

# How to use OpenMP within TiViPE

Technical Report: Version 1.0.0



Tino Lourens
TiViPE
Kanaaldijk ZW 11
5706 LD Helmond
The Netherlands
tino@tivipe.com

September 5, 2011

# Contents

**Abstract**

The aim of TiViPE is to integrate different technologies in a seamless way using graphical icons [5]. Due to these icons the user does not need to have in depth knowledge of the underlying hardware architecture.

This report elaborates on the use of Open Multi Processing (OpenMP). OpenMP is driven by compiler directives in C, C++, or Fortran, hence should be used within the software libraries used for TiViPE rather than incorporation the code into the graphical icons directly.

# 1 Introduction

OpenMP is an application programming interface (API) that supports multi-platform *shared memory parallel applications in C/C++ and Fortran* on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior [3].

OpenMP is an implementation of multi-threading, a method of parallelization whereby the master ”thread” (a series of instructions executed consecutively) ”forks” a specified number of slave ”threads” and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors [3].

The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed. This directive always starts with `#pragma omp ...` where the dots denote optional directives. Each thread has an ”id” attached to it which can be obtained using a function (called `omp_get_thread_num()`). The thread id is an integer, and the master thread has an id of ”0”. After the execution of the parallelized code, the threads ”join” back into the master thread, which continues onward to the end of the program.[3].

This report is outlined as follows: Section 2 provides a hello world example using OpenMP. The power of OpenMP is that it is compact, Section 3 illustrates that the API (version 2.0) contains 21 routine calls only. Section 4 provides an example on matrix multiplication. The reports finishes with a summary and conclusions.

# 2 OpenMP hello world

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The number of threads can be assigned by the runtime environment based on environment variables or in code using functions. The OpenMP functions are included in a header file labeled ”omp.h” in C/C++.

Figure 1 illustrates a hello world example. The first part determines the default number of threads used on your machine. Typically this number is between 1 and 20 depending on the type of processor. Note that future cpu generations likely yield more cores. This default number can be modified by setting `OMP_NUM_THREADS`.

Little additional effort is required to include OpenMP directives into the normal C++ code. The compiler takes care of parallelization of the code by means of these directives, making it a powerful and easy to use API. The execution model is a master thread that forks a set of worker threads to yield a parallel region, at the end synchronization takes places and the threads join to become a single master thread again. In the example of Figure 1 two parallel regions have been provided by pragma statements.

OpenMP makes use of threads that have access to the same, globally shared, memory. All threads have access to this shared memory. Data can be made private, in that case every thread has a copy of the data. No other thread can access this data and changes are only visible to the thread owning the private data. Hence making the loop variables private is a logical choice. It implies that every thread can modify this variable to the desired index at its own speed.

## 2.1 Compiler directives

A pragma directive is defined as

```
#pragma omp directive [[clause [clause] ...]
```

```
#include <omp.h>
#include <stdio.h>

void main ()
{
  // 1.  Determine useful number of threads
  int thid, nthreads;
#pragma omp parallel private(thid)
  {
    thid = omp_get_thread_num();
    // executed by master/first thread only
    if (thid == 0) // identical to #pragma omp master
      {
nthreads = omp_get_num_threads();
printf ("There are by default %d threads on your machine.\n",
nthreads);
fflush (stdout);
      }
  }
  // 2.  Execute using a fixed number of threads
  int n = (nthreads / 100) + 5; // = 5 forall cpu's
#pragma omp parallel private(thid) num_threads (n)
  {
    thid = omp_get_thread_num();
    // executed by master thread only
#pragma omp master
    {
      int nthreads = omp_get_num_threads(); // equal to n
      printf ("There are %d threads.\n", nthreads);
      fflush (stdout);
    }
    // synchronize: wait until all threads are finished
#pragma omp barrier
    // executed by all threads
    printf ("Hello World from thread %d\n", thid);
    fflush (stdout);
  }
}
```

