Pure Parallel Computing on MacPros

Ross Maloney

December 10, 2020
(typo & edit correction)

Contents

Chapter 1

Standard way: OpenMP

1.1 Introduction

Moore's Law states the number of transistors on a semi-conductor chip will double every 18 months. Increased transistors in a computer chip means greater functionality can be produced. The first computer chips had a 8-bit word length, then 16, then 32, and now 64-bits is common. Each of these increases increased the speed of the computation the chip can handled, but required more transistors to implement the associated logic elements. However, as this occurs the size of the transistors and their inter-connections decrease in size, resulting in increase electrical resistance and thus the power to operate the chip. The alternate approach to increasing the speed is to increase the clock frequency used to drive the chip. This also increases the power dissipated in the chip. Continuing either of these two paths results in the chip becoming too hot to cool by air.

Another approach to increasing the speed of computation is to divide the computation into parts and perform those parts simultaneously. This is parallel computing. But only some problems can be speed-up by this process. Processing of vectors is one such applicable problem it having given rise to the Cray-1 computer described in **russell1978**. This approach requires two or more processors. Each core of a multi-core is an independent processor each of which has shared access to the system's memory. The first of such multi-core chips was brought to market by IBM in 2001. A multi-core has a lower clock frequency than a single processor together will a lower transistor density of the chip.

Open Multi-Processor, or OpenMP, is a standard to facilitate the writing of parallel computing programs which would be portable across computer systems employing Symmetric Multiple Processors (SMP). It's use is an example of *thread programming* although threads are not always explicitly referenced. Version 1.0 of the OpenMP specifications is tabulated in **urbanic2015** as having been released in 1997. The parallel processing units in SMP are part of the CPU. This is the situation of multi-core processors. Although OpenMP has expanded in scope, multi-cores can be regarded as it's roots.

This work builds upon the approach taken in **fu2016** which uses OpenACC which is another parallel computing standard aim, at least initially, using attached GPUs. The aim of all parallel computing is to increase the speed of execution of each complete piece of C code. In this work the original code is executed in a serial manner. By introducing different OpenMP constructs into that code, parallel execution is obtained. Which of those constructs have the most effect? How much of an effect? Three different C compilers and their associated libraries are used.

1.2 Tools to be used

Three C compilers and their associated libraries are used:

- gcc 6.3.0

- gcc 8.2.0
- PGI Comumity compiler release 18.10

In the case of the two gcc compilers, the command line used to compile and link each OpenMP code was:

```
gcc -fopenmp -O3  -o example example.c
```

and with the PGI compiler:

```
pgcc -mp -fast -o example example.c
```

In the case of the non-OpenMP (serial) code, the command lines used were:

```
gcc -O3 -lgomp -o example example.c
```

and

```
pgcc -o example example.c
```

respectively. The same source file was used in each case. Each of the three compilers support all of OpenMP specifications up to 4.5. Each executable was run 25 times and the mean of the resulting execution times was calculated. The maximum and minimum execution value was also recorded. The system command `ls -l example` was used to obtain the size of the executable produced.

In all examples, execution timing was obtained via the `omp_get_wtime()` function which is part of the OpenMP library and specification. An outline is in Figure 1.1. A time measurement was made before and after the code to be measured. The difference between these two measures, expressed in milli-seconds, was taken as the execution tome of the embedded code.

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

int main()
{
  double ticks;

  ticks = omp_get_wtime();

  code being times goes here

  ticks = omp_get_wtime() - ticks;
  printf("Elapse time: %f millisec\n", 1000.0*ticks);
  return 0;
}
```

Figure 1.1: Code to time program execution

Timing using the standard `clock()` function with the `time.h` header file was tried in comparison to the timing technique of Figure 1.1. When both were used to time sequential and single thread program executions similar results were obtained. However with more than one thread, using `clock()` gave significant longer, and erroneous, execution times than obtained using `omp_get_wtime()`. This could be due to the turbo-boost function of the processor and the difference between the time reference `clock()` and `omp_get_wtime()` use.

All compiling, linking and execution was performed on an Apple MacPro 2013 model running Debian Linux 9.7 with DWM 6.1 as it's window manager. By using the `dmidecode -t 4` system program, the MacPro was measured as having 6 core Intel Xeon E5-1650 v2 processor running at 3.5 GHz.

From this it followed, the processor had SSE, SSE2, and AVX streaming instructions available, operating on a 256-bit register in each core. Thus each core could process 4 64-bit values per clock cycle, or 8 32-bit values per clock cycle. Each core had 2 threads, thus 12 threads were available for use. The `dmidecode -t 17` system program indicated 16 GB pf DDR3 memory was install in 4 lots of 4 GB each.

Table 1.1: OpenMP constructs considered

	Construct	Example
Directive	parallel	#pragma omp parallel
	single	#pragma omp serial
	sections	#pragma omp sections
	for	#pragma omp for
	simd	#pragma omp simd
	barrier	#pragma omp barrier
	task	#pragma omp task
	declare simd	#pragma omp declare simd
Clause	aligned	aligned(x, y)
	collapse	collapse(2)
	firstprivate	firstprivate(c)
	lastprivate	lastprivate
	nowait	nowait
	private	private(c, i)
	reduction	reduction(+: sum)
	sefelen	safelen(3)
	schedule	schedule
	shared	shared
	simdlen	simdlen(32)
	uniform	uniform(c)
Library	tasks	num_tasks()
	teams	num_teams()
	threads	num_threads(10)

Table 1.1 collects together the OpenMP constructs which are used in the following examples. This table does not show all constructs which are available in OpenMP specification 4.5. Specification 4.5 is used as the reference for it was the latest supported by the available compilers.

The examples chosen had to have sufficient computing involved to enable execution time to indicate the affect of parallel computing. All constructs shown in Table 1.1 are not applicable to every parallel computing situation so different types of examples were used. This selection also indicates the computing situations in which parallel processing is applicable. The results obtained give an indication of the consequence which might be obtained in using parallel computing via OpenMP in real-world problems.

1.3 The most elementary parallel computing construct

Parallel programs operate on the *together-divide-together* principle. By this is meant the program executes as a single execution stream, then divides into separate (parallel) execution streams, and then returns to a single execution stream. This process can repeated one, or many times, in the one parallel execution program. In every well constructed parallel program it starts with a single execution stream and ends with a single execution stream.

In all parallel processing the fundamental element is a *thread*. All processing is done in one or more threads. The constructs contained in Table 1.1 are linked to creating and handling of threads. This remains true if the OpenMP constructs of Table 1.1 are replace by OpenACC constructs. Accelerators which were the initial focus of OpenACC also operate via threads in a related manner to threads in multi-core CPUs.

In a parallel program constructed using OpenMP, each parallel execution part is contained within

a clause proceeded by a `#pragma omp parallel` statement. The code in Figure 1.2 is an example. The first executable statement is the `At the start` print statement. The is executed once, i.e. it is a non-parallel executed statement, or a statement executed by a single thread.. The final `At the end` print statement is also a single thread executed statement. The parallel portion follows the `#pragma omp parallel` line and consists of the `Hello, second`, and `third` print statements all of which are contained between the { and } brackets. These brackets are required; they are part of the OpenMP syntax for the `#pragma omp parallel` construct. Two or more threads execute these statements. Each thread executes all these statements and any logic which is contained within those statements. Those threads execute simultaneously. It is this simultaneous execution of threads which gives rise to the parallel execution of these executable statements.

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

int main()
{
  printf("At the start\n");
  #pragma omp parallel
  {
    printf("Hello\n");
    printf("   second\n");
    printf("      third\n");
  }
  printf("At the end\n");
  return (0);
}
```

Figure 1.2: A very simple parallel program

One execution run of the code of Figure 1.2 when the computer had 12 threads available produced this output:

```
At the start
Hello
   second
      third
Hello
   second
      third
Hello
   second
Hello
      third
   second
      third
Hello
Hello
   second
   second
      third
      third
Hello
Hello
   second
Hello
Hello
   second
      third
Hello
```

```
   second
      third
   second
      third
   second
      third
      third
Hello
   second
      third
At the end
```

Note the `At the start` and `At the end` lines appear as the first and final lines of the output, respectively. There are 12 occurrences of `Hello, second` and `third` lines. This corresponds to each of the 12 available threads executing the three statements in the parallel clause of the program. However, `Hello, second` and third do not always occur together. This is due to parallel executing threads taking different times to execute the code and then compete to output their string to the common output.

