Evaluating Multicore Programming for Matrix Multiplication

Ross Maloney

November 11, 2023

Abstract

Reported here is an exploration of execution times resulting from different parallel programming approaches. All programs are written in C and executed on a Mac Pro 2019 running Linux and the `clang` compiler. Each program performed matrix multiplication on square arrays of dimensions 1000 to 10000 containing 64 bit floating point values. Results from standard programming approaches to matrix multiplication are contrasted to `OpenCL` and `OpenCilk` approaches. Determination of the control value for `OpenCL` and `OpenCilk` which yields the fastest result is first made. Matrix multiplication is taken as a typical application to which multi-cores can be applied. All data are tabulated with indicative tabulations including 16 graphs.

The availability of PCs on the market with multiple processor cores raises the question as how to make full use of the potential they offer. Such potential is reduction in execution of application programs. Using those cores in parallel provides an opportunity for achieving such a goal.

Matrix multiplication is used here as the subject application. This application was used as it is a technique often encountered in STEM computing as well as an indicator of how other algorithms might behave under comparable programming. Because of it's nature matrix multiplication can be readily broken into Multicore processing which are then mapped onto the multi-cores of a PC. The different manners of performing such mappings were the subject of this work together with their resulting application's execution times. .The PC used here was a Mac Pro 2019 having 28 hyper-threading cores and 48GB of memory. This Mac Pro was run under Debian Linux and the programs were compiled using `clang` version 14.0.6.

In this work the matrices used contain 64-bit double precision floating point values. There are differing opinions as to what precision should be used in scientific computation. Ansorge (2022) suggests (page 9) 32-bit calculation is adequate for most application. By contrast, Turner et al. (2018) use Python as their implementation platform stating (page 19) it maps onto 64-bit hardware. Occurrence of ill-condition matrices as discussed in Turner et al. (2018) (page 128) are common in scientific and engineering computation. So the error reduction resulting from higher precision 64-bit computation in applications such as matrix multiplication of high dimensions is warranted.

There are a number of approaches to programming parallel computing, and multi-cores specifically. The two of interest in this work were OpenCL as described in Munshi et al. (2012), and OpenCilk as outlined in Curtis et al. (2020). Initial work on OpenCL was performed by Apple Inc. but released as a standard in 2009, of which Bourd (2017) is a later version. By contrast, OpenCilk, under the name Cilk, started as a research project at MIT in 1994, became an Intel Inc. product, and returned to MIT in 2019 as OpenCilk. While OpenCL covers both CPUs and GPUs, OpenCilk is directed at CPUs. Both were implemented as open source. For the particular case of Multicore CPUs, did OpenCL or OpenCilk compiler and library generate application code which executed fastest?

1 Program codes used

Two broad approaches were considered and contained in separate groups. In one group standard matrix multiplication was performed. In this group successive elements of the first matrix were accessed along a row, and in the second matrix successive elements were access along a column. In the other group, the second matrix involved in the multiplication was assumed to have been transposed before the multiplication. Thus successive elements of the first matrix were again access along a row as too were successive elements of the second matrix. en the following, the first group (reference) is identified by a *r* while the second (transposed) group is identified by a *t*. In both groups the matrices were stored in memory in the standard row order of C.

Three programming approaches were used: `serial`, `OpenCl`, and `OpenCcilk`. Both the *r* and *t* groups contained each of these approaches.

All execution times were measured using the code of Figure 1. The value returned by the function `my_get_wtime()` was a 64-bit integer containing the *time of day* measured in nano-seconds from the standard time reference point.

```
1  /* Version 2
2   *
3   * Return clock time as a 64-bit integer in units of nano-seconds.
4   *
5   * Coded by:   Ross Maloney
6   * Date:       25 November 2022
7   */
8
9  #include <time.h>
10 #include <unistd.h>
11
12 long my_get_wtime()
13 {
14   struct timespec ts;
15   long            seconds;
16
17   clock_gettime(CLOCK_MONOTONIC, &ts);
18   seconds = ts.tv_sec;
19   return(seconds*1000000000 + ts.tv_nsec);
20 }
```

Figure 1: C function used to measure execution time

Table 1 summarises the approaches used in this work. Each approach was contained in a separate file. Included in that file were loops to print the top 10 by 10 sub-matrix of the resulting product matrix. This was used to check the correctness of the product produced, it being assumed if this sub-matrix was correct then the whole product matrix would also be correct.

Included in these files was code to generate numeric values to fill the elements of the two matrices being multiplied. One set of elements were of known values. Then were initially used to check the accuracy of the resulting product and thus the functioning of the code of the file. The second set was of random values in the range 0 to 100, minus 50. These matrices were used when measuring execution time. The filling of these matrices was not included in the execution time measured. However setting up those matrices for processing by the matrix multiplication approach contained in the file was included in the execution time measure.

Performing the matrix multiplication in a serial manner provided a base for checking the parallel programming approaches and also gave a execution time baseline. This baseline could be reduced if the code was optimized at compile time as well as changing the order in while elements in the two matrices were handled. These approaches are the top four in the Table 1 tabulation. The approaches tabulated below them used parallel computing.

The parallel computing approaches were divided into two; using OpenCL and using OpenCilk. Pinned memory was also used in the OpenCL with normal memory access. In both approaches tiled and non-tiled partitioning of the matrices were used. Optimization of the OpenCL approach was seen in preliminary work as having little effect on the execution time of the resulting code.

Table 1: Catalogue of methods used

Identifier	code listing	language	loop order used						Opt
refc/tstdc ijk	Figure 2	clang	i	j	k				
refc/tstdc ijk -O3		clang	i	j	k				-O3
refc/tstdc ikj		clang	i	k	j				
refc/tstdc ikj -O3		clang	i	k	j				-O3
r/tclnon	Figure 3	OpenCL	i						
r/tclpinned		OpenCL	i						
r/tcltiled		OpenCl	i	j					
rcilk	Figure 4	OpenCilk	i	k	j				-O3
tcilk		OpenCilk	i	j	k				-O3
rcilktiled		OpenCilk	i	k	j	i	k	j	-O3
tcilktiled		OpenCilk	i	j	k	i	k	j	-O3

Also, if the second matrix being multiplied was accessed by column instead of by row, the reduction in address calculation was thought to reduce execution time of the multiplication. This would be the case if the second matrix had been transposed before the multiplication. This gave rise to the regular (`r`) and transposed (`t`) groups.

Figure 2 lists a program which performs serial multiplication of two matrices `A[][]` and `B[][]`. After forming the product it prints the execution time taken to form the product, prints that time, and then prints the top 10 by 10 portion of the product result for checking. These parts of the code were used in measuring both the execution time and the correctness of the product produced.

All the configurations used for serial multiplication in this work are contained in Figure 2. Line 5 sets the dimension of the matrix multiplication. Lines 17 and 18 together with lines 23 to 26 were used to obtain a matrix produce which was used to check correct operation of all other code used in this work. These lines filled the `A[][]` and `B[][]` matrices with known values. Execution timing was measured using lines 27 and 28 which filled the `A[][]` and `b[][]` matrices with random 64-bit floating point values less than 100.0 in absolute value, negative below, positive above 50.0. Lines pertaining to checking as opposed to measuring were removed when the opposite operation was performed.

Lines 34 through 36 control the order in which the elements of the three matrices are accessed. Line 34 is the `i` loop, line 35 is the `j` loop, and line 36 is the `k` loop. The order of these lines were changed in different configurations of this program.

