

Parallel Gene Sequence Alignment on CUDA

April 8, 2009 / CS 484

Duane Johnson

Objective

The Needleman-Wunsch algorithm can be implemented in parallel as well as serial. This paper reports on a parallel solution using Nvidia's CUDA extensions to C on a general-purpose graphics processing unit (GPGPU).

Solution

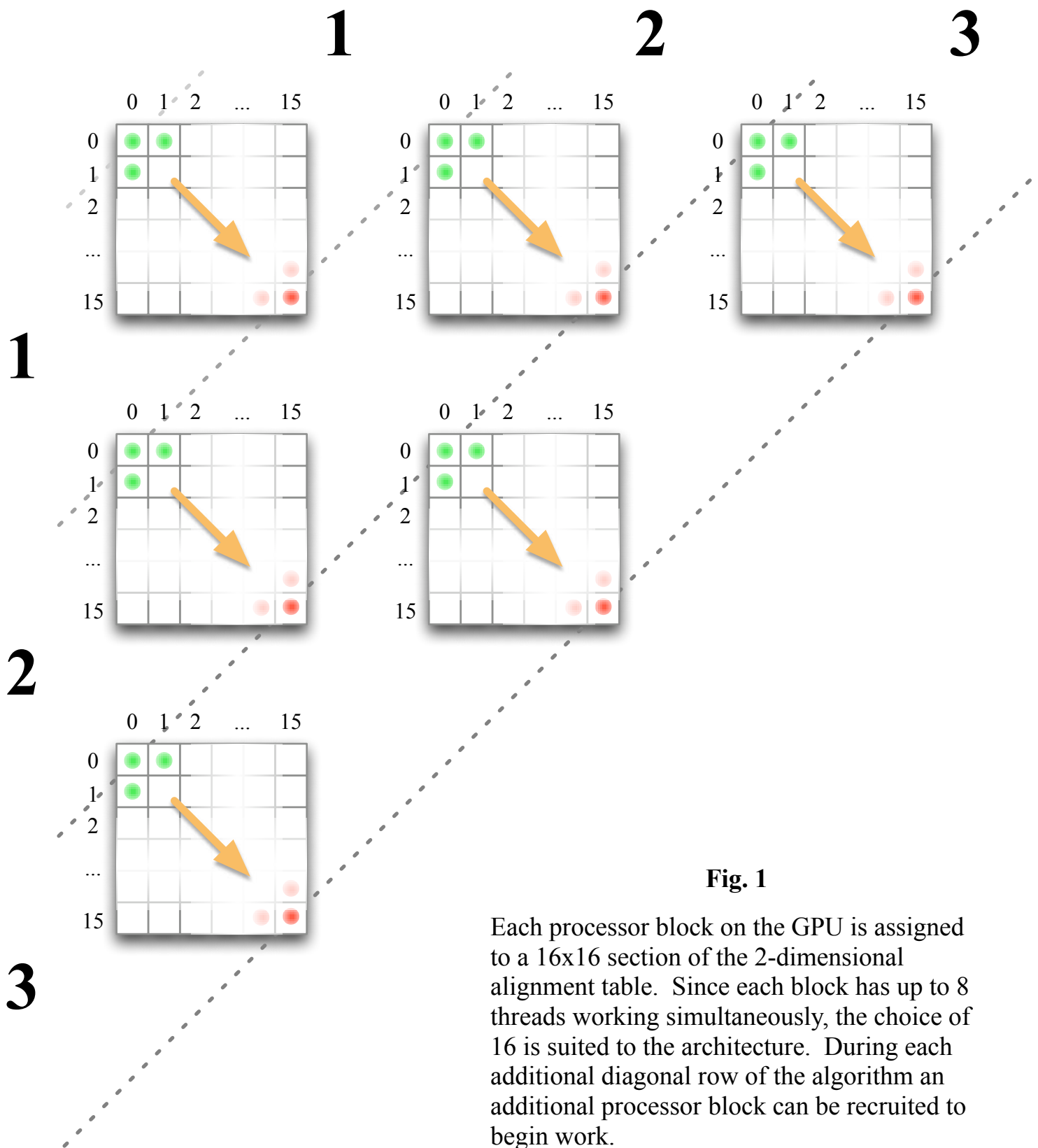
Instead of solving the problem in left-to-right and top-to-bottom fashion, this parallel algorithm works from the upper-left corner to the lower-right corner of the scoring table. It also partitions the problem into smaller units that are sized appropriately for the 240-core 8-series graphics processor 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.