The number of threads available for the parallel construct can be reduced. One means of doing this is by using a `num_threads()` clause in the `#pragma omp parallel` construct. If 7 threads were to be used, the `#pragma omp parallel` construct in the code of Figure 1.2 would be replace by `#pragma omp parallel num_threads(7)`. With this change made, output such as:

```
At the start
Hello
Hello
   second
   second
      third
      third
Hello
   second
      third
Hello
   second
      third
Hello
   second
      third
Hello
   second
      third
Hello
   second
      third
At the end
```

was obtained, again with 12 threads available but only 7 requested for use.

Two other alternatives are available for requesting the use of different number of threads. One alternative is to include the OpenMP runtime library call `omp_set_num_threads()` before the `#pragma omp parallel` construct. To set 7 threads for use until changed within the program, the function call would be `omp_set_num_threads(7);`. Within a OpenMP program, the runtime library function call `omp_get_num_thread()` returns the number of threads which have been set to be used. The other alternative is to set the environment variable `OMP_NUM_THREADS` before executing the OpenMP program. In the Bash shell, this environment variable would be set by a:

```
OMP_NUM_THREADS=7
```

command. The current value set for this variable in the Bash shell could be obtained via a:

```
echo $OMP_NUM_THREADS
```

command.

It is uncommon to execute an identical piece of code on each of the available cores of the CPU. This was the effect of the `#pragma omp parallel` directive alone, as in this Section. Instead, the `#pragma omp parallel` directive is used to denote a portion of code which is to be executed in parallel. Other directives are then inserted in the range of the `#pragma omp parallel` directive to define how the parallel execution is performed.

1.4 Execute slabs of code once but in order

The exact opposite to using the `#pragma omp parallel` alone is obtained by using a `#pragma omp single` primitive in the range of a parallel directive. In this situation the identified block of code is executed once, by a thread which the computer system selects. The whole of an identified block of code is executed by the selected thread. If there are more than one block of such designated statements, then those statements are executed one block after the other in the order in which they appear in the code. Each such block of statements is executed to completion before the next block in order commences. The code in Figure 1.3 is a simple example.

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

int main()
{
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("Message 1 from %d\n", omp_get_thread_num());
      printf("Message 2 from %d\n", omp_get_thread_num());
      printf("Message 3 from %d\n", omp_get_thread_num());
    }
    #pragma omp single
    {
      printf("Message 4 from %d\n", omp_get_thread_num());
      printf("Message 5 from %d\n", omp_get_thread_num());
    }
    #pragma omp single
    {
      printf("Message 6 from %d\n", omp_get_thread_num());
      printf("Message 7 from %d\n", omp_get_thread_num());
      printf("Message 8 from %d\n", omp_get_thread_num());
      printf("Message 9 from %d\n", omp_get_thread_num());
    }
  }
}
```

Figure 1.3: Three sements of code for executing once, but in order

In the code of Figure 1.3 three blocks of code are to be executed. For simplicity each block consists only of `printf()` statements as their executable content. The `printf()` statements print their number together with the number of the thread on which each statement was executed. This thread number is

obtained by the `omp_get_thread_num()` library function provided by OpenMP.

An example of the output produced by executing the code of Figure 1.3 on a 12 thread processor is:

```
Message 1 from 10
Message 2 from 10
Message 3 from 10
Message 4 from 4
Message 5 from 4
Message 6 from 7
Message 7 from 7
Message 8 from 7
Message 9 from 7
```

Notice the 3, 2, 4 grouping of the message lines which correspond to the 3, 2, 4 executable statements in the code of Figure 1.3. Each group of statements indicates the group was executed on the same thread. Different runs of this program would use different threads, but this collective behaviour remains. The same thread might be reused to execute another block of code at the computer's discretion.

The above output for this example was generated using the gcc compiler. If the pgcc compiler was used, only thread 0 was used since pgcc does not parallelize when there is a call (in this case `printf()`) present.

1.5 Parallel portions and conventional portions

All of a program may not require parallel execution, although some portions do. The non-parallel portions behave like a conventional program, i.e. execution is performed using a single thread. The code of Figure 1.3 shows this since thread execution using the specific example of a `#pragma omp single` directives withing a `#pragma omp parallel` directive. Of significance here is in the code of Figure 1.3 the range of the `#pragma omp parallel` directive covers the whole program.

An alternative manner of achieving the same result is shown in Figure 1.4. In that code two `#pragma omp parallel` directives are used each containing executable blocks of code. Although `printf()` calls were used there, any C statement, or combination of C statements, could be contained in those blocks. Those two blocks do not overlap. The code between the ranges of the two `#pragma omp parallel` directives is not executed in parallel, and therefore is executed as conventional C code.

One execution of the Figure 1.4 code gave the output:

```
Message 1 from thread 6
Message 2 from thread 6
Message 3 from thread 6
Message 4 from thread 2
Message 5 from thread 2
Message 6 from thread 0
Message 7 from thread 0
Message 8 from thread 0
Message 9 from thread 4
Message 10 from thread 4
Message 11 from thread 6
Message 12 from thread 6
```

The behaviour seen in this output is identical to the behaviour of the program of Figure 1.3. Notice here thread 6 is used twice, but this is not significant.

This *fall back to serial* principle apply to all code which is outside of the range of a `#pragma omp`

parallel directive. If the #pragma omp single directives in the example if Figure 1.4 were replace my any other (well most!) OpenMP directive, the principle remains.

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

int main()
{
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("Message 1 from thread %d\n", omp_get_thread_num());
      printf("Message 2 from thread %d\n", omp_get_thread_num());
      printf("Message 3 from thread %d\n", omp_get_thread_num());
    }
    #pragma omp single
    {
      printf("Message 4 from thread %d\n", omp_get_thread_num());
      printf("Message 5 from thread %d\n", omp_get_thread_num());
    }
  }
      printf("Message 6 from thread %d\n", omp_get_thread_num());
      printf("Message 7 from thread %d\n", omp_get_thread_num());
      printf("Message 8 from thread %d\n", omp_get_thread_num());
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("Message 9 from thread %d\n", omp_get_thread_num());
      printf("Message 10 from thread %d\n", omp_get_thread_num());
    }
    #pragma omp single
    {
      printf("Message 11 from thread %d\n", omp_get_thread_num());
      printf("Message 12 from thread %d\n", omp_get_thread_num());
    }
  }
}
```

Figure 1.4: Two parallel parts each executing once, and a conventional part

Which of these two alternative approaches is used is a matter of programming style. In both cases the executable code produced is the same.

The above output for this example was generated using the gcc compiler. If the pgcc compiler was used, only thread 0 was used since pgcc does not parallelize when there is a call (in this case printf()) present. As opposed to the gcc output, the pgcc generated output showed no sign of parallel execution.

1.6 Simultaneous execution of blocks of code

Parallel programs are at their best when blocks of code to be executed simultaneously. The #pragma omp parallel directive encloses such code in it's range. As shown in Section 1.4, the #pragma omp single directive counter-acts that directive to enable execution of code in a serial fashion. The opposite to this is the #pragma omp sections directive. The sections directive defines slabs of code which are to be executed in parallel.

The range of the #pragma omp sections covers the code to be executed in parallel. It is then

necessary to sub-divide this range into parts. Which parts are to be executed in parallel? Answering this is done using the `#pragma omp section` directive. So the `#pragma omp sections` directive within the range of a `#pragma omp parallel` directive gives the blocks of code which are to be executed in parallel as opposed to individual statements (as in Section 1.3). Then the `#pragma omp section` directive defines which pieces of code are to be excecuted in the context of the code withing the range of the `#pragma omp sections` scope.

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

int main()
{
  printf("Message 1 from thread %d\n", omp_get_thread_num());
  printf("Message 2 from thread %d\n", omp_get_thread_num());
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      {
        printf("Message 3 from thread %d\n", omp_get_thread_num());
        printf("Message 4 from thread %d\n", omp_get_thread_num());
        printf("Message 5 from thread %d\n", omp_get_thread_num());
      }
      #pragma omp section
      {
        printf("Message 6 from thread %d\n", omp_get_thread_num());
        printf("Message 7 from thread %d\n", omp_get_thread_num());
      }
      #pragma omp section
      {
        printf("Message 8 from thread %d\n", omp_get_thread_num());
        printf("Message 9 from thread %d\n", omp_get_thread_num());
        printf("Message 10 from thread %d\n", omp_get_thread_num());
      }
    }
  }
  printf("Message 11 from thread %d\n", omp_get_thread_num());
  printf("Message 12 from thread %d\n", omp_get_thread_num());
  printf("Message 13 from thread %d\n", omp_get_thread_num());
}
```

Figure 1.5: Mix of parallel and serial execuing code

Figure 1.5 is an example of a program with a portion of parallel code embedded in conventional serial code. The executables used are all `printf()` statements which access the thread number executing the statement. Each `printf()` output has a unique number. Statements 1 and 2 are in the first serial portion of the program while statements 11, 12, and 13 are in the final serial portion. Statements 3 through 10 are in the parallel computing part. Statements 3, 4, and 5 are in one block, statement 6 and 7 in another, and statements 8, 9, and 10 in the remaining block. Those three blocks are executed in parallel.

The thread number allocated to execute each block of code generally changes with each run of the program. The output produced by one run is:

