if( show_debug )
        printf( "Debug Output ON\n" );
    if( show_help )
    {
        printf( "\nOptions:\n" );
        printf( "  -f <default.fasta>      FASTA input file.\n");
        printf( "  -o <default.align>      Table output file.\n");
        printf( "  --show                  Show table output in standard output.\n");
        printf( "  --serial                Do the alignment in serial (rather than parallel).\n");
        printf( "  --debug                 Show some debug output.\n");
        printf( "  --help                  This help message.\n");
    }
    else
    {
        Grid* grid_h = grid_new();
        Grid* grid_d;
        Grid* grid_result;

        grid_init_file( grid_h, input );
        printf("Size of Grid: %d x %d\n", grid_h->w - 2, grid_h->h - 2);

        if( (grid_h->w - 2) % BLOCK_SIZE != 0 ||
```

```
            (grid_h->h - 2) % BLOCK_SIZE != 0)
        {
            printf( "Aborted. Sequence must be a multiple of %d.\n", BLOCK_SIZE );
        }
        else
        {
            grid_align_setup( grid_h );

            if( align_serial )
            {
                double s = when();
                grid_result = grid_alignment_serial( grid_h );
                double f = when();
                printf("Completed alignment in %f sec.\n", (f-s));
            }
            else
            {
                printf("Copying to device...\n");
                grid_d = grid_copy_to_device( grid_h );
                printf("Starting parallel alignment...\n");
                double s = when();
                grid_alignment_parallel( grid_d, grid_h->w, show_debug );
                double f = when();
                printf("Completed alignment in %f sec.\n", (f-s));
                grid_result = grid_copy_from_device( grid_d );
                printf("Result copied to main memory.\n");

            }

            if( show_alignment )
                grid_show( grid_result );

            if( strlen( output ) > 0 )
            {
                FILE* out = fopen( output, "w" );
                grid_save( out, grid_result );
                fclose( out );
            }
        }
    }
}
```