**Figure 1:** *OpenMP hello world example:* `main.cpp`.

```
TEMPLATE        = app
QT             += gui network opengl multimedia
CONFIG         += qt thread release
QMAKE_CXXFLAGS += -openmp
win32 {
    CONFIG     += console
}
SOURCES         = main.cpp
OBJECTS_DIR     = obj
TARGET          = mpTest
DESTDIR         = bin
```

**Figure 2:** *Qmake project file:* `src.pro`.

A \ is used to continue a statement over multiple lines.
These directives are

```
parallel
for
barrier
sections
section
single
master
critical
atomic
ordered
flush [(list)]
schedule (static | dynamic | guided | auto [, chunk])
schedule (runtime)
```

The available clauses are

```
if (scalar expression)
private (list)
shared (list)
firstprivate (list)
lastprivate (list)
default (none|shared)}
nowait
num_threads (scalar integer expression)
reduction (operator : list)
```

Details on compiler directives can be found at [4].

## 2.2 Project file

Figure 2 provides the project file. The line `QMAKE_CXXFLAGS += -openmp` should be included to process all given pragma statements, see also the hello world example of Figure 1. When omitted all pragmas are ignored, resulting in a warning that variable `n` is not used.

### 2.3 Setup and compilation of hello world example

The gnu gcc compiler by default supports OpenMP. The code provided in `main.cpp` and `src.pro` as illustrated in Figures 1 and 2, respectively should stored in the same directory. Being in this directory the code should compile to `bin/mpTest` when the following is done (assuming that *gcc* and *qt* have been installed properly):

```
qmake src.pro
make
```

Under Microsoft windows, one can use the visual studio compiler by calling `nmake`. The Microsoft compiler Visual Studio express edition 2008, is found at `http://www.microsoft.com/express/Downloads/#2008-Visual-CPP`. Execute file `vcsetup.exe` to install this compiler. By default the express edition does not contain OpenMP, for instance the omp.h file is missing in `%SDK_HOME%\VC\ include` directory.

Installing Microsoft Windows SDK for Windows 7 and .NET Framework 3.5 SP1 `winsdk_web.exe` is therefore required. It is found at `http://www.microsoft.com/downloads/en/details.aspx?FamilyID=c17ba869-9671-4330-a63e-1fd44e0e2505&displaylang=en`

After both packages are installed, a similar action needs to be performed (compared to the gnu compiler):

```
qmake src.pro
nmake
```

The compiled executable `bin/mpTest.exe` contains 3 main libraries:

```
c:\windows\winsxs\x86_microsoft.vc90.crt_1fc8b3b9a1e18e3b_9.0.30729.
  5570_x-ww_0517bbc6\MSVCR90.DLL
c:\windows\winsxs\x86_microsoft.vc90.openmp_1fc8b3b9a1e18e3b_9.0.
  30729.5570_x-ww_214ee422\VCOMP90.DLL
c:\windows\system32\KERNEL32.DLL
```

A dependency to the visual studio omp library (vcomp90.dll) has been created.

When this file is executed it should provide something like:

```
There are by default 2 threads on your machine.
There are 5 threads
Hello World from thread 0
Hello World from thread 2
Hello World from thread 1
Hello World from thread 3
Hello World from thread 4
```

## 3 OpenMP available commands

The header file (omp.h) contains a compact set of routine calls:

```
void   omp_set_num_threads   (int _Num_threads);
int    omp_get_num_threads   (void);
int    omp_get_max_threads   (void);
int    omp_get_thread_num    (void);
```

```
int    omp_get_num_procs      (void);
void   omp_set_dynamic        (int _Dynamic_threads);
int    omp_get_dynamic        (void);
int    omp_in_parallel        (void);
void   omp_set_nested         (int _Nested);
int    omp_get_nested         (void);
void   omp_init_lock          (omp_lock_t * _Lock);
void   omp_destroy_lock       (omp_lock_t * _Lock);
void   omp_set_lock           (omp_lock_t * _Lock);
void   omp_unset_lock         (omp_lock_t * _Lock);
int    omp_test_lock          (omp_lock_t * _Lock);
void   omp_init_nest_lock     (omp_nest_lock_t * _Lock);
void   omp_destroy_nest_lock (omp_nest_lock_t * _Lock);
void   omp_set_nest_lock      (omp_nest_lock_t * _Lock);
void   omp_unset_nest_lock    (omp_nest_lock_t * _Lock);
int    omp_test_nest_lock     (omp_nest_lock_t * _Lock);
double omp_get_wtime          (void);
double omp_get_wtick          (void);
```