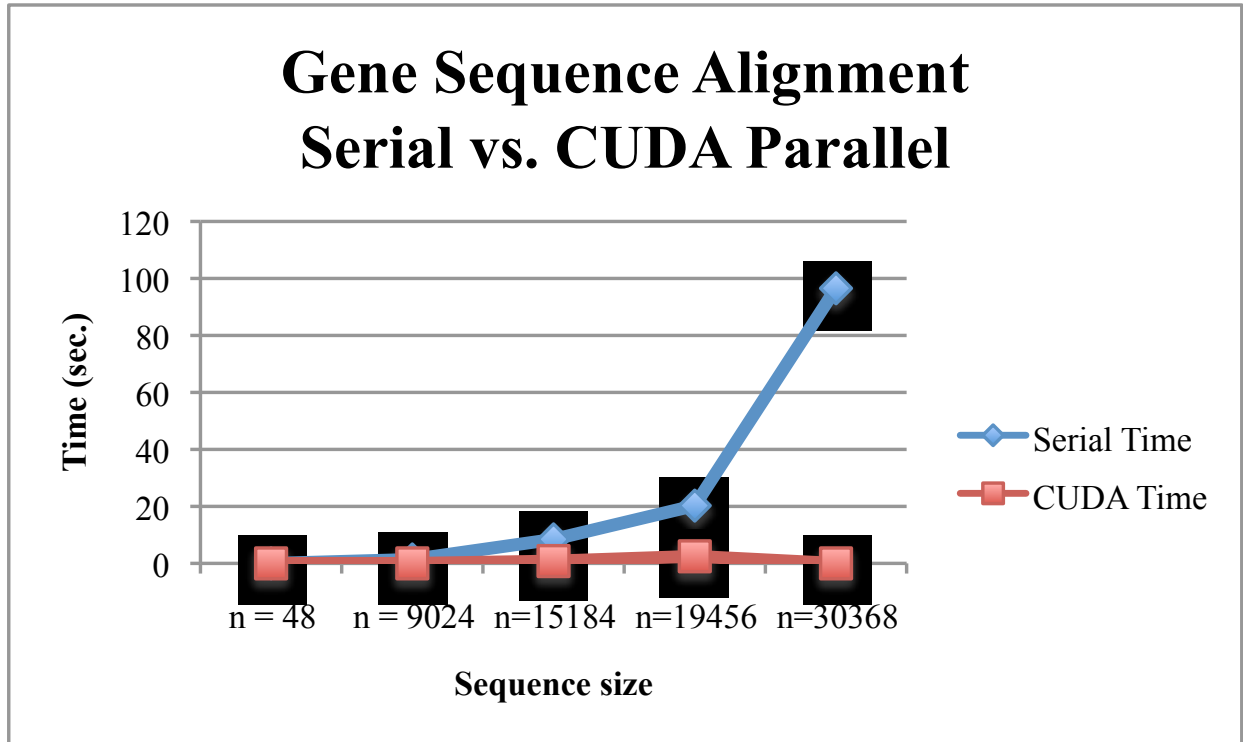


Fig. 2 Parallel alignment yields a speedup of about 10 X

	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 of $n < 20,000$ by at least an order of ten.

```

1  #include <stdio.h>
2  #include <sys/time.h>
3  #include <string.h>
4  #include <assert.h>
5  #include <cuda.h>
6
7  #include "misc.h"
8
9  #define BLOCK_SIZE 16
10 // Costs
11 #define INDEL 3
12 #define SUBST 1
13 #define MATCH 0
14
15
16 typedef struct GRID
17 {
18     int w; // Width of grid
19     int h; // Height of grid
20     int* box;
21     int success;
22 } Grid;
23
24 Grid* grid_new();
25 Grid* grid_init( Grid* g, int w, int h );
26 Grid* grid_init_file( Grid* g, char* filename );
27 Grid* grid_free( Grid* g );
28 Grid* grid_clear( Grid* g );
29 Grid* grid_copy( Grid* a, Grid* b );
30 Grid* grid_copy_to_device( Grid* g );
31 Grid* grid_copy_from_device( Grid* g );
32 Grid* grid_set_seq_row( Grid* g, char* seq, int w );
33 Grid* grid_set_seq_col( Grid* g, char* seq, int h );
34 Grid* grid_show( Grid* g );
35 Grid* grid_save( FILE* f, Grid* g );
36 Grid* grid_alignment_serial( Grid* g );
37 Grid* grid_alignment_parallel( Grid* g );
38
39 Grid* grid_new()
40 {
41     Grid* g = (Grid*)malloc( sizeof( Grid ) );
42
43     g->w = 0;
44     g->h = 0;
45     g->box = NULL;
46     g->success = true;
47
48     return g;
49 }
50
51 Grid* grid_init( Grid* g, int w, int h )
52 {
53     assert( g != NULL );
54     assert( g->success == true );
55
56     // Free memory if necessary
57     grid_free( g );
58
59     g->w = w + 2;
60     g->h = h + 2;
61     g->box = (int *)malloc( g->w * g->h * sizeof( int ) );