```
Message 1 from thread 0
Message 2 from thread 0
Message 6 from thread 5
Message 8 from thread 8
Message 9 from thread 8
Message 10 from thread 8
```

```
Message 7 from thread 5
Message 3 from thread 4
Message 4 from thread 4
Message 5 from thread 4
Message 11 from thread 0
Message 12 from thread 0
Message 13 from thread 0
```

Here there are 13 lines of output produced by the 13 `printf()` statements of the program. The first two lines and the last three lines show execution of the two serial portions of the program. These both use thread 0. The remaining 8 lines, between those serial lines, show the parallel processing. Each of those parallel portions has a single thread assigned to execute a block. Thread 4 was allocated to execute the block containing statement 3, 4, and 5. Thread 5 was allocated to execute the block containing statements 6 and 7. Thread 8 was allocated to execute statements 8, 9, and 10. In all cases, one thread was allocated to execute one block of code.

Notice the order in the lines of output produced by the parallel executing statements. The line containing message 3 occurs before message 4, which occurs before message 5. The line containing message 6 occurs before message 6. The line containing message 8 occurs before the line containing 9, which occurs before message 10. This is in order of their occurrence in their respective blocks of code. Each block is executing according to the program. However, the appearance of message 6 before message 3 indicate parallel execution of the block containing those message producing `printf()` statements, and one is not dependent on the other. One block appears to be executing faster than others.

The above output for this example was generated using the gcc compiler. If the pgcc compiler was used, only thread 0 was used since pgcc does not parallelize when there is a call (in this case `printf()`) present. With no parallelization, the messages appear in numerical order as given in the code.

1.7 Parallel for loop

A common focus for introducing parallel computing is to loops. One reason is loops are responsible for a large portion of the computing time of a program in which they are contained. Being able to use loops is one reason for writing a program. OpenMP has the `#pragma omp for` directive embedded in the range of a `#pragma omp parallel` directive to handle loops.

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

int main()
{
  int i;

  omp_set_num_threads(10);
  #pragma omp parallel
    #pragma omp for
    for (i=0; i<7; i++) {
      printf("Message A%d  thread %d\n", i, omp_get_thread_num());
      printf("Message B%d  thread %d\n", i, omp_get_thread_num());
      printf("Message C%d  thread %d\n", i, omp_get_thread_num());
    }
}
```

Figure 1.6: Parallel execution of a for-loop

Processing is handled analogous to the `#pragma omp sections` directive.

The for-loop is decomposed by the compiler so a thread is assigned to each iteration. Each of those threads, and thus iterations of the for-loop, are processed simultaneously.

The number of threads available compared with those required influences the behaviour of the processing. This is particular the case here.

If the code of Figure 1.6 had 7 threads available, this number matching the iterations of the for loop, then and output such as:

```
Message A3   thread 3
Message B3   thread 3
Message C3   thread 3
Message A1   thread 1
Message A6   thread 6
Message B1   thread 1
Message C1   thread 1
Message B6   thread 6
Message C6   thread 6
Message A0   thread 0
Message A4   thread 4
Message B0   thread 0
Message C0   thread 0
Message B4   thread 4
Message C4   thread 4
Message A5   thread 5
Message B5   thread 5
Message C5   thread 5
Message A2   thread 2
Message B2   thread 2
Message C2   thread 2
```

was produced.

The above output for this example was generated using the gcc compiler. If the pgcc compiler was used with this code, only thread 0 would be used since pgcc does not parallelize when there is a call (in this case `printf()`) present.

Chapter 2

Simple OpenMP timed execution

In this work the assumption is made that the purpose of parallel computing is to reduce execution of the application code.

In this chapter the use of OpenMP directives and associated clauses are explored as a manner of achieving faster execution times. This was done by running a series of examples under different conditions and examining their individual and between example behaviours. The questions:

1. What reduction in execution time did increasing thread use have on each example?
2. Does increasing the number of threads improve the speedup?
3. Is their a "problem size affect" for application of OpenMP?
4. Can execution time reductions be explained?
5. Do all algorithms follow the same execution time reductions?
6. Does increasing the complexity of OpenMp directives have corresponding improvement in execution time reduction?
7. Which clauses when added to OpenMp directives most decrease execution times?

were used to guide this examination. The answers to these questions are interesting in their own right.

2.1 Method used for comparison

In the following examples comparison was by time of execution of the code. The data was obtained by compiling and running the resulting executable on a Mac Pro powered by a 6 core Intel Xeon E5 processor having a base clock speed of 3.5 GHz with 12 MB of L3 cache and a turbo boost to 3.9GHz. The memory was 16 GB of 1866 MHz DDR3 ECC memory divided into four 4 GB modules.

The operating system and environment used was the pre-release Debian 11 which included gcc-9. However, this compiler was found not to provide good support for OpenMP. So the source code distribution of gcc-9.2 was downloaded from `ftp://ftp.gnu.org` and compiled using the gcc-9 compiler from the Debian distribution. That compiling was done using the configuration statement:

```
configure -prefix=/usr --disable-multilib --enable-languages=c,c++,brig
```

All subsequent executables were generated using compilers and libraries which resulted. On the Debian 11 pre-release operating system the dwm 6.2 window manager under X Window was used. A split screen was employed. In one screen vim was running with the document open into which the results were typed after reach run completed. The other window was used to run the timing execution.

The program was edited for each change, then compiled. The resulting executable was run 25 times in succession.

The timing printed by the program was entered into the result program of Figure 2.1 which was executed by the free42 app running on a Iphone Xs. The result program was started by storing a value 0 into register 1 (which in the terminology used in Figure 2.1 is denoted by (01)) of the HP42S app. Each subsequent value to be processed by the result program would be entered into the HP42S app and then `XEQ "AA"` pressed. For example, if the value to be processed was 45.67, then the entry `45.67 XEQ` would be used using the `XEQ` the key on the top right hand corner of the HP42S app keyboard, then `AA` would be pressed when it appears under the `XEQ` word on the HP42S display. After each such entry, the running mean and the count of the numbers which have been processed is displayed on the HP42S app's display. After each entry is processed, register 02 of the HP42S app contains the minimum value, register 03 contains the maximum value, and register 05 contains the mean. Although they are not used, the program of Figure 2.1 also calculates the sum of the data values squared in register 06, which together with the data sum in register 04, could be used to calculate the data variance.

Line	Label	Program step	Explanation
1		LABEL "AA"	
2		RCL R[01]	R[0] $\rightarrow$ X; X $\rightarrow$ Y
3		X != 0	X not equal to 0
4		GTO 01	if true, skip
5		X<>Y	put Y register back into X
6		STO R[02]	store X into register 02 (min)
7		STO R[03]	store X into register 03 (max)
8		STO R[04]	store X into register 04 (sum)
9		X^2	square data value
10		STO R[06]	store X into register 06 (sum of sqrs)
11		1	put a 1 into X
12		STO R[01]	store 1 into register 01 (n)
13		GTO 09	exit
14	LBL 01		
15		X <> Y	exchange contents of X and Y
16		RCL R[02]	get min value from register 02
17		X <> Y	exchange contents of X and Y
18		X < Y	is min less than data value?
19		STO R[02]	yes, then replace min value
20		RCL R[03]	get max value from register 03
21		X <> Y	exchange contents of X and Y
22		X > Y	is max less than data value?
23		STO R[03]	yes, then replace max value
24		ENTER	copy X into Y
25		X^2	square the data value
26		RCL R[06]	get sum of squares
27		+	data squared to total
28		STO R[06]	store sum of squaares
29		X <> Y	exchange X and Y
30		RCL R[04]	get sum
31		+	add data value to sum
32		STO R[04]	store sum in register 04
33		1	put the value 1 into X
34		RCL R[01]	get count of data entries
35		+	add X and Y
36		STO R[01]	store the count of data entries
37		/	Y / X
38		STR R[05]	store new mean in register 05
39	LBL 09		
40		RCL R[01]	show number of data entries
41		END	

Figure 2.1: free42 program to calculate mean, maximum and minimum of numbers entered

Each timing result was entered into the HP42S app at the end of each execution of the program being timed. After all 25 timing measurements were obtained, the registers of the HP42S app were used to obtain the mean execution time, together with the maximum and minimum execution measured and these values recorded. The time displayed by the program was rounded manually to one decimal place before being entered in the HP42S app. No other user programs were active in the Mac Pro when the execution timings were being performed.