# 4 Application

## 4.1 Matrix multiplication

For GP-GPUs matrix multiplication has been used as an example to demonstrate that this algorithm can be processed in parallel and has been given as an example by Nvidia for both OpenCL and CUDA, see OpenCL section 2.5 [2], and for CUDA section 3.2.3 [1].

Matrix multiplication is as follows

$$C_{i,j} = \sum_{k=1}^{p} A_{i,k} B_{k,j} \tag{1}$$

for all $1 \leq i \leq m$ and $1 \leq j \leq n$, where $i$ and $j$ denote the row and column index, respectively.

### 4.1.1 Matrix dimensions

In the implementation the matrices are square and have the size of 768x768 elements in the example below:

```
#define BLOCK_SIZE 64

#define NN 12
#define WA (NN * BLOCK_SIZE)
#define HA (NN * BLOCK_SIZE)
#define WB (NN * BLOCK_SIZE)
#define HB WA
#define WC WB
#define HC HA

#include <omp.h>
```

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <assert.h>

double As[BLOCK_SIZE][BLOCK_SIZE];
double Bs[BLOCK_SIZE][BLOCK_SIZE];
```

By modifying `BLOCK_SIZE` and `NN` one can tune to a good performance making (optimal) use of the available cache memory.

### 4.1.2 CPU implementation

A straight forward implementation of matrix multiplication is given below

```
void ParMatMul (double *C,
                double *A,
                double *B,
                int    hA,
                int    wA,
                int    wB,
                int    nthreads)
{
  double sum;
  int i, j, k;
  double a, b;
#pragma omp parallel default (none) num_threads (nthreads) \
  shared (A, B, C, hA, wB, wA) private (i, j, k, sum, a, b)
  {
#pragma omp for schedule (static) nowait
    for (i = 0; i < hA; i++)
      for (j = 0; j < wB; j++)
        {
          sum = 0.0;
          for (k = 0; k < wA; k++)
            {
              a    = A[i * wA + k];
              b    = B[k * wB + j];
              sum += a * b;
            }
          C[i * wB + j] = sum;
        }
  } // end omp parallel
}
```

### 4.1.3 CPU block implementation

An alternative way of matrix multiplication is to keep the amount of cache memory in mind. Like this is done in the GPU examples. This implementation divides the matrices into blocks and processes them block wise. It is implemented as follows:

```
void ParBckMatMul (double *C,
                   double *A,
                   double *B,
                   int    hA,
                   int    wA,
                   int    wB,
                   int    nthreads)
{
  int bx, by;
  int tx, ty;
  int aBegin, aEnd, aStep, bBegin, bStep;
  int a, b, c, k;
  double Asub, Bsub, Csub;
  int size = hA * wB;
  memset (C, 0, size * sizeof (double));

  int hA_grid = hA / BLOCK_SIZE; // hA % BLOCK_SIZE == 0
  int wB_grid = wB / BLOCK_SIZE; // wB % BLOCK_SIZE == 0

  for (by = 0; by < hA_grid; by++)
    for (bx = 0; bx < wB_grid; bx++)
      {
        aBegin = wA * BLOCK_SIZE * by;
        aEnd   = aBegin + wA - 1;
        aStep  = BLOCK_SIZE;
        bBegin = BLOCK_SIZE * bx;
        bStep  = BLOCK_SIZE * wB;
        for (a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
          {
#pragma omp parallel for default (none) num_threads (nthreads) \
  shared (A, B, As, Bs, a, b, wA, wB) private (ty, tx) \
  schedule (static)
            for (ty = 0; ty < BLOCK_SIZE; ty++)
              {
                for (tx = 0; tx < BLOCK_SIZE; tx++)
                  {
                    As[ty][tx] = A[a + wA * ty + tx];
                    Bs[ty][tx] = B[b + wB * ty + tx];
                  }
              }
#pragma omp parallel for default (none) num_threads (nthreads) \
  shared (As, Bs, C, bx, by, wB) private (ty, tx, k, c, Asub, Bsub, Csub) \
  schedule (static)
```