```

```

62
63     assert( g->box != NULL);
64
65     return g;
66 }
67
68 Grid* grid_init_file( Grid* g, char* filename )
69 {
70     assert( g != NULL );
71     assert( g->success == true );
72     assert( filename != NULL );
73     assert( strlen(filename) > 0 );
74
75     FILE* input = fopen( filename, "r" );
76     if( input != NULL )
77     {
78         int i = -1;
79         // int cur = 0;
80         char* line;
81         char* seq[2];
82         int seq_size[2] = {0, 0};
83
84         seq[0] = (char *)malloc( 1 );
85         seq[1] = (char *)malloc( 1 );
86         while( (line = readline( input )) )
87         {
88             int length = strlen( line );
89             if( length == 0 )
90             {
91                 // Skip blank lines
92             }
93             else if( line[0] == '>' )
94             {
95                 // Begin sequence after this line
96                 i++;
97             }
98             else if( i >= 0 && i <= 1 )
99             {
100                 // Grab contents and add it to our seq
101                 int j;
102                 for( j = 0; j < length; j++ )
103                 {
104                     if( line[j] != ' ' && line[j] != '\t' && line[j] != '\n' )
105                     {
106                         seq[i][ seq_size[i]++ ] = line[j];
107                         seq[i] = (char*)realloc( seq[i], seq_size[i] + 1 );
108                     }
109                 }
110             }
111             free( line );
112         }
113         // Be nice and add a trailing null char so sequences can be printed
114         seq[0][ seq_size[0] ] = '\0';
115         seq[1][ seq_size[1] ] = '\0';
116
117         // Grid sequences point to newly loaded sequences
118         grid_init( g, seq_size[0], seq_size[1] );
119         grid_set_seq_row( g, seq[0], seq_size[0] );
120         grid_set_seq_col( g, seq[1], seq_size[1] );
121
122         fclose( input );

```

```

123     }
124     else
125     {
126         g->success = false;
127     }
128
129     return g;
130 }
131
132 Grid* grid_free( Grid* g )
133 {
134     assert( g->success == true );
135
136     if( g->box != NULL )
137     {
138         free( g->box );
139         g->box = NULL;
140     }
141
142     return g;
143 }
144
145 Grid* grid_clear( Grid* g )
146 {
147     assert( g->success == true );
148     assert( g->box != NULL );
149
150     int size = g->w * g->h * sizeof(int);
151     memset( g->box, 0, size );
152
153     return g;
154 }
155
156 // Copy grid 'a' to grid 'b'
157 Grid* grid_copy( Grid* a, Grid* b )
158 {
159     assert( a->success == true && b->success == true );
160     assert( a->w == b->w && a->h == b->h );
161     assert( b->box != NULL );
162
163     int size = a->w * a->h * sizeof(int);
164     memcpy( b->box, a->box, size );
165
166     return a;
167 }
168
169 // Copies a Grid object to the device and returns a DEVICE pointer to the copy
170 Grid* grid_copy_to_device( Grid* g )
171 {
172     assert( g->success == true );
173     assert( g->box != NULL );
174
175     // Create a temp Grid object where we will setup a Device pointer to the box data
176     Grid tmp;
177     tmp.w = g->w;
178     tmp.h = g->h;
179     tmp.success = true;
180
181     // Allocate room for the object AND the object's box data
182     Grid* grid_d;
183     int size = sizeof( int ) * tmp.w * tmp.h;