The time values were obtained using the `omp_get_wtime()` function which is standard in the OpenMP library. This time function was positioned in the code where timing was to start and finish thus removing parts of the program's execution not part of the timing. The difference between these two times was taken as the execution time of the code, recorded to one decimal point accuracy.

Sequential execution of the code was also performed. This is shown in the multi-thread results as a 0 thread. The code in this case was compiled without the OpenMP switch in the command line. The `omp_get_wtime()` function was linked in from the `gomp` library.

For a piece of code, the *speedup* is defined as the execution time of that code relative to the execution time of the sequential execution time of that code. It is calculated by dividing the mean execution time of the code by the mean execution time of the sequential version of the code.

Where as speedup is a measure of change in execution time achieved, there is a theoretical value speedup which might be available. If two threads are used in place of one, the execution speed might be expected to be halved. If three threads were used, then the execution speed could be expected to be a third of the one thread execution speed. This theoretical speedup here is calculated by dividing the mean sequential execution time of the code by the number of threads being used to execute the code. This provides a theoretical measure against which the execution speed could be measured.

2.2 Using parallel and for directives alone

Four programs are used: one finds prime numbers within a given range of values, and the other three each multiply two square matrices together. In each of those three programs the matrices are stored in different ways in the computer's memory. In each case, the code was changed (by changing a single constant) and re-compiled to present a problem *size*, and each piece of code was executed with a number of sizes. The number of threads to be used was entered as a parameter on the command line statement which executed the program's executable. With the sequential execution of the program, the command line parameter was ignored.

The portion of the code to be executed in parallel was contained within the statement=nts:

```
#pragma omp parallel
#pragma omp for
```

which are the OpenMP `parallel` and `for` directives. Only one portion of each program was contained with these statements. The other portions of the program initialized the program, performed timing operations, and printed data relevant to the program executed.

2.2.1 Finding prime numbers

The code in Figure 2.2 is a modified version of the example on page 279 of **czech2016** which computes prime numbers using the sieve of Eratosthenes. The modifications here are the `#ifdef _OPENMP` clauses to handle change of threads in use, and the method for measuring execution timing. The arrays `a[]` and `primes[]` were moved from being stored on the stack to the heap so to enable their dimension to be larger than if on the stack. The code calculates the prime numbers in the range `[2..N]` where `N` is a constant.

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

#define N 10000000
#define S (int)sqrt(N)
#define M N/10

long int a[100000];
long int primes[M];

int main(int argc, char** argv)
{
  long int i, k, number, remainder;
  long int no_of_div;
  long int no_of_primes = 0;
  double ticks;

#ifdef _OPENMP
  printf("threads = %s\n", argv[1]);
  omp_set_num_threads(atoi(argv[1]));
#endif
  ticks = omp_get_wtime();

  #pragma omp parallel
  #pragma omp for
  for ( i = 2; i <= S; i++ ) a[i] = 1;
  for ( i = 2; i <= S; i++ )
    if ( a[i] == 1 ) {
      primes[no_of_primes++] = i;
      for ( k = i + i; k <= S; k += i ) a[k] = 0;
    }

  no_of_div = no_of_primes;
  #pragma omp parallel
  #pragma omp for
  for ( number = S + 1; number <= N; number++ ) {
    for ( k = 0; k < no_of_div; k++ ) {
      remainder = ( number % primes[k] );
      if ( remainder == 0 ) break;
    }
    if ( remainder != 0 ) {
      #pragma omp critical
      primes[no_of_primes++] = number;
    }
  }