Figure 2 is the *r* group configuration of the serial multiplication. When line 37 was changed to:

```
C[i][j] += A[i][k] * B[j][k];
```

the *t* group configuration resulted. The listing in Figure 2 is a composite from which pieces were removed to produce the code to perform the required purposes. After the code of Figure 2 was configured

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 15

double A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

int main()
{
  int i, j, k;
  long ticks, my_get_wtime(void);
  double marker;
  double markerA, markerB;
  int ii, jj;

  markerA = 1.0f;
  markerB = SIZE*SIZE;

  srand(time(NULL));
  for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++) {
      markerA++;
      markerB--;
      A[i][j] = markerA;
      B[i][j] = markerB;
      A[i][j] = rand() % 100 - 50;
      B[i][j] = rand() % 100 - 50;
      C[i][j] = 0.0f;
    }

  ticks = my_get_wtime();

  for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
      for (k = 0; k < SIZE; k++)
        C[i][j] += A[i][k] * B[k][j];

  ticks = my_get_wtime() - ticks;
  marker = ticks;

  printf("DIM = %d   Elapse time: %.11f milli-sec\n",
         SIZE, marker/1000000.0);
  printf("\n");
  for (ii = 0; ii < 10; ii++) {
    for (jj = 0; jj < 10; jj++)
      printf("%.1f  ", C[ii][jj]);
    printf("\n");
  }

  return(0);
}
```

Figure 2: Standard C code to perform serial matrix multiplication

and contained in a file named `refc.c` it was compiled using the command:

```
clang refc.c my_get_wtime.c -o refc -O3
```

where `my_get_wtime.c` was the file containing the timer code of Figure 1 and `-O3` was the optimization level, if appropriate.

Figure 3 contains OpenCL code used in this work. Line 5 defines the size of the matrices being multiplied and thus the dimension of the execution. Lines 9 through 28 contain the kernel code. Lines 70 through 116 set up the OpenCL system for executing the kernel together with preparing the data for transfer to, and from, the kernel code. Line 118 to 128 display the execution time of the OpenCL implementation of matrix multiplication and give a 10 by 10 sample of the resulting product for checking.

As in Figure 2, lines 52 through 63 of the Figure 3 code set up the matrices. The lines containing the `markerA` and `markerB` variables set up the `A` and `B` matrices for checking for the correct functioning of the code. These lines were remove when measuring execution time.

Line 101 contains the parameters which controls how the kernel code accesses the matrix data passed to it.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <CL/cl.h>
4
5  #define SIZE 10
6
7  double A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
8
9  const char *programSource =
10 "__kernel \n"
11 "void matmul(__global double *A, \n"
12 "            __global double *B, \n"
13 "            __global double *C) \n"
14 "{                              \n"
15 "  int i, dx, dy, lx, ly;       \n"
16 "  int gx, gy;                  \n"
17 "                               \n"
18 "  const int xhi = get_global_id(0); \n"
19 "  const int yhi = get_global_id(1); \n"
20 "  const int N = get_global_size(0); \n"
21 "  double acc;                       \n"
22 "                                    \n"
23 "  acc = 0;                          \n"
24 "  for (i = 0; i < N; i++)           \n"
25 "    acc += B[i*N + xhi] * A[yhi*N + i]; \n"
26 "  C[yhi*N + xhi] = acc;             \n"
27 "                               \n"
28 "} \n";
29
```

Continues on next page

```
30 int main()
31 {
32   int  i, j;
33   long  ticks;
34   long  my_get_wtime(void);
35   double marker;
36   double  markerA, markerB;
37   cl_platform_id    platform;
38   cl_device_id      device;
39   cl_int            ret;
40   cl_uint           ret_num_devices, ret_num_platforms;
41   cl_context        context;
42   cl_mem            a_mem_obj, b_mem_obj, c_mem_obj;
43   cl_command_queue command;
44   cl_program        program;
45   cl_kernel         kernel;
46   size_t            global[3];
47   size_t            local[3];
48   long              dataSize;
49   size_t            size_ret;
50
51 /* prepare data */
52   markerA = 1.0f;
53   markerB = SIZE*SIZE;
54   for (i = 0; i < SIZE; i++)
55     for (j = 0; j < SIZE; j++)  {
56       markerA++;
57       markerB--;
58       A[i][j] = markerA;
59       B[i][j] = markerB;
60       A[i][j] = rand() % 100 - 50;
61       B[i][j] = rand() % 100 - 50;
62       C[i][j] = 0.0f;
63     }
64
65   dataSize = SIZE * SIZE * sizeof(double);
66
67   ticks = my_get_wtime();
68
69 /* setup CPU */
70   ret = clGetPlatformIDs(1, &platform, &ret_num_platforms);
71   ret = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &device,
72                        &ret_num_devices);
73
74   context = clCreateContext(NULL, 1, &device, NULL, NULL, &ret);
75   command = clCreateCommandQueueWithProperties(context, device, 0, &ret);
76
77   a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, dataSize,
78                              NULL, &ret);
79   b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, dataSize,
80                              NULL, &ret);
81   c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY, dataSize,
82                              NULL, &ret);
```

Continues on next page

```
83
84    ret = clEnqueueWriteBuffer(command, a_mem_obj, CL_TRUE, 0, dataSize,
85                               A, 0, NULL, NULL);
86    ret = clEnqueueWriteBuffer(command, b_mem_obj, CL_TRUE, 0, dataSize,
87                               B, 0, NULL, NULL);
88    ret = clEnqueueReadBuffer(command, c_mem_obj, CL_TRUE, 0, dataSize,
89                              C, 0, NULL, NULL);
90
91    program = clCreateProgramWithSource(context, 1,
92                            (const char **)&programSource, NULL, &ret);
93    ret = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
94    kernel = clCreateKernel(program, "matmul", &ret);
95
96    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_mem_obj);
97    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_mem_obj);
98    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_mem_obj);
99
100   global[0] = SIZE;  global[1] = SIZE;
101   local[0] = 5;  local[1] = 5;
102   ret = clEnqueueNDRangeKernel(command, kernel, 2, NULL, global,
103                                local, 0, NULL, NULL);
104
105   ret = clEnqueueReadBuffer(command, c_mem_obj, CL_TRUE, 0, dataSize,
106                             C, 0, NULL, NULL);
107
108   ret = clFlush(command);
109   ret = clFinish(command);
110   ret = clReleaseKernel(kernel);
111   ret = clReleaseProgram(program);
112   ret = clReleaseMemObject(a_mem_obj);
113   ret = clReleaseMemObject(c_mem_obj);
114   ret = clReleaseCommandQueue(command);
115   ret = clReleaseContext(context);
116
117   ticks = my_get_wtime() - ticks;
118   marker = ticks;
119
120   printf("DIM = %d    Elapse time: %.11f milli-sec\n",
121          SIZE, marker/1000000.0);
122   printf("\n");
123   for (i = 0; i < 10; i++)  {
124     for (j = 0; j < 10; j++)
125       printf("%.1f  ", C[i][j]);
126     printf("\n");
127   }
128
129   return(0);
130 }
```

Figure 3: OpenCL C code to perform parallel matrix multiplication

In the OpenCL pinned memory case the `CL_MEM_READ_ONLY` in lines 77 and 79 were replaced by `CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PTR`. Also `CL_MEM_WRITE_ONLY` in line 81 was replaced by `CL_MEM_WRITE_ONLY | CL_MEM_ALLOC_HOST_PTR`.

For the `t` (transposed) group of OpenCL, only line 25 was replace by the line:

```
acc += A[yhi*N + i] * B[xhi*N + i];
```

The kernel code in lines 9 through 28 of Figure 3 for the OpenCL tiled case was replaced by the code:

```
1  const char *programSource =
2  "__kernel \n"
3  "void matmul(__global double *A, \n"
4  "            __global double *B, \n"
5  "            __global double *C) \n"
6  "{                              \n"
7  "  int  i;                     \n"
8  "                              \n"
9  "  const int xhi = get_global_id(0); \n"
10 "  const int yhi = get_global_id(1); \n"
11 "  const int N = get_global_size(0); \n"
12 "  double  acc;                        \n"
13 "                                      \n"
14 "  acc = 0;                            \n"
15 "  for (i = 0; i < N; i++)             \n"
16 "    acc += B[i*N + xhi] * A[yhi*N + i]; \n"
17 "  C[yhi*N + xhi] = acc;               \n"
18 "                              \n"
19 "} \n";
```

while in the `t` (transposed) group, line 25 in the tiled kernel code was replaced by:

```
acc += A[yhi*N + i] * B[xhi*N + i];
```

The particular style of writing OpenCL programs in this work was to include the kernel and mainline in the one file. If that file was named `rclnon.c` which contained the code of Figure 3 then it was compiled using the command:

```
clang rclnon.c my_get_wtime.c -o rclnom -l OpenCL
```

where `my_get_wtime.c` was the file containing the timer code of Figure 1 and `OpenCL` was the OpenCL library linked to the `clang` compiler and linker.