```
            for (ty = 0; ty < BLOCK_SIZE; ty++)
              {
                for (tx = 0; tx < BLOCK_SIZE; tx++)
                  {
                    Csub = 0.0;
                    for (k = 0; k < BLOCK_SIZE; k++)
                      {
                        Asub  = As[ty][k];
                        Bsub  = Bs[k][tx];
                        Csub += Asub * Bsub;
                      }
                    c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
                    C[c + wB * ty + tx] += Csub;
                  }
              }
          }
      }
}
```

### 4.1.4 Filling matrices with random values

A routine for random initialization of a matrix is as follows

```
void randomInit (double *a,
                 int    size)
{
  for (int i = 0; i < size; i++)
    a[i] = (rand() % 999) / 1000.0; // 0..1 with steps of 0.001
}
```

### 4.1.5 Main program

The main routine is as follows

```
void main ()
{
  int nthreads = 1;
  printf("Enter number of threads (1-16): ");
  scanf("%d", &nthreads);

  assert (nthreads >= 1 && nthreads <= 16);

  unsigned int total_size = 0;
  srand ((int) time (NULL));

  unsigned int size_A = WA * HA;
  unsigned int mem_size_A = sizeof (double) * size_A;
  double *h_A = (double *) malloc (mem_size_A);
  assert (h_A);
```

```
  total_size += mem_size_A;

  unsigned int size_B = WB * HB;
  unsigned int mem_size_B = sizeof (double) * size_B;
  double *h_B = (double *) malloc (mem_size_B);
  assert (h_B);
  total_size += mem_size_B;

  randomInit (h_A, size_A);
  randomInit (h_B, size_B);

  unsigned int size_C = WC * HC;
  unsigned int mem_size_C = sizeof (double) * size_C;
  double *h_C = (double *) malloc (mem_size_C);
  assert (h_C);
  total_size += mem_size_C;

  double tstart = omp_get_wtime();
  ParBckMatMul (h_C, h_A, h_B, HA, WA, WB, nthreads);
  //ParMatMul (h_C, h_A, h_B, HA, WA, WB, nthreads);
  double tend = omp_get_wtime();

  double t = tend - tstart;
  printf ("threads = %d, matMul cost %8.3f (s)\n", nthreads, t);
  printf ("size(A) = (%d, %d)\n", HA, WA);
  printf ("size(B) = (%d, %d)\n", HB, WB);
  printf ("Total memory size = %6.3f (MB)\n", total_size / 1048576.0);
  printf ("Total memory size = %6.3f (MB)\n", total_size / 1048576.0);
  printf ("GFLOPS/s = %6.3f \n", (2.0 * HA) * WA * WB /
  (1000000000.0 * t));