  ticks = omp_get_wtime() - ticks;
  printf("Elapse time: %lf milli-sec\n", ticks*1000.0);
  return(0);
}
```

Figure 2.2: OpenMP code to find prime numbers in an interval

Table 2.1 contains the results of running the code of Figure 2.2. The values tabulated are the execution times for six different ranges over which prime numbers were sought. This range corresponds to the value of `N` set in the code. Each runtime value is in units of milli-seconds. The `mean` value in the table is the average execution value for the 25 runs performed. The `min` and `max` are the minimum and maximum, respectively, execution values of the 25 values which formed the mean.

Table 2.1: Execution speeds of prime number code

Threads		Prime search interval size					
		10^3	10^4	10^5	10^6	10^7	10^8
0	min	0.1	1.5	14.2	389.7	9337.3	223440.9
	mean	0.1	1.5	17.8	445.7	9789.1	223565.9
	max	0.1	1.6	21.1	467.8	9906.6	223622.2
1	min	0.1	1.6	16.2	374.2	9661.8	230162.6
	mean	0.1	1.6	18.9	446.4	10012.8	230631.9
	max	0.2	1.6	24.9	492.5	10263.7	230807.5
2	min	0.2	1.0	9.8	205.7	4950.7	116344.0
	mean	0.2	1.2	12.6	230.9	5165.2	116581.6
	max	0.3	1.2	14.2	259.4	5217.6	116744.0
3	min	0.2	0.8	10.2	121.5	3249.2	78050.5
	mean	0.3	1.0	10.6	141.1	3435.1	78220.1
	max	0.3	1.0	10.9	164.8	3496.0	78302.6
4	min	0.3	0.8	7.3	79.7	2445.0	58836.7
	mean	0.3	0.9	8.2	102.0	2578.8	58964.8
	max	0.4	1.0	8.5	130.3	2639.5	59126.9
5	min	0.3	0.8	6.6	57.0	2081.5	47182.0
	mean	0.4	0.9	7.0	77.1	2098.1	47437.6
	max	0.7	1.2	7.3	96.1	2123.9	47557.0
6	min	0.3	0.9	5.4	47.9	1726.2	39661.2
	mean	0.4	1.0	6.3	65.4	1766.4	39802.2
	max	1.2	1.1	6.5	84.4	1787.6	39863.7
7	min	0.4	0.9	6.9	69.2	1881.0	45480.4
	mean	0.4	1.1	7.3	83.7	1997.6	45755.6
	max	0.4	1.7	7.7	99.5	2040.5	45823.6
8	min	0.4	1.0	6.2	56.8	1721.6	39935.3
	mean	0.5	1.1	6.7	77.5	1757.5	40107.9
	max	1.3	1.9	6.9	94.8	1787.1	40249.0
9	min	0.5	1.1	5.8	44.7	1446.3	35657.9
	mean	0.5	1.2	6.3	59.9	1539.1	35802.6
	max	0.6	1.7	6.6	74.5	1603.5	35858.0
10	min	0.5	1.1	5.4	43.0	1386.9	33811.2
	mean	0.6	1.3	6.1	54.9	1443.5	34111.0
	max	0.6	2.1	6.4	50.3	1486.4	34166.5
11	min	0.6	1.2	5.2	33.9	1324.3	31831.9
	mean	0.6	1.3	6.2	44.6	1357.4	31978.7
	max	0.6	1.7	6.7	55.9	1402.5	32834.3
12	min	0.6	1.3	5.3	30.1	466.9	10920.8
	mean	0.6	1.3	6.3	31.8	468.9	10977.9
	max	0.7	1.4	6.6	33.6	473.0	11038.0

Figure 2.3 is a plot of the execution time speedup obtained for all prime number search intervals to which the program code of Figure 2.2 was applied. The speedup values were calculated from the data of Table 2.1. Each speedup line shows values relative to the mean execution time of the corresponding sequential execution. The black line in Figure 2.3 shows the theoretical speedup if this was determined only by the number of threads used.

Figure 2.4 is a plot of the execution times for the two lowest integer ranges over which the prime number program was run. It follows on from Figure 2.3 which shows the poor speedup performance for the 10^3, 10^4 and 10^5 to a lesser extent. The error bars represent the minimum and maximum values corresponding to the mean value, all of which are from Table 2.1. The 10^3 (E3) search range is shown to have greater execution time when large numbers of threads are used. In the 10^4 (E4) range, increasing thread use to 3 resulted in reduced execution time. Further increase in thread use resulted in a reduced reduction to execution time, but lower than the sequential execution time. Only minor reduction in execution time was obtained.

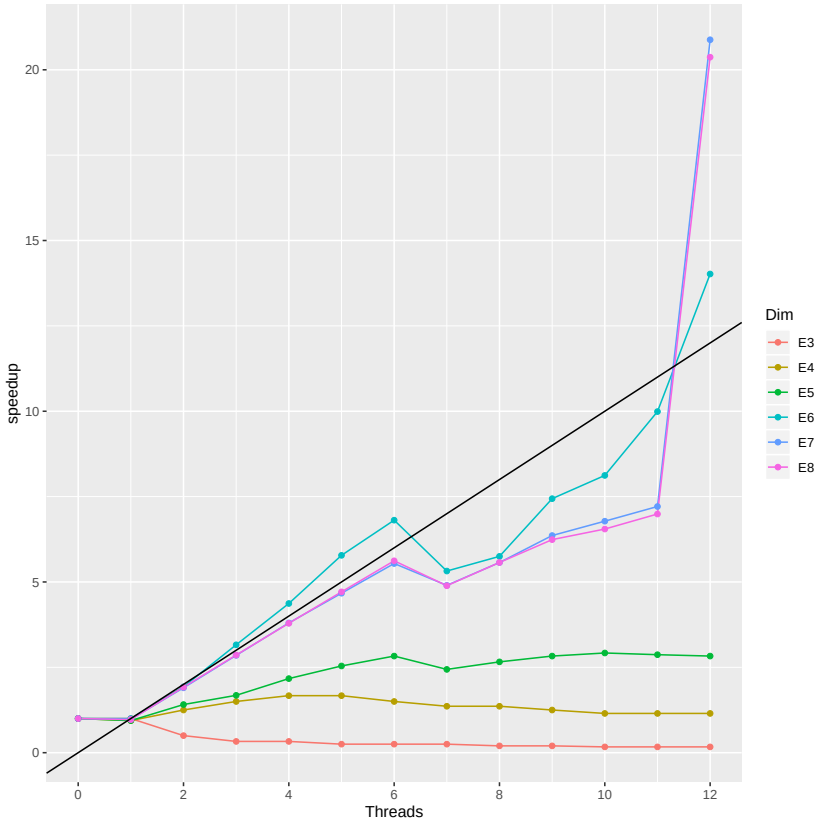

Figure 2.3: Speedup obtained with the prime number program using OpenMP primatives alone

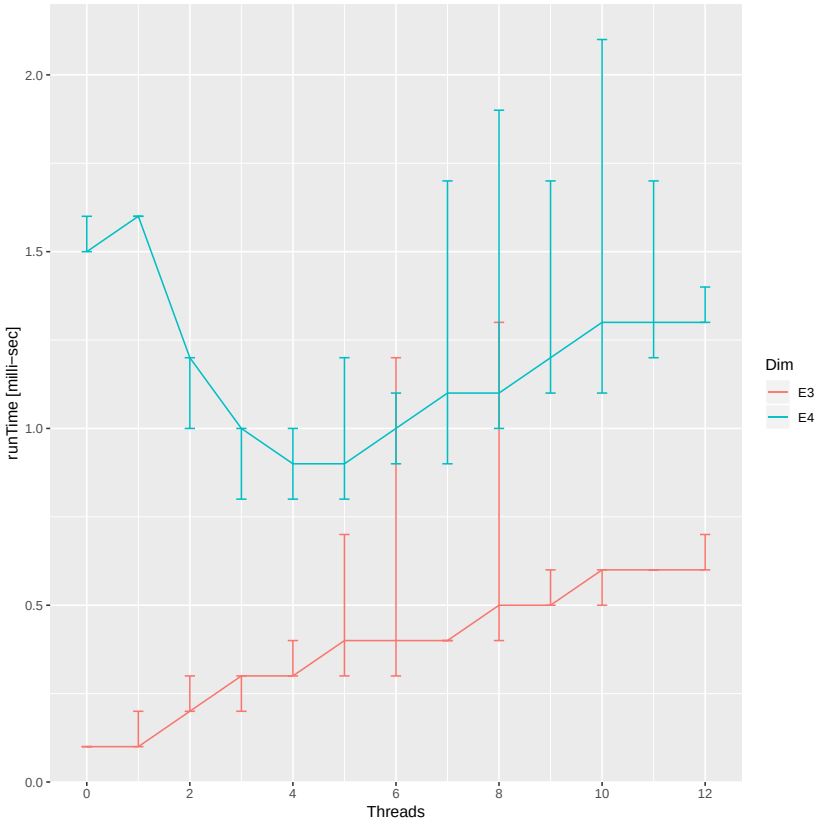

Figure 2.4: Plot of prime numbers program behaviour in 10^3 to 10^4 search range

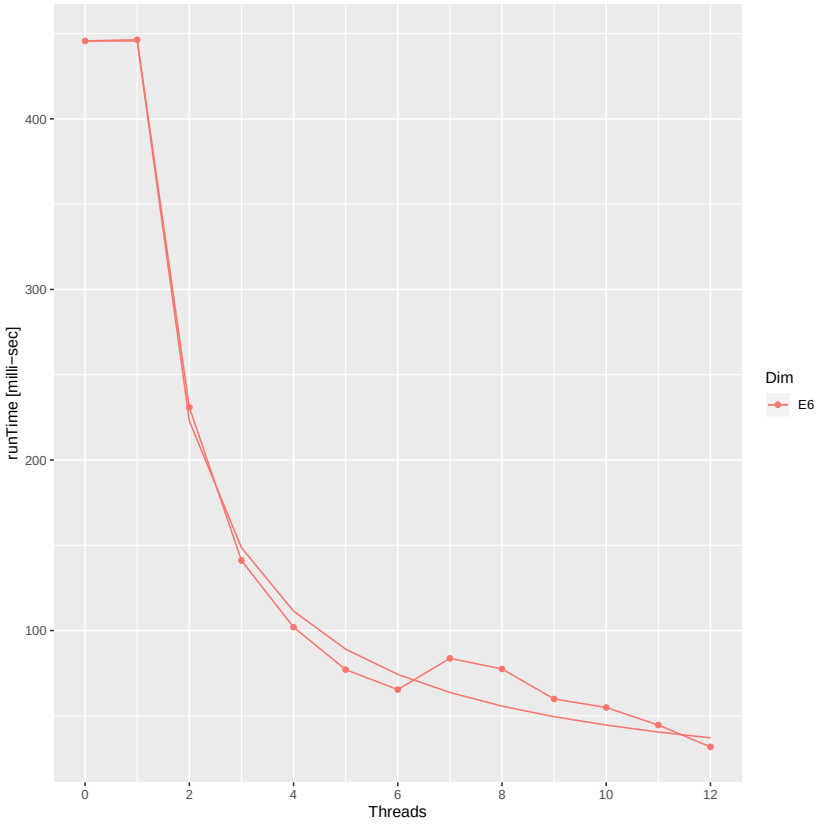

Figure 2.5: Execition time of prime number program using 10^7 range

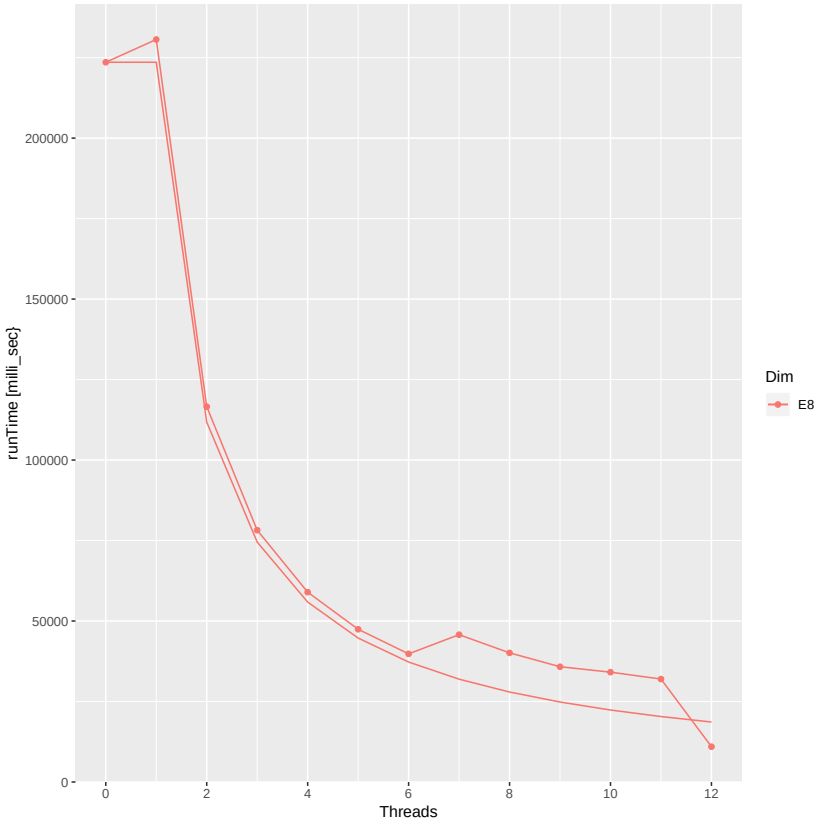

Figure 2.6: Execition time of prime number program using 10^8 range

Figure 2.3 suggests the OpenMP induced speedup is more pronounced for the 10^7 and 10^8 search ranges. Figures 2.5 and 2.6 plot the mean execution execution time from Table 2.1 for these two ranges. The execution times are plotted together with their data point. In both plots the other curve shows the theoretical execution time of each. Both plots show using 6 to 11 threads resulted in greater execution time compared to the theoretical, but with 12 threads better than the theoretical was obtained.

In these, and also subsequent trials, the maximum or minimum execution time values often occurred accompanying the first or final run of the program in the sequence. These maximum and minimum values were obtained by inspecting the execution times printed by each program run when the sequence was completed. In such a sequence of program runs, the *dimension* of the program and the number of threads used were constant. Whether this is chance or something which can be explained maybe worth further investigation beyond this current project.

2.2.2 Overview of matrix multiplication

Multiplication of two matrices is both a common numerical operation performed in scientific computation and one requiring a large amount of computation. If the two matrices are square with dimension $n \times n$, then the number of multiplications and additions required is n^3. For larger values of n, this *operation count* can be large. The existence of this large operation count, and the frequent occurrence in practical computing of matrix multiplication, make this process a good candidate for the study of parallel computing.

Three means of storage and consequent methods of handling the multiplication of the arrays was used. The programs each have three arrays `a[], b[]`, and `c[]` of integer values; arrays `a[]` and `b[]` are multiplied together producing the result in array `c[]`. These arrays had equal numbers of row and columns which was set by the constant `DIM` which corresponded to the dimension n of the matrix. This constant was changed to match the problem size being studied and then recompiled. The number of threads used to perform the multiplication was entered as a command line parameter each time the program was executed. The manner of storing and handling the matrices causes address calculation and data handling overheads which influence the speedup parallel processing can achieve.

The matrices `a` and `b[]` were filled with random integer values modulo 100. This filling was not taken as part of the execution time. However, the manner of storing the arrays was taken into consideration during this filling process. This was such that any particular set of such random numbers would produce the same matrix product independent of the manner in which the arrays were stored.

2.2.3 Matrix multiplication using row/column matrix storage

The code shown in Figure 2.7 uses the conventional row and column storage of the `a[], b[]` and `c[]` matrices holding the multiples and the result, respectively, as two dimensional arrays. Because C is the programming language used, each array is internally stored by rows. So from the perspective of address calculation in the matrix multiplication $a \times b$ each element in the array `a[]` is next integer in memory, but the value from the `b[]` matrix involves a multiplication by the `DIM` constant to locate the required integer in memory holding the `b[]` array. The addressing of the `c[]` array is the same as the `a[]` array.

In Figure 2.7 the array dimension `DIM` is set as 4000 but this was changed to fulfil each size requirement then recompiling the code.

Table 2.2 shows the results obtained. The values in the body of the table are execution times in units of milli-seconds. Of particular note is the doubling of the execution times in the 4000 and 5000 dimension cases between the sequential (tread 0) and thread 1 case. This was unexpected since sequential execution uses one thread. However, the minimum, maximum, and mean values shown in the table suggest this is not an error.

Figure 2.9 is a plot of the execution time speedup obtained for all matrix dimensions to which the program code of Figure 2.7 was applied. The speedup values were calculated from the data of Table 2.2. Each speedup line shows values relative to the mean execution time of the corresponding sequential execution. The black line in Figure 2.9 shows the theoretical speedup if this was determined only by the number of threads used.

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

#define DIM 4000

int a[DIM][DIM], b[DIM][DIM], c[DIM][DIM];

int main(int argc, char * argv[])
{
  int i, j, k;
  int dot;
  int threads;
  double ticks;

#ifdef _OPENMP
  printf("threads = %s  DIM = %d\n", argv[1], DIM);
  omp_set_num_threads(atoi(argv[1]));
#endif

  srand(time(NULL));
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)  {
      a[i][j] = rand() % 100;
      b[i][j] = rand() % 100;
    }

  ticks = omp_get_wtime();
  #pragma omp parallel
  #pragma omp for
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
    {
      dot = 0;
      for (k=0; k<DIM; k++)
         dot += a[i][k]*b[k][j];
      c[i][j] = dot;
    }
  ticks = omp_get_wtime() - ticks;
  printf("Elapse time: %lf milli-sec\n", ticks*1000.0);
  return 0;
}
```