Figure 4 shows the code used for the `r` group OpenCilk matrix multiplication. Line 6 sets the dimension of the matrices and thus the amount of execution performed. Lines 18 through 31 set up the contents of the matrices. The lines containing the `markerA` and `markerB` variables were used in checking the correct operation of the code. They were removed for measuring execution time. Lines 35 to 37 contain the loops which executes the matrix multiplication code of line 38. Lines 40 to 50 determined the execution time, display that measure, then print the top left-hand 10 by 10 part of the result matrix for checking.

For the `t` (transposed) group. line 38 Figure 4 was changed to:

```
C[i][j] += A[i][k] * B[j][k];
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cilk/cilk.h>

#define SIZE 10

double A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

int main()
{
  int j, k;
  int ii, jj;
  long ticks, my_get_wtime(void);
  double marker;
  double markerA, markerB;

  markerA = 1.0f;
  markerB = SIZE*SIZE;

  srand(time(NULL));
  for (ii = 0; ii < SIZE; ii++)
    for (jj = 0; jj < SIZE; jj++) {
      markerA++;
      markerB--;
      A[ii][jj] = markerA;
      B[ii][jj] = markerB;
      A[ii][jj] = rand() % 100 - 50;
      B[ii][jj] = rand() % 100 - 50;
      C[ii][jj] = 0.0f;
    }

  ticks = my_get_wtime();

  cilk_for (int i = 0; i < SIZE; i++)
    for (k = 0; k < SIZE; k++)
      for (j = 0; j < SIZE; j++)
        C[i][j] += A[i][k] * B[k][j];

  ticks = my_get_wtime() - ticks;
  marker = ticks;

  printf("DIM = %d   Elapse time: %.11f milli-sec\n",
         SIZE, marker/1000000.0);
  printf("\n");
  for (ii = 0; ii < 10; ii++) {
    for (jj = 0; jj < 10; jj++)
      printf("%.1f  ", C[ii][jj]);
    printf("\n");
  }

  return(0);
}
```

Figure 4: OpenCilk C code to perform matrix multiplication

For the OpenCilk tiled, lines 35 to 38 of the code in Figure 4 were replaced by the code:

```
1    cilk_for (int ih = 0; ih < SIZE; ih += S)
2        cilk_for (int kh = 0; kh < SIZE; kh +=S)
3      for (jh = 0; jh < SIZE; jh +=S)
4          for (il = 0; il < S; ++il)
5       for (kl = 0; kl < S; ++kl)
6     for (jl = 0; jl < S; ++jl)
7       C[ih+il][jh+jl] += A[ih+il][kh+kl] * B[kh+kl][jh+jl];
```

In this code segment and in Figure 4 the loop sequence `i, j, k` was changed to `i, k, j` which Leiserson (2018) shows reduces execution time. In the `t` (transposed) group, line 7 in that code was changed to:

```
C[ih+il][jh+jl] += A[ih+il][kh+kl] * B[jh+jl][kh+kl];
```

All OpenCilk code for each example was contained in one file. If that file was named `rcilk.c` then the command to compile and link the executable was:

```
clang -fopencilk -O3 rcilk.c my_get_wtime.c -o rcilk
```

where `rcilk.c` is the name of the OpenCilk example code and `my_get_wtime.c` contains the source code listed in Figure 1 for measuring the execution time.

2 Finding control values

Depending on the approach use, execution time of matrix multiplication can be influenced by the manner of coding. In the serial matrix multiplication code of Figure 2, the order of loops 34 to 36 can be changed to reduce the execution time. Employing a transpose of the second matrix is an alternative. Such changes can also be used in the OpenCilk code of `rcilk`. However the execution times of the OpenCL of Figures 3 and OpenCilk code of Figure 4 in controlled by a parameter in the code. For OpenCL this is the parameter setting of the `local` array in line 101 of Figure 3. From Leiserson (2018) for OpenCilk it is the value of the parameter `S` in the tiled approach. In both cases the parameter is controlling memory size use.

Experiments were performed to determine the value of those control parameters in all five parallel processing approaches in both the reference (`r` group and transposed (`t`) group. For a particular approach a single value was to be applied to all matrix sizes under consideration. That value was to yield the minimum, or near minimum, execution time for all those matrix sizes.

A particular set of values were tried for being the control value. The values 2, 4, 5, 10, 20, 25, 50, and 100 were tried. Each of those values were multiples of the matrix sizes being considered. Matrix sizes 2000, 4000, 6000, and 8000 were used in this search judged adequate to cover the 1000 to 10000 matrix sizes considered in this work.

Each file containing an approach as in Table 1 was configured for each combination for candidate control variable and matrix size. The file was compiled and executed once. The execution time displayed was recorded. Those recordings together with the approach and control value are show in Table 2 and Table 3. In each case the multiplied matrices contained elements of known values and thus the accuracy of the result matrix produced was checked.

In those tables, the `Reference` entry is the serial multiplication execution time corresponding to the matrix size using `i, j, k` loop ordering and no optimization level. Also the `rcilk` values were not effected by control value but were shown for comparison purposes. As expected in each size grouping

in the tables, the execution times for the parallel approaches are less than the `Reference` entry. The tiled approaches would be expected to execute faster than their associated un-tiled approaches. This is true with respect to the OpenCilk approach. In the OpenCL approach of Table 2 this was true except for matrix size 2000. This was not true in the matrix transposed group of Table 3. Checking of the result matrix produced in all cases suggested the coding to be correct; the same code was used in all cases.

Table 2: Quick search of speed control for reference matrix multiplication

dimension	Reference	value	rclnon	rclpined	rcltiled	rcilk	rcilktile
2000	18891.6					115.7	
		2	514.4	443.6	473.9		115.1
		4	545.0	527.4	745.4		86.2
		5	418.6	418.9	728.8		102.5
		10	499.9	416.7	702.4		91.8
		20	502.4	441.7	516.5		97.4
		25	504.2	438.3	x		98.0
		50	559.6	465.9	x		107.1
		100	0	0	0		123.5
4000	408974.7					1088.2	
		2	5959.4	6024.0	4079.3		800.5
		4	6193.5	6170.1	5331.6		480.4
		5	6157.9	6122.7	4953.5		588.9
		10	5750.3	5680.9	4461.4		519.2
		20	6389.5	6340.3	2040.4		619.9
		25	6946.0	6865.9	x		639.4
		50	9697.5	9548.1	x		656.5
		100	0	0	0		705.0
6000	916924.9					3668.3	
		2	19318.8	20920.6	12522.4		2627.6
		4	22786.5	22964.1	18185.6		1848.2
		5	21356.9	21255.5	16704.7		1892.6
		10	24533.8	24524.2	14856.9		2171.3
		20	24527.4	24021.3	9070.9		2633.8
		25	20247.1	20455.9	x		2458.2
		50	30216.9	29862.3	x		2112.7
		100	0	0	0		2310.7
8000	4060482.2					8809.9	
		2	64415.1	64999.6	39893.2		6142.9
		4	61786.6	61429.5	38732.8		4781.3
		5	70619.2	69750.9	39673.9		4746.6
		10	132923.3	134009.8	36084.7		4812.2
		20	210068.2	206520.9	21961.1		8378.2
		25	182990.6	180663.9	x		6632.4
		50	115300.8	118498.2	x		5066.0
		100	0	0	0		5332.6

Note in Table 2 and Table 3 tne `x` and `0` entries. The `x` case records where the multiplication result matrix contained error values. The `0` case records where the result matrix produced containing zeros only. The control values associates with those results were neglected. They occur in the same places in Table 2 and Table 3.

From the data in Table 2 and Table 3 the control values in Table 4 were obtained. Such values are the same for both the `r` and `t` groups except for the OpenCl tiled approach (`rcltiled` and `tcltiled`). A control value was selected for an approach overall and all matrix sizes in that approach, but in some

instances the value adopted does not correspond to the minimum execution time at all matrix sizes.