  free (h_A);
  free (h_B);
  free (h_C);
}
```

### 4.1.6 Timings

Figure 3 illustrates the timing results for different matrix sizes. The implementation of matrix multiplication is compact and straight forward, but when looking at the performance of multiplication on a single thread, there are considerable differences, as illustrated in the left part of this Figure 3. The performance might differ more than 4.5 times for the Intel i5 460m and up to 16 times for the AMD Athlon 64 X2, just depending on the size of the matrices.

The AMD processor performs best between square matrix sizes between 32 and 96 for a single thread and between 64 and 96 for multiple threads. Its is clear that there is a match with the amount of memory required for the matrices and the cache size of the chip. The Athlon processor has 2x128KB L1 and 2x512KB L2 cache. Assuming that matrices A and B are kept in the cache and both are of type double it implies NxNx8x2/1024 (KB) of memory are required for these two matrices. For N = 90 and

180 these cache limits are reached, explaining a dip at 128 and to a lesser extend at 512.

Using multiple threads makes sense for matrices larger than 64x64. In that case the overhead of using threads versus not using threads pays off. The computational time on the AMD processor in that case is around 0.5 (ms).

For the Intel i5 460m and i7 2600k a similar performance curve can be observed. The i5 processor has 2 cores and 2x64KB, 2x256KB, and 3MB, L1, L2, and L3 caches, respectively. Hence we observe the good performance at a wider range due to the L3 cache. However, when matrices of 1024x1024 or larger are used this processor also shows a considerable dip in performance. The Intel i7 2600k desktop processor (with 4 cores) has similar L1 and L2 cache sizes compared to the i5 460m, but differes in its L3 cache: L1 D 4x32KB L1 I 4x32KB L2 4x256KB L3 8MB cores. The performance of this chip is better compared to the i5, but its performacne curves are similar to the i5.