Figure 2.7: OpenMP code implementing square matrix multiplication

A number of observations follow from Figure 2.9. In all cases the speedup dropped below that of the sequential execution when thread 1 was used, i.e. in the transition from sequential to parallel execution. From this thread 1 execution, there was increased speedup as more threads were put into use, up to 6 threads. Then the speedup dropped. Depending on the dimension of the matrices involved, the speedup would improve. Only in the 2000 and 5000 dimension cases did the speedup exceed to theoretical speedup when using the maximum of 12 threads.

Figure 2.8 is a plot of the execution time of the 1000x1000 matrix multiplication. Figure 2.9 indicates

behaviour of this multiplication (as also with the 100 case) to be less affected by using increase threads than the other cases. Figure 2.8 indicates execution is always below the expected execution time as show by the line without the data point dots. Notice also the enhanced execution time in the sequential-parallel transition.

Table 2.2: Execution times of row/column square integer matrix multiplication using OpenMP directives

Threads		Array Dimension						
		100	500	1000	2000	3000	4000	5000
0	min	1.1	127.3	1309.0	14983.7	60012.1	151338.4	332182.8
	mean	1.1	172.8	1382.8	15020.0	61536.8	154002.0	336815.9
	max	1.1	191.9	1467.3	15056.7	62094.5	154765.6	339974.9
1	min	1.3	154.8	2240.0	23995.5	77265.6	350714.3	687864.8
	mean	1.4	214.4	2380.7	24206.8	81078.5	358592.9	694153.3
	max	1.4	255.1	2427.4	24531.4	82499.3	362882.6	709421.5
2	min	0.8	90.1	1136.9	11945.6	39543.4	176499.4	397065.3
	mean	0.8	103.6	1203.0	12082.1	40622.1	177762.4	348572.9
	max	0.8	111.2	1229.8	12210.7	41128.7	178274.0	349929.5
3	min	0.6	33.9	700.2	7946.2	26574.5	118532.3	233703.5
	mean	0.6	54.5	777.1	8036.4	27168.7	119017.5	235401.3
	max	0.7	66.8	813.8	8125.2	27587.9	119174.5	236820.8
4	min	0.5	23.8	553.0	5704.8	20145.0	89194.1	179016.3
	mean	0.6	37.4	580.5	6034.9	20558.3	89591.8	179720.0
	max	0.6	54.0	611.5	6134.9	21007.1	90100.1	180130.5
5	min	0.5	22.3	437.8	4744.8	16135.4	71378.4	144583.6
	mean	0.5	30.1	464.3	4836.4	16525.6	72129.1	145343.1
	max	0.6	47.6	485.0	4901.6	16816.3	72886.7	146095.5
6	min	0.5	19.1	320.8	3951.6	13660.7	60547.7	122687.7
	mean	0.5	22.6	379.4	4063.2	13916.6	60960.9	123529.4
	max	0.5	31.4	404.9	4095.4	14163.9	61087.1	123759.5
7	min	0.7	30.4	600.7	4634.5	15351.6	61533.9	117535.5
	mean	0.7	45.4	642.6	4886.3	15784.7	62179.7	118966.0
	max	1.5	64.9	687.8	4997.2	16100.6	63008.0	120584.7
8	min	0.7	24.1	516.0	4172.7	13339.7	54129.9	102855.5
	mean	0.7	34.5	562.1	4247.2	13801.0	54652.1	104421.5
	max	0.7	47.3	610.9	4300.7	14096.1	55024.9	109876.5
9	min	0.7	21.7	475.2	3725.4	11737.9	48313.5	91977.7
	mean	0.7	26.0	519.0	3768.3	11922.3	48595.0	92353.2
	max	0.7	35.7	566.7	3804.6	12088.3	48952.9	93007.5
10	min	0.7	19.1	570.7	3760.1	11349.0	44380.0	84033.1
	mean	0.8	24.2	648.8	3839.9	11707.3	44998.0	84745.9
	max	1.5	37.2	702.7	3907.9	11884.0	45667.5	85917.2
11	min	0.7	17.5	664.2	3564.0	10298.0	40936.4	76487.0
	mean	0.7	20.4	726.7	3625.1	10607.4	41439.7	77232.1
	max	0.8	23.7	770.4	3703.1	11010.0	44927.7	80497.6
12	min	0.7	15.6	599.2	1355.3	4087.7	14460.5	27475.6
	mean	0.8	17.5	611.2	1366.4	4248.5	14702.4	27517.7
	max	1.6	21.0	618.2	1378.1	4290.0	14823.9	27597.5