```

```

184     cudaMalloc( (void*)& grid_d, sizeof( Grid ) );
185     cudaMalloc( (void*)& tmp.box, size );
186
187     // Copy the object and the box data to the device
188     cudaMemcpy( grid_d, &tmp, sizeof( Grid ), cudaMemcpyHostToDevice);
189     cudaMemcpy( tmp.box, g->box, size, cudaMemcpyHostToDevice);
190
191     // Return the DEVICE pointer
192     return grid_d;
193 }
194
195 Grid* grid_copy_from_device( Grid* g )
196 {
197     // Copy the object from the device
198     Grid* grid_h = (Grid*)malloc( sizeof( Grid ) );
199     cudaMemcpy( grid_h, g, sizeof( Grid ), cudaMemcpyDeviceToHost);
200
201     // assert( grid_h->success == true );
202
203     // Copy the box data from the device
204     int size = sizeof( int ) * grid_h->w * grid_h->h;
205     int* box = (int*)malloc( size );
206     cudaMemcpy( box, grid_h->box, size, cudaMemcpyDeviceToHost);
207     grid_h->box = box;
208
209     assert( grid_h->box != NULL );
210
211     // Return the HOST pointer
212     return grid_h;
213 }
214
215 Grid* grid_set_seq_row( Grid* g, char* seq, int w )
216 {
217     assert( g != NULL );
218     assert( g->box != NULL );
219     assert( g->w >= w );
220
221     for( int i = 2; i < g->w; i++ )
222     {
223         g->box[i] = seq[i-2];
224     }
225
226     return g;
227 }
228
229 Grid* grid_set_seq_col( Grid* g, char* seq, int h )
230 {
231     assert( g->h >= h );
232
233     int i = 2 * g->w;
234     int j;
235     for( j = 0; j < h; j++, i+=g->w )
236         g->box[i] = seq[j];
237
238     return g;
239 }
240
241 // Show small grids as text output. NOTE: Will not work for values > 48
242 Grid* grid_show( Grid* g )
243 {
244     return grid_save( stdout, g );

```

```

245 }
246
247 Grid* grid_save( FILE* f, Grid* g )
248 {
249     fprintf( f, "Show Grid: %d x %d\n", g->w - 2, g->h - 2 );
250     int i, j;
251     for( i = 0; i < g->h; i++ )
252     {
253         for( j = 0; j < g->w; j++ )
254         {
255             int c = g->box[ i * g->w + j ];
256             if( i == 0 || j == 0 )
257             {
258                 if( c == 0 ) fprintf( f, "   " );
259                 else fprintf( f, " %c ", c );
260             }
261             else fprintf( f, "%3d ", c );
262         }
263         fprintf( f, "\n" );
264     }
265     return g;
266 }
267
268 __host__ __device__ int min3( int a, int b, int c )
269 {
270     return (a < b ? (a < c ? a : (b < c ? b : c)) : (b < c ? b : c));
271 }
272
273 __host__ __device__ Grid* grid_align_setup( Grid* g )
274 {
275     // Initialize corner
276     g->box[1 * g->w + 1] = 0;
277
278     // Prepare first horizontal line
279     for( int i = 2; i < g->w; i++ )
280         g->box[1 * g->w + i] = (i - 1) * INDEL;
281
282     // Prepare first vertical line
283     for( int i = 2; i < g->h; i++ )
284         g->box[i * g->w + 1] = (i - 1) * INDEL;
285
286     return g;
287 }
288
289 // Aligns a BLOCK_SIZE x BLOCK_SIZE segment of a grid. 'g' is a Grid in DEVICE memory.
290 __global__ void cuda_grid_align_block( Grid* g, int k_major )
291 {
292     int t = threadIdx.x;
293     int row = g->w;
294     int x_init, y_init;
295     int x_block = g->w / BLOCK_SIZE - 1;
296
297     if( k_major <= x_block )
298     {
299         x_init = (k_major - blockIdx.x) * BLOCK_SIZE + 2;
300         y_init = (blockIdx.x) * BLOCK_SIZE + 2;
301     }
302     else
303     {
304         x_init = (x_block - blockIdx.x) * BLOCK_SIZE + 2;
305         y_init = (k_major - x_block + blockIdx.x) * BLOCK_SIZE + 2;

```



```

306     }
307
308     // Increasing Breadth
309     for( int k = 0; k < BLOCK_SIZE * 2; k++ )
310     {
311         if( t <= k && k - t < BLOCK_SIZE)
312         {
313             int x = x_init + (k - t);
314             int y = (y_init + t) * row;
315
316             int diag = g->box[(y - row) + (x - 1)];
317             int vert = g->box[(y - row) + (x)];
318             int horz = g->box[(y) + (x - 1)];
319
320             int c1 = diag + (g->box[x] == g->box[y] ? MATCH : SUBST);
321             int c2 = vert + INDEL;
322             int c3 = horz + INDEL;
323
324             g->box[x + y] = min3(c1, c2, c3);
325         }
326         __syncthreads();
327     }
328 }
329
330 // Single-processor Alignment
331 Grid* grid_alignment_serial( Grid* g )
332 {
333     grid_align_setup( g );
334
335     // Setup for diagonal alignment solution
336     int width = g->w - 2;
337     int col = 1, row = g->w;
338
339     // Increase diagonally
340     for( int k = 0; k < 2 * width; k++ )
341     {
342         int i_max = (k < width ? k : 2 * width - k - 2);
343         for( int i = 0; i <= i_max; i++ )
344         {
345             int x, y;
346             if( k < width )
347             { // Increasing breadth
348                 x = (2 + k - i);
349                 y = (2 + i) * row;
350             }
351             else
352             { // Decreasing breadth
353                 x = (1 + width - i);
354                 y = (3 + k - width + i) * row;
355             }
356             int diag = g->box[(y - row) + (x - col)];
357             int vert = g->box[(y - row) + (x)];
358             int horz = g->box[(y) + (x - col)];
359             int c1 = diag + (g->box[x] == g->box[y] ? MATCH : SUBST);
360             int c2 = vert + INDEL;
361             int c3 = horz + INDEL;
362             g->box[x + y] = min3(c1, c2, c3);
363         }
364     }
365
366     return g;