Table 3: Quick search of speed control for transposed matrix multiplication

dimension	Reference	value	tclnon	tclpined	tcltiled	tcilk	tcilktile
2000	16236.8					226.8	
		2	468.5	402.3	433.2		106.7
		4	484.7	380.8	712.9		82.2
		5	414.7	403.8	714.5		91.0
		10	375.8	384.9	685.1		104.8
		20	399.3	410.4	513.3		95.1
		25	404.6	402.6	x		108.5
		50	410.6	404.6	x		174.3
		100	0	0	0		182.0
4000	133634.5					1665.0	
		2	1967.1	2003.6	2110.9		776.9
		4	1756.4	1800.0	4296.5		521.4
		5	1797.7	1801.0	4263.0		565.8
		10	1807.7	1820.9	4045.2		654.1
		20	1808.7	1820.9	2723.1		593.9
		25	1830.4	1835.6	x		646.2
		50	2115.1	2095.9	x		1166.4
		100	0	0	0		1303.8
6000	455283.9					5497.4	
		2	6569.6	6471.8	6938.0		3229.9
		4	5591.9	5567.7	13961.8		1938.8
		5	5579.0	5604.5	13789.1		1966.2
		10	5758.6	5756.3	13594.8		2178.5
		20	5831.1	5805.1	8691.7		2047.8
		25	6080.1	6159.3	x		2219.2
		50	6943.1	6847.1	x		3896.0
		100	0	0	0		3854.7
8000	1076570.9					12920.6	
		2	16638.4	16625.0	17274.6		6510.9
		4	13004.9	13165.2	32604.3		4346.5
		5	13028.4	13011.6	32381.8		4736.2
		10	13415.3	13439.8	31869.7		5744.3
		20	14014.1	14120.2	20749.2		5843.3
		25	15828.4	15723.6	x		5372.8
		50	15698.1	15814.2	x		9770.5
		100	0	0	0		12386.5

Table 4: Control parameter value selected for each parallel approach

rclnon	5
rclpinned	5
rcltiled	20
rcilktiled	4
tclnon	5
tclpinned	5
tcltiled	2
tcilktiled	4

3 Data

Table 5 and Table 6 record data of running the programs of Section 1 with the control variables of Table 4. Table 5 contains data relating to matrix multiplication in a *normal* manner (group `r`) while Table 6 assumed the second matrix in the multiplication had been transposed (group `t`).

Table 5: Milli-second execution times for standard matrix multiplication using different approaches

dimemsion	method	min	mean	max
10000	refc ijk	6126232.3	6412915.7	6524000.3
	refc ijk -O3	3215057.1	3375421.0	3394645.0
	refc ikj	1883818.3	1911570.7	1921813.8
	refc ikj -O3	553269.8	580625.3	600819.6
	rclnon	142454.8	144520.0	147276.4
	rclpinned	142431.6	144544.9	146433.7
	rcltiled	41273.6	41465.8	41659.9
	rcilk	16306.1	17111.6	17430.5
	rcilktiled	8146.1	9634.2	11197.2
9000	refc ijk	3065278.8	3166870.7	3230376.3
	refc ijk -O3	1566319.7	1600727.5	1614482.8
	refc ikj	1366675.5	1392771.8	1406623.7
	refc ikj -O3	411817.9	429753.7	435612.5
	rclnon	71079.0	75082.1	77770.7
	rclpinned	72984.0	75027.0	76694.5
	rcltiled	30311.6	30421.8	30515.9
	rcilk	12184.8	12526.7	14599.6
	rcilktiled	5830.5	6771.0	8347.0
8000	refc ijk	3880762.2	3973611.9	4078624.0
	refc ijk -O3	2137535.3	2205810.6	2247684.6
	refc ikj	960184.6	979998.2	982507.3
	refc ikj -O3	279133.1	300284.3	304771.1
	rclnon	67391.0	69036.5	70813.2
	rclpinned	67219.0	68758.7	70879.3
	rcltiled	21905.3	21975.5	22068.2
	rcilk	8170.3	8730.1	9690.5
	rcilktiled	4076.6	4791.1	5548.7
7000	refc ijk	1218387.0	1263997.5	1286114.2
	refc ijk -O3	614078.7	622680.2	625982.2
	refc ikj	646136.8	655318.6	658160.5
	refc ikj -O3	191637.1	200120.0	203548.7
	rclnon	17446.0	18193.5	18908.0
	rclpinned	17616.8	18380.0	18995.5
	rcltiled	14196.0	14294.3	14398.1
	rcilk	5520.7	5849.9	6006.8
	rcilktiled	2695.5	3177.2	4401.9
6000	refc ijk	886629.0	912064.9	925541.0
	refc ijk -O3	415646.5	425851.8	430862.4
	refc ikj	403888.9	411763.9	413284.2
	refc ikj -O3	118983.9	124427.3	126983.6
	rclnon	20275.6	20890.3	21232.1
	rclpinned	20282.3	21003.6	21544.3
	rcltiled	8988.0	9030.8	9102.4
	rcilk	3547.8	3701.5	3821.4
	rcilktiled	1531.2	1816.2	2230.5

Continued on next page

dimemsion	method	min	mean	max
5000	refc ijk	447859.0	460195.8	465215.5
	refc ijk -O3	210671.7	216007.7	217651.9
	refc ikj	237416.2	238532.1	239471.8
	refc ikj -O3	67176.3	71091.2	72600.0
	rclnon	4925.8	5004.4	5044.2
	rclpinned	4945.0	5016.3	5058.3
	rcltiled	5262.9	5303.1	5343.4
	rcilk	2072.6	2167.9	2510.5
	rcilktiled	955.2	1095.5	1395.9
4000	refc ijk	399818.9	406195.8	410563.6
	refc ijk -O3	163680.6	170792.3	172048.7
	refc ikj	120219.3	121507.1	122037.7
	refc ikj -O3	33471.9	35671.5	36343.4
	rclnon	6032.4	6101.5	6189.1
	rclpinned	6028.1	6110.3	6179.8
	rcltiled	2899.6	2925.9	2947.2
	rcilk	1057.1	1107.3	1177.2
	rcilktiled	441.8	490.8	633.2
3000	refc ijk	90772.0	93149.3	94979.6
	refc ijk -O3	40796.9	41431.7	41959.5
	refc ikj	50431.0	51000.6	51398.3
	refc ikj -O3	13471.3	14608.9	14849.0
	rclnon	960.7	988.8	1010.3
	rclpinned	975.2	997.5	1012.7
	rcltiled	1264.1	1281.9	1307.9
	rcilk	464.4	484.6	494.6
	rcilktiled	205.1	214.7	245.5
2000	refc ijk	18324.0	19320.5	20454.9
	refc ijk -O3	8993.8	9607.6	10733.9
	refc ikj	14491.1	14601.2	15202.7
	refc ikj -O3	2468.6	2710.1	3166.7
	rclnon	380.2	399.0	420.8
	rclpinned	370.6	394.2	411.7
	rcltiled	482.0	495.8	515.2
	rcilk	127.8	133.5	150.4
	rcilktiled	77.2	83.9	103.7
1000	refc ijk	2269.7	2337.4	2348.3
	refc ijk -O3	1039.4	1059.3	1070.4
	refc ikj	1815.8	1856.8	1893.8
	refc ikj -O3	306.6	337.8	354.9
	rclnon	197.2	206.3	215.4
	rclpinned	200.9	209.0	220.9
	rcltiled	202.2	216.1	237.9
	rcilk	34.9	38.1	45.6
	rcilktiled	29.4	32.0	46.0

Each table shows the execution time of nine approaches to matrix multiplication; four using serial approaches and five using parallel multicore approaches. Of those five multicore approaches, three used OpenCL and two OpenCilk. Each program for each configuration was run 25 times and the execution time of each run measured. Such repeats were performed one after the other. From those 25 execution times the maximum, minimum, and mean of them was determined. These statistics indicated the distribution of such data within a particular 25 value set in which they occur. All measurements are in units of milli-seconds.

The selected control values of Table 4 which produced the execution times in Table 2 and Table 3 generally fall in the range of execution values in the corresponding entries of Table 5 and Table 6.