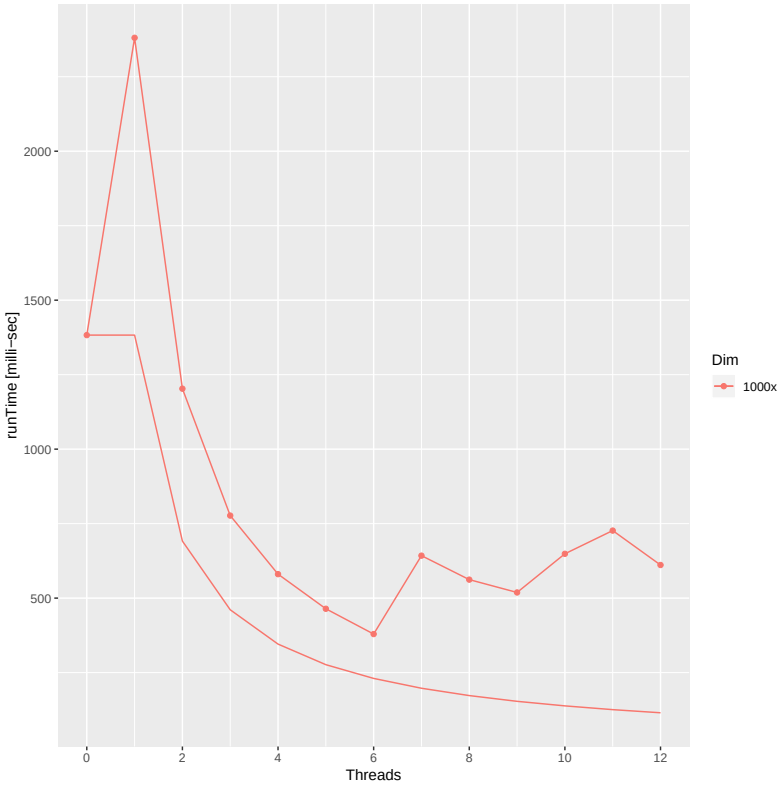

Figure 2.8: Execution time of 1000x1000 matrix multiplication row/column program using OpenMP primatives alone

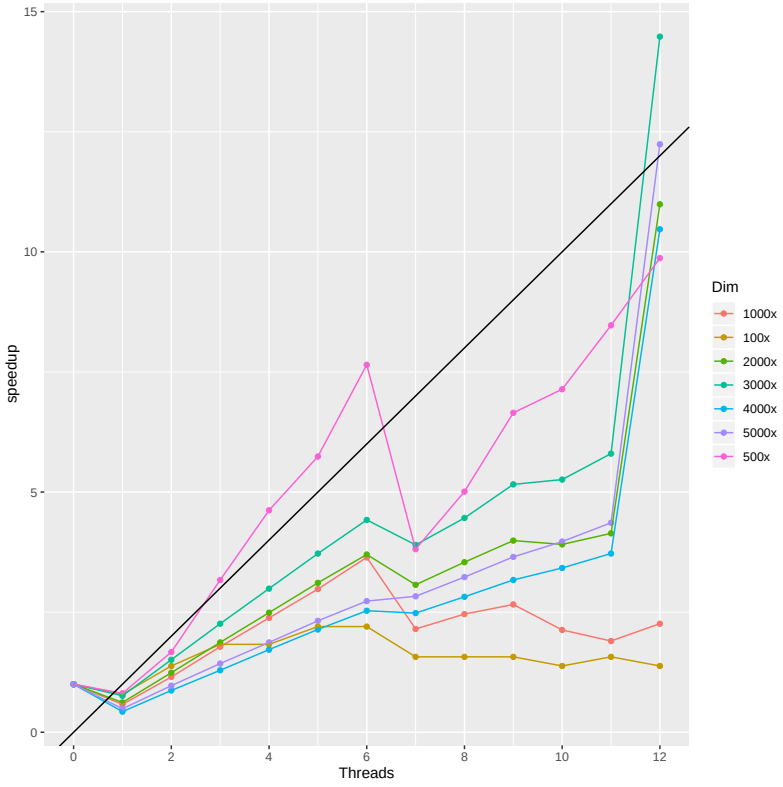

Figure 2.9: Speedup obtained with matrix multiplication row/column program using OpenMP primatives alone

2.2.4 Matrix multiplication using row/row matrix storage

Can execution time behaviour of Section 2.2.3 be improved by reducing the hidden but present address calculation required to find the matrix elements to multiply and thus partial explaining the results?

To examine this question, the code of Figure 2.7 was modified. Instead of storing the second matrix (matrix `b[]` in the code) in a standard column fashion, it was stored by rows. The elements of each column were stored in successive memory locations. So instead of accessing a column element from the second matrix in a multiplication, a row element access was used. This was done by replacing the `dot += a[i][k]*b[k][j];` statement with `dot += a[i][k]*b[j][k];`. For the calculation to remain correct, matrix `b[]` needed to be initialized with it's column and row elements swapped. This was done by replacing the statement `b[i][j] = rand() % 100;` by `b[j][i] = rand() % 100`. The code resulting is shown in Figure 2.10.

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

#define DIM 100

int a[DIM][DIM], b[DIM][DIM], c[DIM][DIM];

int main(int argc, char * argv[])
{
  int i, j, k;
  int dot;
  int threads;
  double ticks;

#ifdef _OPENMP
  printf("threads = %s  DIM = %d\n", argv[1], DIM);
  omp_set_num_threads(atoi(argv[1]));
#endif

  srand(time(NULL));
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++) {
      a[i][j] = rand() % 100;
      b[j][i] = rand() % 100;
    }