The problem can be alleviated if the matrices are subdivided into blocks, and the blocks are first copied into data blocks that fit within the caches of the processor. Figure 4 shows the performance for matrices of at least 1024x1024 in size, for different block sizes.

Good performance is achieved for block sizes between 48 and 112, and 48 and 120, for AMD Athlon 64 X2 and Intel i5 460m/i7 2600k, respectively. The two matrices A and B require for 48, 112, and 120, 36, 192, and 225 KByte of memory. The peak performance is at a block size of 80 and 88, indicating that the best performance is reached at 100KB to 121KB of memory.

Figure 4 clearly demonstrates that the caches need to be filled in order to receive a good speed up using multiple threads. OpenMP demonstrates that use of multiple threads is always beneficial if the blocks used are 32x32 in size or larger. Using blocks speeds up the code 30 (AMD) and 15 (Intel) times compared to the standard solution.

It is interesting to see that the use of 4 threads on the Intel i5 (a dual core with four threads) gives a 10% performance increase when 4 threads are used, reaching the maximum performance of 4 Gflops/s for matrix multiplication. The same holds for the i7 2600k for 4 cores with 8 threads.

The AMD dual core procduces a 3.5 times speed up when using 4 threads compared to a single thread. This can be only due to a redivision of data that might fit better into the cache memory of the chip.

# 5 Summary and conclusions

OpenMP is a powerful API, by adding pragma statements to the code, the compiler takes care of dividing the code over the worker threads, and one can expect a speedup up the number of cores in a CPU. For the AMD Athlon 64 X2 and Intel i5 460m, these speed up where (as expected) around 2 times. This holds especially when the processors can compute independently.