All execution time measurement runs were performed using the maximum cores appropriate. In the serial programs one core was used. In the parallel programs of OpenCL and OpenCilk all 56 hyper

Table 6: Milli-second execution times for transposed matrix multiplication using different approaches

dimemsion	method	min	mean	max
10000	tstdc ijk	2082958.4	2110271.5	2174791.2
	tstdc ijk -O3	957517.6	976547.0	993223.6
	tstdc ikj	6026938.6	6361206.9	6517824.2
	tstdc ikj -O3	3136153.8	3275376.8	3319590.7
	tclnon	25380.0	25592.0	25786.0
	tclpinned	25359.7	25614.5	25906.7
	tcltiled	32466.9	33030.4	33485.6
	tcilk	24471.5	25054.9	25471.8
	tcilktiled	8556.4	9587.1	11061.2
9000	tstdc ijk	1516656.0	1534996.9	1577609.2
	tstdc ijk -O3	697563.0	712722.2	724438.7
	tstdc ikj	2948074.4	3080465.3	3134138.2
	tstdc ikj -O3	1262249.5	1295244.9	1310990.2
	tclnon	18633.2	18729.2	18819.0
	tclpinned	18631.3	18715.3	18816.5
	tcltiled	22885.9	23584.1	24156.3
	tcilk	17889.4	18425.4	20156.9
	tcilktiled	6165.9	6835.2	7974.1
8000	tstdc ijk	1065740.6	1080075.8	1114902.2
	tstdc ijk -O3	490193.5	500071.2	507957.3
	tstdc ikj	3916042.2	4028316.6	4101388.3
	tstdc ikj -O3	2184960.9	2244687.4	2292376.8
	tclnon	12920.4	13011.9	13124.1
	tclpinned	12956.7	13010.3	13097.1
	tcltiled	16716.9	17145.3	17430.7
	tcilk	12534.6	12843.5	13018.3
	tcilktiled	4147.0	4732.2	5808.5
7000	tstdc ijk	714337.9	721412.5	744637.0
	tstdc ijk -O3	328329.5	333871.4	339319.6
	tstdc ikj	1173183.8	1220462.4	1236052.6
	tstdc ikj -O3	504904.1	512521.5	516930.9
	tclnon	8677.8	8709.0	8736.1
	tclpinned	8673.7	8707.3	8741.9
	tcltiled	10573.7	10888.4	11132.2
	tcilk	8349.8	8560.1	8719.0
	tcilktiled	2683.9	3050.3	3902.3
6000	tstdc ijk	449188.7	454208.9	465596.1
	tstdc ijk -O3	206771.3	210303.3	214523.4
	tstdc ikj	852012.1	876584.0	890383.5
	tstdc ikj -O3	351015.9	354730.5	357728.2
	tclnon	5554.1	5586.6	5617.2
	tclpinned	5545.0	5591.4	5640.0
	tcltiled	6818.0	6952.5	7328.1
	tcilk	5312.6	5470.9	5657.3
	tcilktiled	1676.2	1815.7	2126.6

Continued on next page

dimemsion	method	min	mean	max
5000	tstdc ijk	260336.1	262353.3	263842.7
	tstdc ijk -O3	119413.9	121942.0	128147.1
	tstdc ikj	428854.9	442209.1	448440.1
	tstdc ikj -O3	162197.2	165266.9	167236.0
	tclnon	3262.6	3299.1	3435.3
	tclpinned	3261.7	3294.7	3326.4
	tcltiled	3941.2	4060.6	4176.9
	tcilk	3095.8	3169.3	3275.8
	tcilktiled	961.6	1051.8	1204.4
4000	tstdc ijk	132729.0	134144.4	137358.2
	tstdc ijk -O3	61062.6	62361.8	65649.9
	tstdc ikj	383555.3	397627.7	409984.5
	tstdc ikj -O3	142084.5	145787.7	148745.9
	tclnon	1758.6	1790.2	1815.5
	tclpinned	1762.1	1787.7	1810.2
	tcltiled	2068.0	2130.4	2300.0
	tcilk	1592.7	1645.1	1705.2
	tcilktiled	490.6	514.3	555.1
3000	tstdc ijk	56032.0	56354.7	56765.7
	tstdc ijk -O3	25668.8	26489.2	27405.1
	tstdc ikj	87328.0	89522.7	91643.0
	tstdc ikj -O3	27627.4	28012.0	28287.7
	tclnon	841.0	863.8	879.6
	tclpinned	841.8	864.3	877.9
	tcltiled	970.8	986.4	997.5
	tcilk	690.0	706.0	713.2
	tcilktiled	215.2	221.9	243.7
2000	tstdc ijk	16163.1	16222.0	16321.5
	tstdc ijk -O3	6932.6	7199.1	7551.0
	tstdc ikj	17463.6	18387.9	20859.6
	tstdc ikj -O3	5759.9	5927.0	7267.6
	tlnon	367.2	383.8	392.1
	tclpinned	369.8	385.7	403.5
	tcltiled	398.5	412.4	426.1
	tcilk	207.3	213.6	230.6
	tcilktiled	79.0	83.3	96.4
1000	tstdc ijk	2019.7	2071.1	2133.0
	tstdc ijk -O3	855.5	867.3	897.9
	tstdc ikj	2118.6	2179.6	2199.7
	tstdc ikj -O3	659.3	664.5	684.1
	tclnon	198.0	205.3	214.9
	tclpinned	198.3	210.2	234.8
	tcltiled	201.8	211.6	223.4
	tcilk	47.4	52.7	75.6
	tcilktiled	28.4	32.4	50.9

-threading cores of the MacPro were used. This was verified using the `xpm` utility which show the cores operating to be running at full capacity. The verification was obtained by running the programs before measurement was performed while `xpm` was executing.

4 Results

Figures 5 to 20 give an overview of the data of Tables 5 and 6. Figures 5 to 12 show the data of Table 5 and indicate the execution speed variation in the reference (`r`) group. Figures 13 to 20 show the data of Table 5 and indicate the execution speed variation in the translated (`t`) group. These two sets of graphs contain the data from matrix sizes 10000, 7000, 4000, and 1000 which give an overview of the data in the range of the respective tables from highest to lowest matrix size. Error bars on all graphs represent the maximum and minimum execution times associated with the mean value which is graphed.