```

```

367 }
368
369 // Aligns a grid. 'g' is a Grid in DEVICE memory.
370 Grid* grid_alignment_parallel( Grid* g, int width, int debug )
371 {
372     int blocks = width / BLOCK_SIZE;
373
374     int k = 0;
375     for( int i = 1; i <= blocks; i++ )
376     {
377         if( debug ) printf("iteration %d (>)\n", k);
378         cuda_grid_align_block<<< i, BLOCK_SIZE >>>( g, k++ );
379     }
380     for( int i = blocks - 1; i > 0; i-- )
381     {
382         if( debug ) printf("iteration %d (<)\n", k);
383         cuda_grid_align_block<<< i, BLOCK_SIZE >>>( g, k++ );
384     }
385
386     return g;
387 }
388
389 int main( int argc, char** argv )
390 {
391     char* input = shell_arg_string( argc, argv, "-f", "default.fasta" );
392     char* output = shell_arg_string( argc, argv, "-o", "" );
393     int show_alignment = shell_arg_present( argc, argv, "--show" );
394     int align_serial = shell_arg_present( argc, argv, "--serial" );
395     int show_debug = shell_arg_present( argc, argv, "--debug" );
396     int show_help = shell_arg_present( argc, argv, "-h" ) ||
397         shell_arg_present( argc, argv, "--help" );
398
399     printf( "CUDA Needleman-Wunsch\n(c) 2009 Duane Johnson\n\n" );
400     printf( "Input FASTA: %s\n", input );
401     if( strlen( output ) > 0 )
402         printf( "Output Alignment: %s\n", output );
403     printf( "Show Alignment: %d\n", show_alignment );
404     printf( "Align in %s\n", (align_serial ? "Serial" : "Parallel") );
405     if( show_debug )
406         printf( "Debug Output ON\n" );
407     if( show_help )
408     {
409         printf( "\nOptions:\n" );
410         printf( "  -f <default.fasta>    FASTA input file.\n");
411         printf( "  -o <default.align>    Table output file.\n");
412         printf( "  --show                Show table output in standard output.\n");
413         printf( "  --serial              Do the alignment in serial (rather than
parallel).\n");
414         printf( "  --debug                Show some debug output.\n");
415         printf( "  --help                This help message.\n");
416     }
417     else
418     {
419         Grid* grid_h = grid_new();
420         Grid* grid_d;
421         Grid* grid_result;
422
423         grid_init_file( grid_h, input );
424         printf("Size of Grid: %d x %d\n", grid_h->w - 2, grid_h->h - 2);
425
426         if( (grid_h->w - 2) % BLOCK_SIZE != 0 ||

```

```

427         (grid_h->h - 2) % BLOCK_SIZE != 0)
428     {
429         printf( "Aborted. Sequence must be a multiple of %d.\n", BLOCK_SIZE );
430     }
431     else
432     {
433         grid_align_setup( grid_h );
434
435         if( align_serial )
436         {
437             double s = when();
438             grid_result = grid_alignment_serial( grid_h );
439             double f = when();
440             printf("Completed alignment in %f sec.\n", (f-s));
441         }
442         else
443         {
444             printf("Copying to device...\n");
445             grid_d = grid_copy_to_device( grid_h );
446             printf("Starting parallel alignment...\n");
447             double s = when();
448             grid_alignment_parallel( grid_d, grid_h->w, show_debug );
449             double f = when();
450             printf("Completed alignment in %f sec.\n", (f-s));
451             grid_result = grid_copy_from_device( grid_d );
452             printf("Result copied to main memory.\n");
453         }
454     }
455
456     if( show_alignment )
457         grid_show( grid_result );
458
459     if( strlen( output ) > 0 )
460     {
461         FILE* out = fopen( output, "w" );
462         grid_save( out, grid_result );
463         fclose( out );
464     }
465 }
466 }
467 }
468

```