For the moment most code optimization are not achieved by the multiple cores, but by properly tuning the data sets. Fitting data sets within the caches might give an increase of performance of up to 20 times. This is still considerably more than just parallelizing the code using multiple cores, especially when there are only 2 or 4.

The latest developments of CPU hardware however show a growing number of cores, hence optimizing by parallelism in the near future might become very attractive, and OpenMP will play a key role in improving speeding up applications.

# References

[1] http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/cuda_c_programming_guide.pdf.

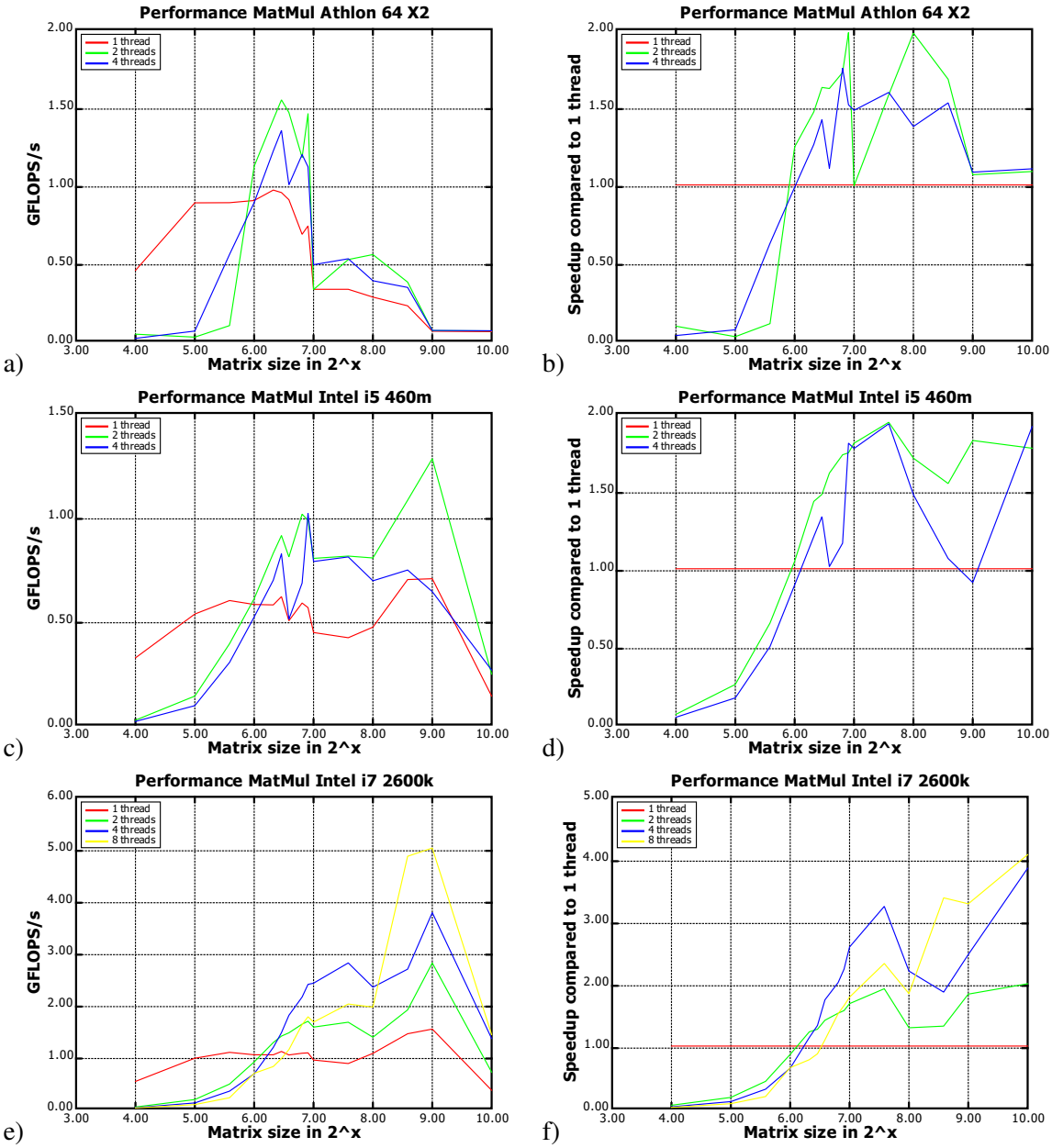


**Figure 3:** *Timings of matrix multiplication measured in GFLOPS/s in the left column and relative speed up compared to a single thread in the right column, for AMD Athlon 64 X2, Intel i5 460m, and Intel i7 2600k, respectively top, middle, and bottom row.*

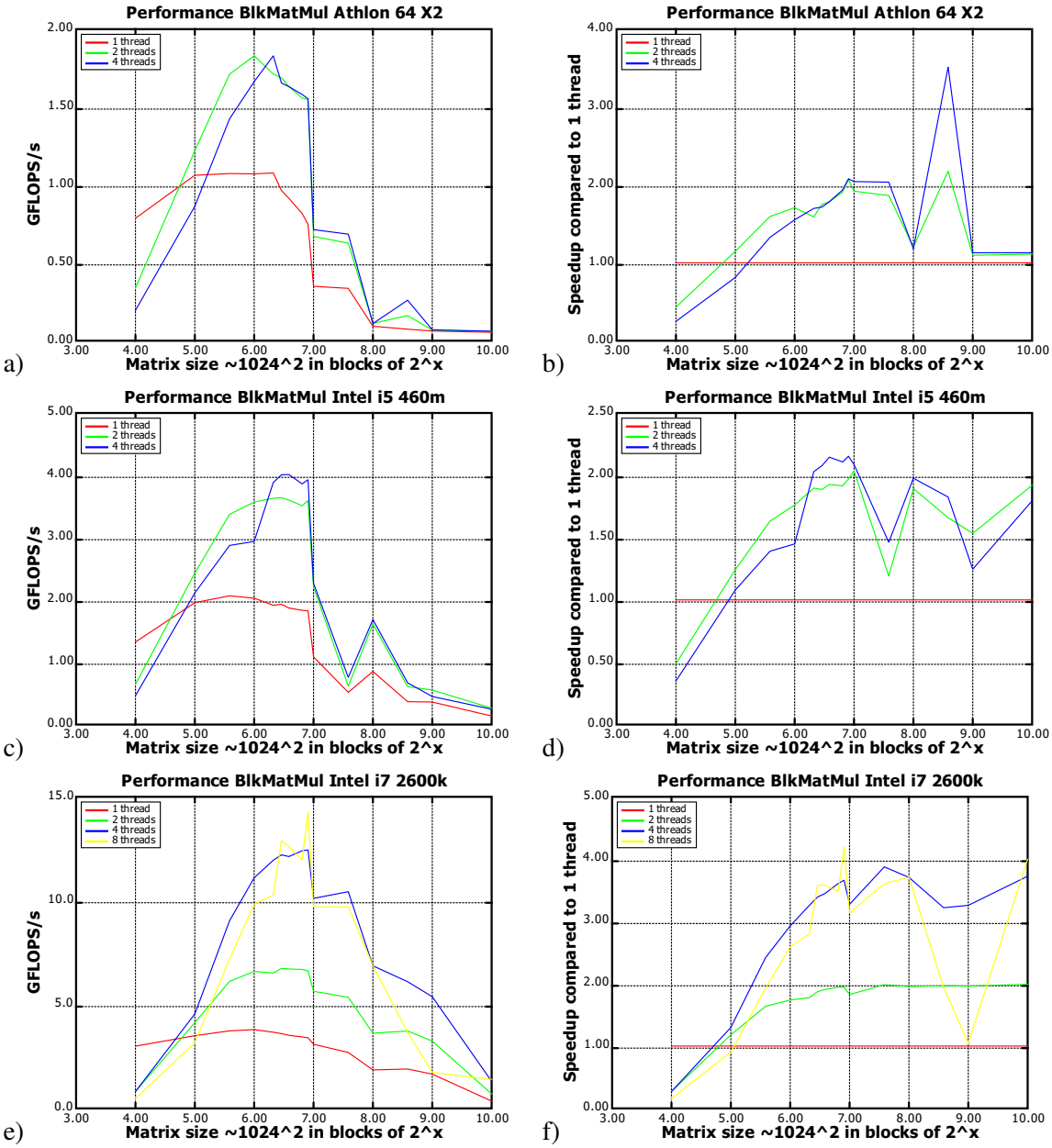


**Figure 4:** *Timings of matrix multiplication* using blocks *measured in GFLOPS/s in the left column and relative speed up compared to a single thread in the right column, for AMD Athlon 64 X2, Intel i5 460m, and Intel i7 2600k, respectively top, middle, and bottom row.*

[2] http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/opencl_programming_guide.pdf.

[3] http://en.wikipedia.org/wiki/openmp.

[4] http://openmp.org/wp/openmp-documents/cspec20.pdf.

[5] T. Lourens. Tivipe –tino's visual programming environment. In *The 28$^{th}$ Annual International Computer Software & Applications Conference, IEEE COMPSAC 2004*, pages 10–15, 2004.