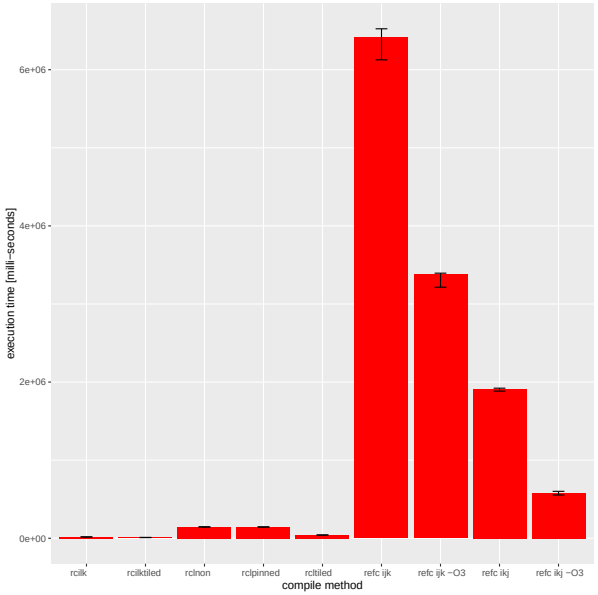

Figure 5: normal 10000 matrix multiplies

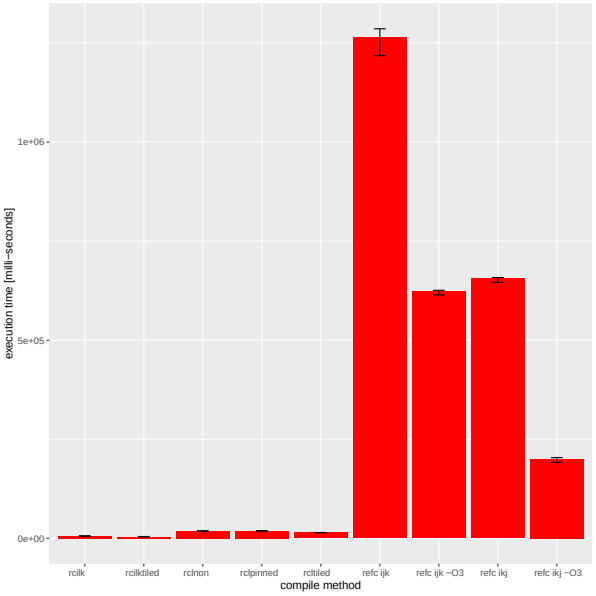

Figure 6: normal 7000 matrix multiplies

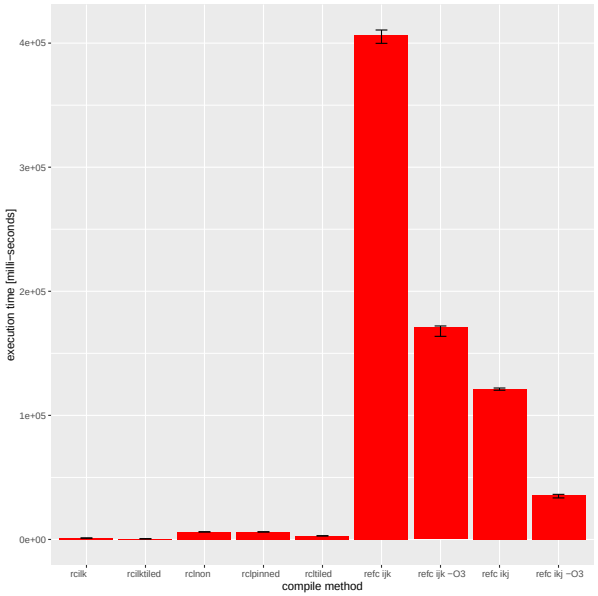

Figure 7: normal 4000 matrix multiplies

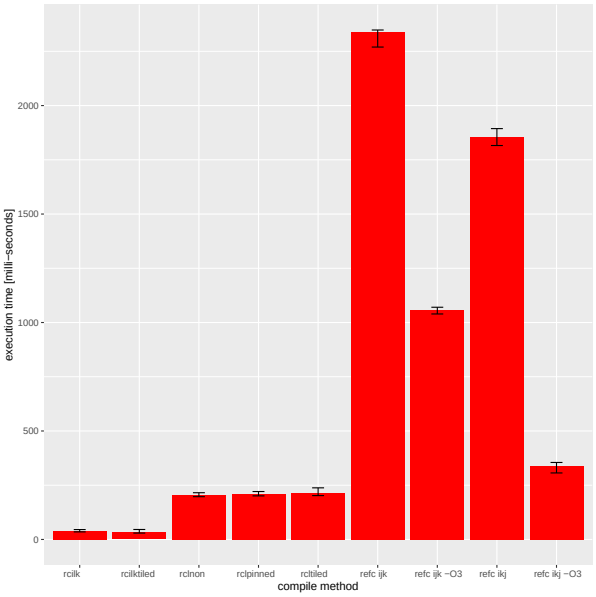

Figure 8: normal 1000 matrix multiplies

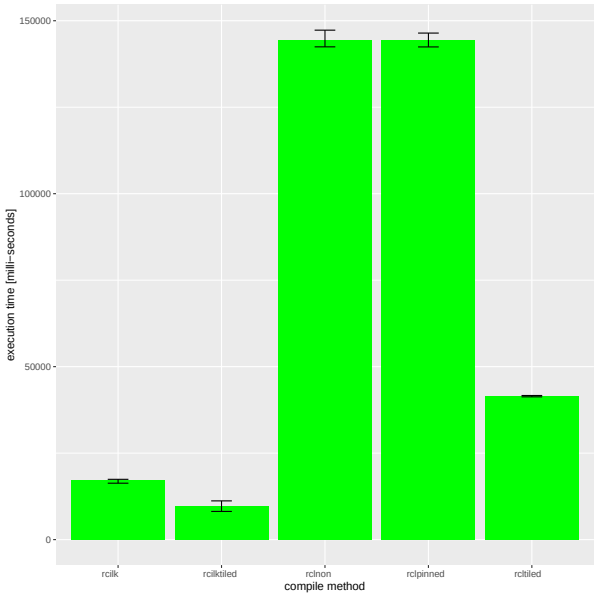

Figure 9: normal 10000 matrix parallel multiplies

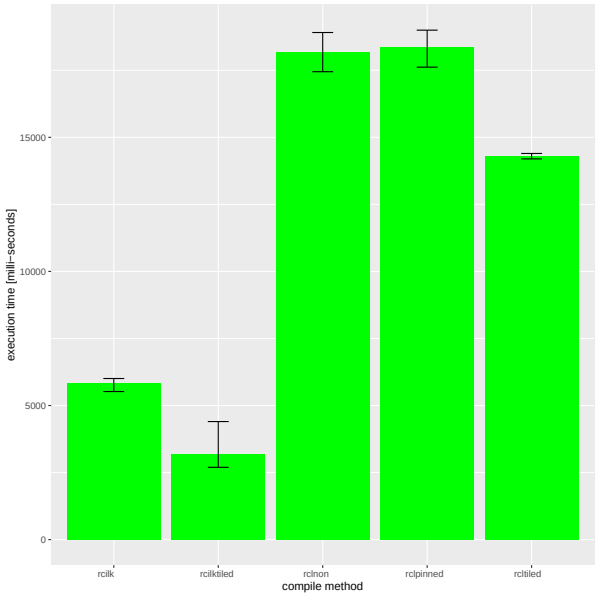

Figure 10: normal 7000 matrix parallel multiplies

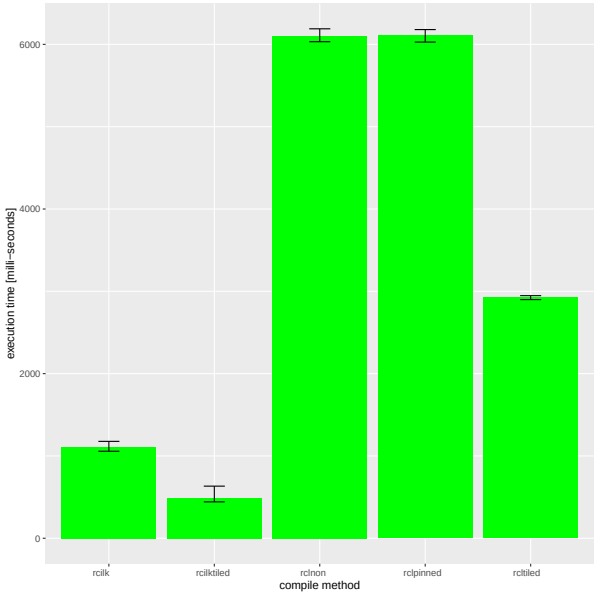

Figure 11: normal 4000 matrix parallel multiplies

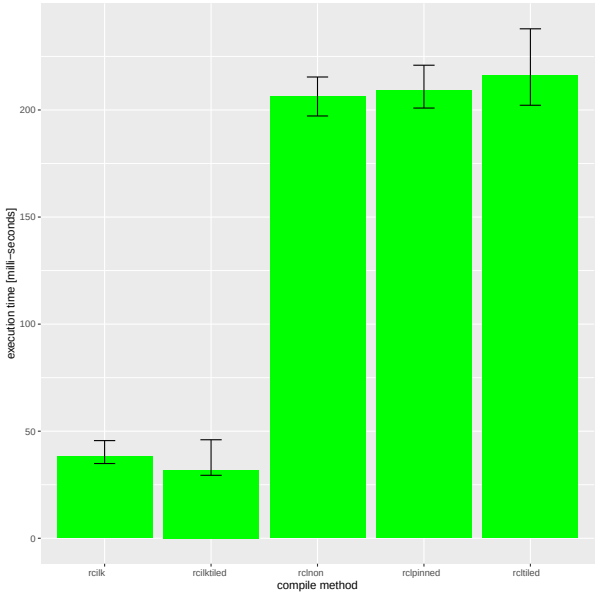

Figure 12: normal 1000 matrix parallel multiplies

Figures 5 to 8 and Figures 13 to 16 indicate the strong execution speed reduction introduced by the parallel programming techniques OpenCL and OpenCilk. Figures 9 to 12 show the variation in execution speed between the individual parallel programming contained in Figures 5 to 8. Figures 17 to 20 show the same variation contained in Figures 13 to 16.

From these graphs the following overall observations with respect to execution times of the executing codes were made:

- parallel processing using either OpenCL or OpenCilk utilize all multiple cores available had a significant effect when using either normal matrix addressing or transposed matrix addressing,

- Greater reductions occurred using OpenCilk than OpenCL.
- Use of optimization in compiling serial code produced a significant reduction,
- Changing of the order of the loops in the serial code for the multiplication process improved the speed when using normal array addressing but had the reverse affect with a transposed matrix,
- Using a transposed matrix had a significant effect when serial matrix multiplication was performed.
- Tiling produced significant reduction using OpenCilk,
- Tiling with OpenCL had a variable effect,

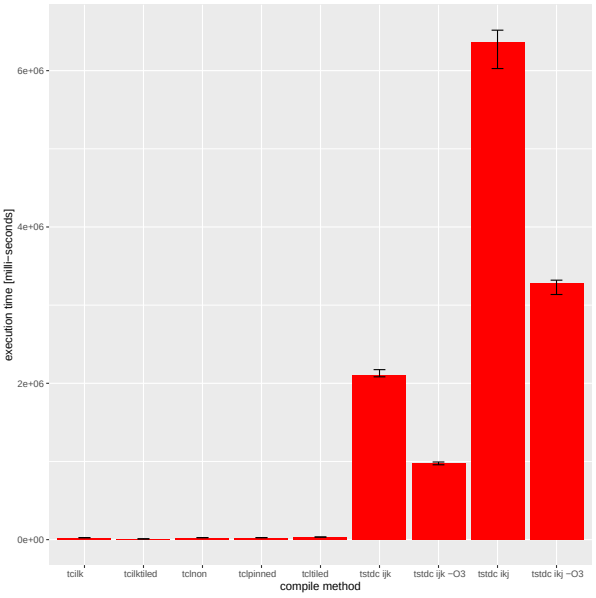

Figure 13: transposed 10000 matrix multiplies

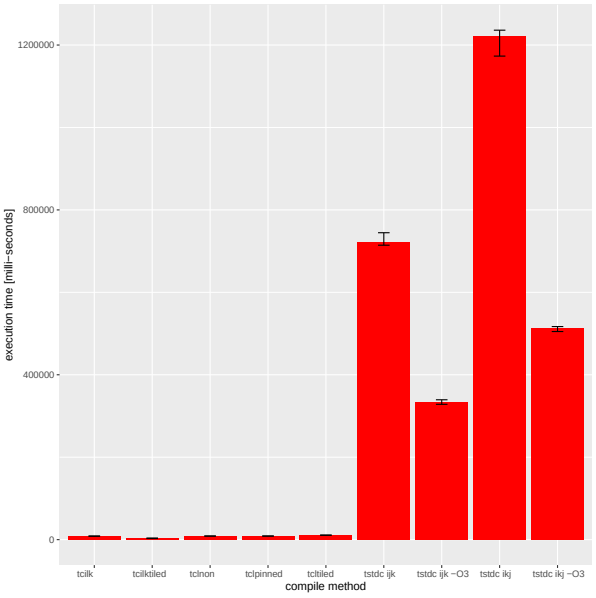

Figure 14: transposed 7000 matrix multiplies

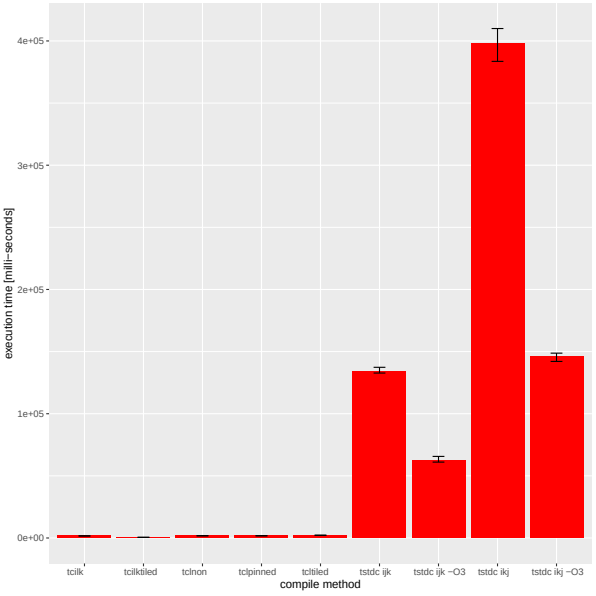

Figure 15: transposed 4000 matrix multiplies

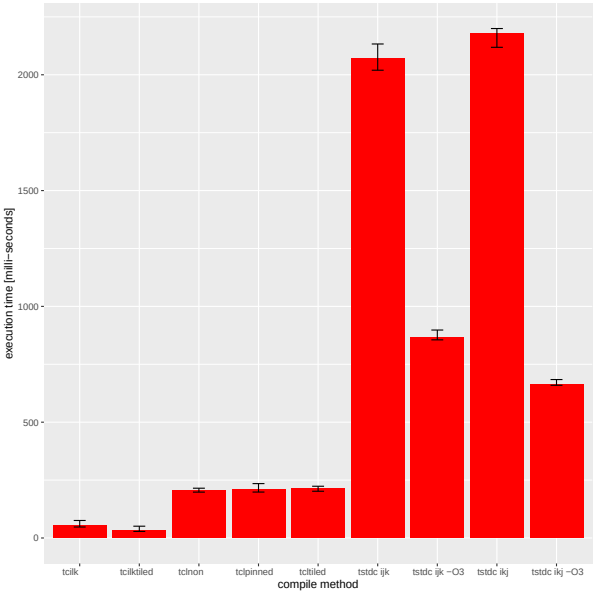

Figure 16: transposed 1000 matrix multiplies

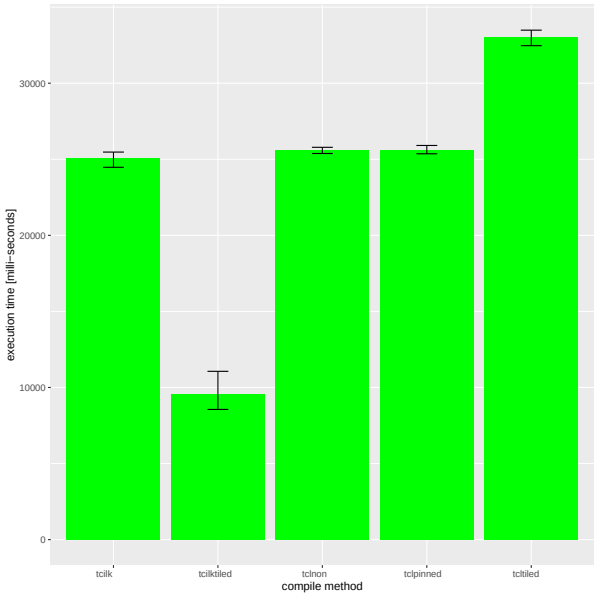

Figure 17: transposed 10000 matrix parallel multiplies

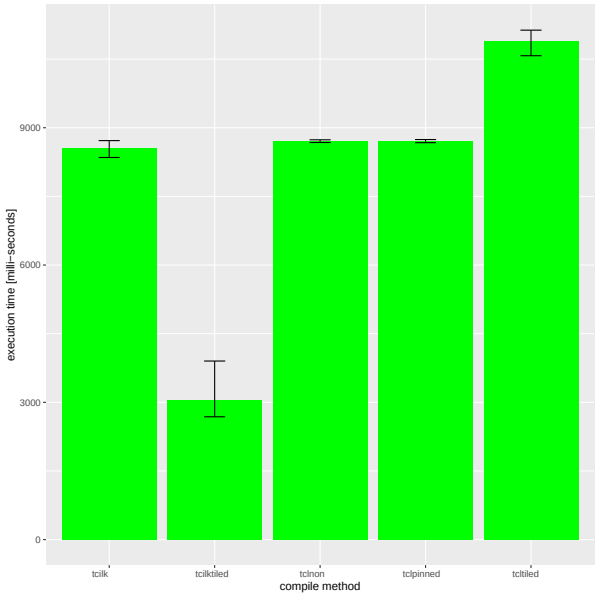

Figure 18: transposed 7000 matrix parallel multiplies

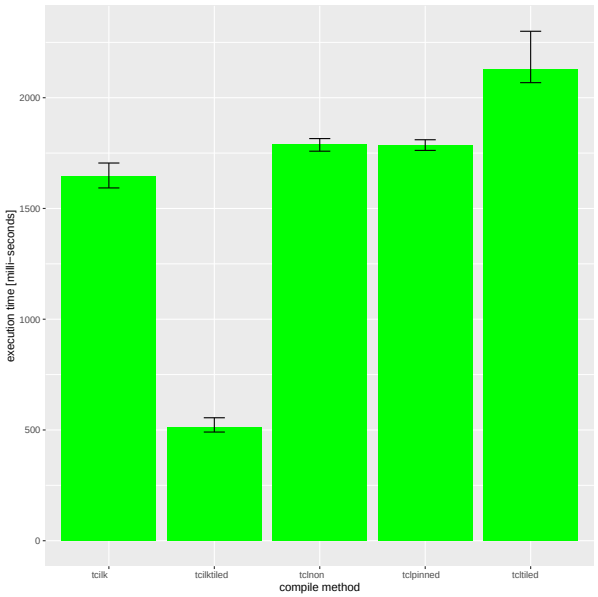

Figure 19: transposed 4000 matrix parallel multiplies

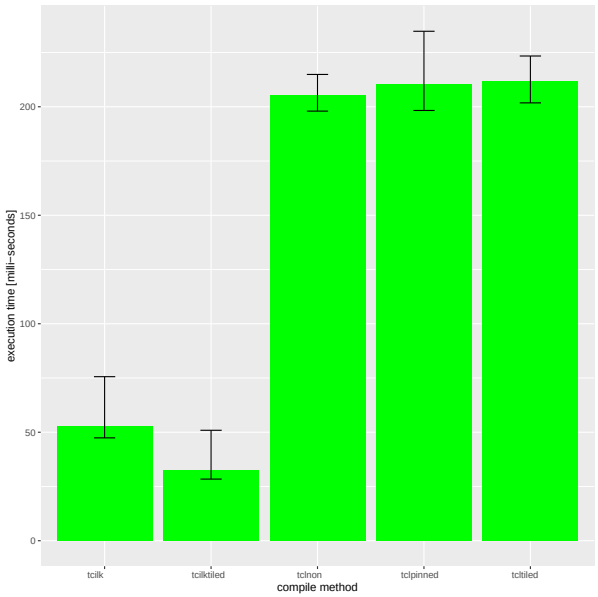

Figure 20: transposed 1000 matrix parallel multiplies

The data enabled an assessment of speed up obtainable using OpenCL and OpenCilk. Speed up is made with respect to a reference point. Table 7 and Table 8 show speed up calculated from the data for normal and transposed matrix multiplication, respectively. In all cases the speed up was calculated using the mean execution times from the data of Table 5 and Table 6. The two columns on the right of each table shows the speed up of tiled and normal OpenCL approach relative to serial matrix multiplication with the loops in `ijk` order. The OpenCL approaches used this loop ordering. The next two columns show the speed up using tiled and normal OpenCilk approach relative to optimized serial matrix multiplication with `ikj` loop ordering. The OpenCilk approaches used this loop ordering together

with optimization. The next two columns show speed up of tiled and normal OpenCilk approaches relative to the same serial matrix multiplication as the OpenCL approaches of the right two columns. The final column (second from the left) shows the speed up of serial matrix multiplication with and without optimization.

When speed ups were taken relative to the same reference point, in this case the serial matrix multiplication with `ijk` loop ordering, the two OpenCilk approaches performed the same if not better than the corresponding OpenCL approaches.

Table 7: Relative speed up observed in normal matrix multiplication

	reference/mean time [Relative Speed up]						
matrix dimension	refc ijk / refc ikj -O3	refc ijk/ rcilktiled	refc ijk/ rcilk	refc ikj -O3/ rcilktiled	refc ikj -O3/ rcilk	refc ijk rcltiled	refc ijk/ rclnon
10000	11	666	375	60	34	155	44
9000	7	468	253	63	34	104	42
8000	13	829	455	63	34	181	58
7000	6	398	216	63	34	88	69
6000	7	502	246	69	34	101	44
5000	6	420	212	65	33	87	92
4000	11	828	367	73	32	139	67
3000	6	434	192	68	30	73	94
2000	7	230	145	32	20	39	48
1000	7	73	61	11	9	11	11

Table 8: Relative speed up observed in transposed matrix multiplication

	reference/mean time [Relative Speed up]						
matrix dimension	tstdc ijk / tstdc ikj -O3	tstdc ijk/ tcilktiled	tstdc ijk/ tcilk	tstdc ikj -O3/ tcilktiled	tstdc ikj -O3/ tcilk	tstdc ijk tcltiled	tstdc ijk/ tclnon
10000	1	220	84	342	131	64	82
9000	1	225	83	189	70	65	82
8000	0	228	84	474	175	63	83
7000	1	237	84	168	60	66	83
6000	1	250	83	195	65	65	81
5000	2	249	83	157	52	65	80
4000	1	261	82	283	89	63	75
3000	2	254	80	126	40	57	65
2000	3	195	76	71	28	39	42
1000	3	64	39	21	13	10	10

A speed up of 56 for the multicore approach relative to the serial approach was expected, this corresponding to the number of cores used. Aligning the serial and multicore approaches through the relative speed up calculations in the two right hand column pairs of Table 7 and Table 8 were their purpose. From both tables, speed ups exceeding 56 were obtained. However, higher speed ups occurred above a matrix dimension of 2000 (generally). It was expected execution time, and thus loss of speed up, would be involved in setting up a multicore processing in each program. Maybe this accounts for the low speed ups in low matrix dimensions. It was thought due to the number of library functions used with OpenCL programs this set up loss would be greater than with OpenCilk, the programs of which had no library functions. Table 7 and Table 8 suggest both OpenCL and OpenCilk have set up cost, but they decrease in significance as the amount of multi-core process increases.

From Table 5 and Table 6 the minimum execution time in each matrix dimension was taken to form Table 9. The mean execution time is tabulated. In all case, the fastest method used tiling in OpenCilk. Also in all cases there was no significant difference between the execution times for normal or transpose matrix addressing.

Table 9: Fastest executing methods for normal and transposed matrix multiplication

	normal		transposed	
dimension	method	mean time	method	mean time
10000	rcilktiled	9634.2	tcilktiled	9587.1
9000	rcilktiled	6771.0	tcilktiled	6835.2
8000	rcilktiled	4791.1	tcilktiled	4732.2
7000	rcilktiled	3177.2	tcilktiled	3050.3
6000	rcilktiled	1816.2	tcilktiled	1815.7
5000	rcilktiled	1095.5	tcilktiled	1051.8
4000	rcilktiled	490.8	tcilktiled	514.3
3000	rcilktiled	214.7	tcilktiled	221.9
2000	rcilktiled	83.9	tcilktiled	83.3
1000	rcilktiled	32.0	tcilktiled	32.4

Table 10 contains from Table 5 and Table 6 the serial approaches of each matrix dimension which gave the minimum executing time. For normal matrix address approach the `ikj` loop ordering and optimization gave the minimum execution time. This was also the case with matrix dimensions 1000 and 2000 when transposed matrix addressing was used. In the higher than 2000 matrix dimensions, the loop ordering `ijk` gave the least execution time for transposed matrix addressing.

Table 10: Fastest standard matrix multiplication methods

	normal		transposed	
dimension	method	mean time	method	mean time
10000	refc ikj -O3	580625.3	tstdc ijk -O3	976547.0
9000	refc ikj -O3	429753.7	tstdc ijk -O3	712722.2
8000	refc ikj -O3	300284.3	tstdc ijk -O3	500071.2
7000	refc ikj -O3	200120.0	tstdc ijk -O3	333871.4
6000	refc ikj -O3	124427.3	tstdc ijk -O3	210303.3
5000	refc ikj -O3	71091.2	tstdc ijk -O3	121942.0
4000	refc ikj -O3	35671.5	tstdc ijk -O3	62361.8
3000	refc ikj -O3	14608.9	tstdc ijk -O3	26489.2
2000	refc ikj -O3	2710.1	tstdc ikj -O3	5927.0
1000	refc ikj -O3	337.8	tstdc ikj -O3	664.5

References

Ansorge, R. (2022), *Programming in Paralllel with CUDA* (Cambrisge University Press).

Bourd, A. (2017), "The OpenCL Specification", version 2.2, Khronos OpenCL Working Group (), `http://khronous.org/registry/OpenCL/specs/opencl-2.2.pdf`, accessed June 2, 2017.

Curtis, D., et al. (2020), "Research anf Teaching with OpenCilk", ACM Symposium on Parallelism in Algorithms and Architectures (), `http://cilk.mit.edu/beta2/opencilk.spaa.2020.pdf`, accessed Nov. 17, 2020.

Leiserson, C. E. (2018), "6.172 Performance Engineering of Software Systems, Lecture 1: Introduction and Matrix Multiplication", MIT Open Courseware, `ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/resources/mit6_172f18_lect1/`, accessed Dec. 28, 2022.

Munshi, A., et al. (2012), *OpenCL Programming Guide* (Addison-Wesley), 648 pp.

Turner, P. R., Arildsen, T., and Kavanagh, K. (2018), *Applied Scientific Computing with Python* (Springer International Publishing AG), 282 pp.