  ticks = omp_get_wtime();
  #pragma omp parallel
  #pragma omp for
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
    {
      dot = 0;
      for (k=0; k<DIM; k++)
         dot += a[i][k]*b[j][k];
      c[i][j] = dot;
    }
  ticks = omp_get_wtime() - ticks;
  printf("Elapse time: %lf milli-sec\n", 1000.0*ticks);
  return 0;
}
```

Figure 2.10: OpenMP code implementing square matrix multiplication using row/row storage

Table 2.3 contains the results obtained by running the code of Figure 2.10 for matrix dimensions of 100 to 5000. With respect to Table 2.3 the minimum, means, and maximum values in some instances have the same value under the 100 column. This is due to one decimal place being retained in recoding execution time through all experiments.

Table 2.3: Execution times of row/row square integer matrix multiplication using OpenMP directives

Threads		Array Dimension						
		100	500	1000	2000	3000	4000	5000
0	min	0.9	38.9	793.3	7602.8	26359.8	64406.6	123473.3
	mean	1.0	76.6	855.1	7831.5	26493.2	64844.1	123832.0
	max	1.0	104.0	907.0	7928.9	26711.3	64899.7	125127.0
1	min	0.9	43.1	825.8	7516.4	26134.2	64405.8	123200.3
	mean	1.0	71.0	871.4	7820.4	26459.1	64537.8	123835.6
	max	1.0	101.7	903.9	7938.9	26686.6	64585.4	124912.3
2	min	0.6	23.1	369.7	3797.8	13347.2	32354.6	62554.6
	mean	0.6	33.1	433.7	4007.1	13383.6	32654.1	62979.5
	max	0.6	49.0	452.6	4193.8	13433.9	32810.1	64572.8
3	min	0.5	17.1	242.5	2753.4	9392.7	22884.8	44861.7
	mean	0.5	20.7	277.1	2792.2	9440.2	22924.8	44978.7
	max	0.5	32.7	303.1	2834.7	9524.7	23185.2	45436.6
4	min	0.4	13.6	181.7	2039.3	6960.8	17169.0	33643.8
	mean	0.5	15.6	207.2	2090.3	7116.0	17299.4	33873.7
	max	0.6	19.2	227.6	2140.4	7220.2	17446.0	34106.8
5	min	0.4	10.3	133.9	1605.8	5647.7	13442.7	27215.6
	mean	0.4	13.6	164.9	1675.1	5751.7	13910.4	27344.1
	max	0.6	17.2	189.1	1717.3	5805.0	13998.9	27400.5
6	min	0.4	8.2	101.2	1380.0	4767.7	11656.3	22937.8
	mean	0.5	9.3	127.6	1414.7	4864.2	11725.8	23049.4
	max	0.5	14.3	154.0	1443.8	4898.8	11774.6	23094.0
7	min	0.6	15.1	139.9	1636.9	5584.4	13944.7	26960.8
	mean	0.6	16.2	176.7	1692.7	5763.2	14124.4	27293.7
	max	0.6	18.2	208.2	1750.1	5852.1	14251.4	27459.3
8	min	0.6	13.5	129.5	1456.5	5079.5	12386.2	23815.2
	mean	0.6	16.0	162.8	1487.1	5110.1	12481.4	24072.5
	max	0.6	20.2	181.4	1525.5	5159.2	12589.3	24139.8
9	min	0.6	10.9	117.5	1290.8	4422.8	11111.8	21544.2
	mean	0.6	13.9	148.6	1320.2	4562.1	11217.1	21758.3
	max	1.0	18.5	187.8	1355.5	4665.5	11332.4	22290.4
10	min	0.6	9.6	112.6	1286.6	4519.9	11185.7	21443.9
	mean	0.7	11.2	142.9	1339.0	4567.9	11282.0	21772.1
	max	0.7	15.6	175.5	1373.2	4625.9	11354.9	22271.8
11	min	0.6	8.9	103.6	1202.7	4186.2	10614.2	20178.7
	mean	0.7	10.4	130.4	1253.3	4319.9	10761.7	20525.0
	max	0.7	14.5	166.3	1307.2	4370.9	11162.7	20738.9
12	min	0.7	8.2	58.1	439.1	1417.6	3457.1	6504.2
	mean	0.7	9.3	60.4	443.4	1426.2	3508.8	6535.9
	max	1.6	12.6	63.8	446.3	1506.1	3783.7	6626.3

Figure 2.11 plots the speedup computed from the values in Table 2.3. The black line shows the speedup which would be expected if it was determined only by the number of threads in use.

Figure 2.9 together with Table 2.2 and Figure 2.11 together with Table 2.3 are used to compare the row/column and row/row matrix multiplication processes. These show:

- The speedup increases for most matrix sizes until 6 threads are in use, then it decreases. However, in Figure 2.11 the speedups for all matrix sizes are closer to the expected speedup line, in some cases exceeding it.
- The drop in speedup in going from sequential execution to parallel execution shown in Figure 2.9

has disappeared from Figure 2.11.

- In Figure 2.11 all but two matrix size multiplication exceeded the expected speedup when using 12 threads, compared with only two exceeding it in Figure 2.9.
- The execution times for both sequential and parallel processing is smaller when using row/row processing.
- There is an increase in speedup in both approaches when using between 7 and 11 threads which is approximately the same.
- In multiplication of low dimension matrices, for example 100x100, there is only minimal improvement is execution time by using parallel processing using either row/column or row/row storage.

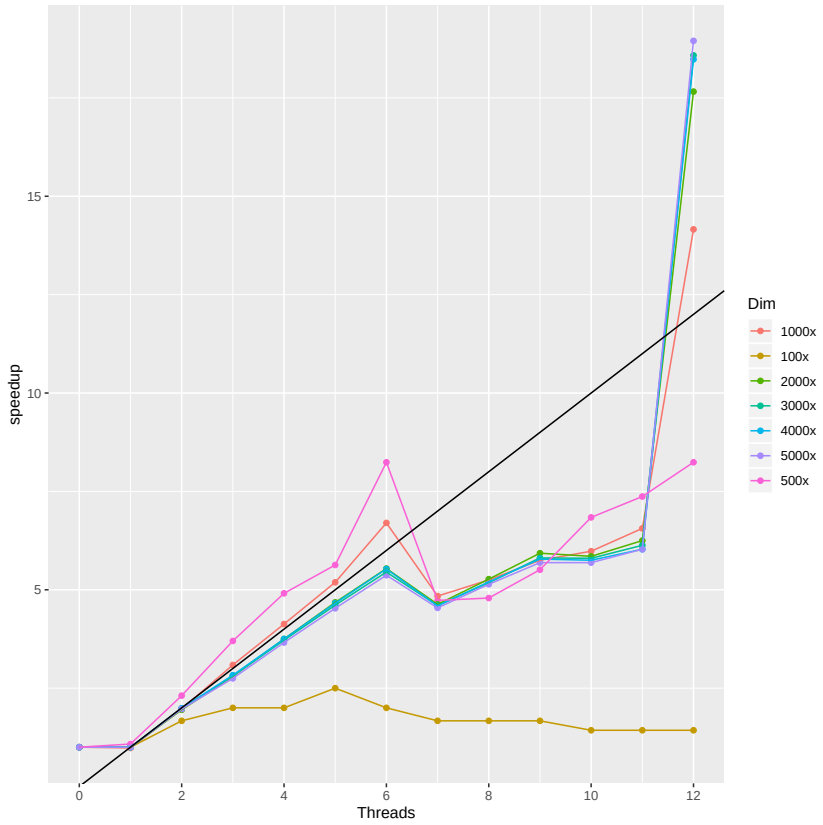

Figure 2.11: Speedup obtained with matrix multiplication row/row program using OpenMP primatives alone

Figure 2.12 is comparable with Figure 2.8 in showing the execution time of the 1000x1000 matrix multiplication. Both show below expected speedup is aligned with greater execution time, in the cases when using 7 to 11 threads.

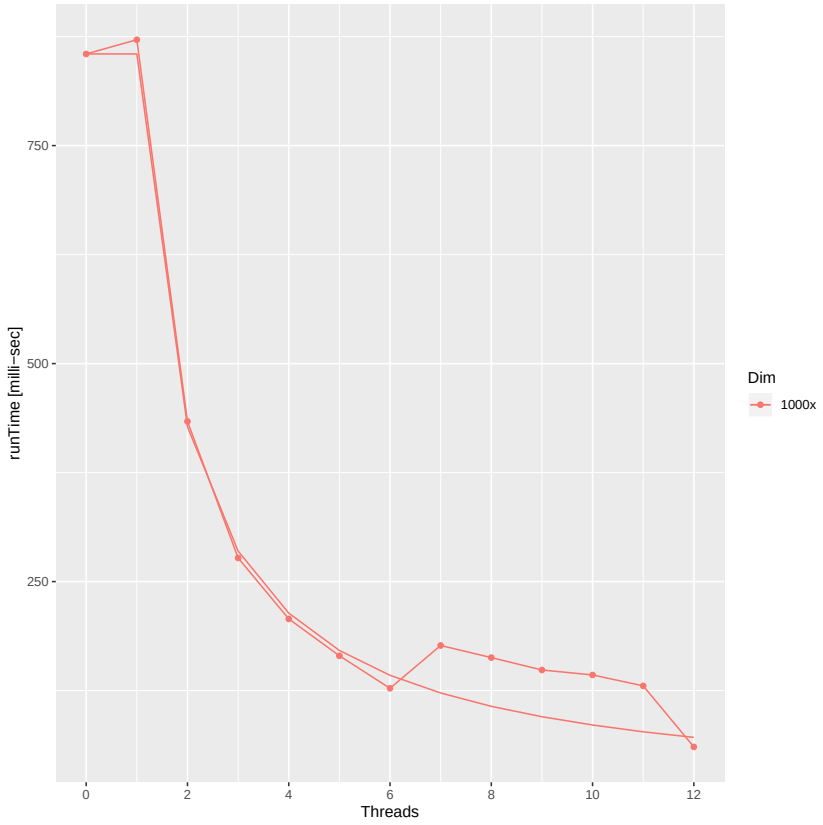

Figure 2.12: Execution time of 1000x1000 matrix multiplication row/row program using OpenMP primitives alone

2.2.5 Matrix multiplication using vector matrix storage

Building upon the advantage shown in Section 2.2.4, can further improvements be obtained in execution time if the rows of columns are changed to vectors of columns?

The storage here is similar too that in the code of Figure 2.10 in that arrays `a[]`, `b[]` and `c[]` are stored by rows. However, here instead of accessing the rows of each matrix within a two dimensional array, each array was accessed as a one dimensional array, or as a vector. The two vectors `a[]` and `b[]` which are involved in performing the multiplication were divided into blocks each containing a row of the matrix. The number of values in each block is the value of the `DIM` constant. Multiplication was then performed using two of these blocks at a time. The result of each multiplication was stored in sequential location of the vector representing the `c[]` array.

Table 2.4 contains the results obtained by running the code of Figure 2.13 for matrix dimensions of 100 to 5000. With respect to Table 2.4 the minimum, means, and maximum values in some instances have the same value under the 100 column. This is due to one decimal place being retained in recoding execution time through all experiments.

Figure 2.14 plots the speedup computed from the values in Table 2.4. The black line shows the speedup which would be expected if it was determined only by the number of threads in use.

Comparing Figure 2.14 with Figure 2.11 indicates little difference in speedup behaviour against thread use for each of the dimensions of matrices multiplied. By comparing the values in Tables 2.4 and Table 2.3, from which these plots were derived, the vector values are slightly, but not significantly, less than the corresponding row/row values.

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

#define DIM 3000

int a[DIM][DIM], b[DIM][DIM], c[DIM][DIM];

int main(int argc, char * argv[])
{
  int i, j, k;
  int dot;
  int threads;
  int *x, *y, *z, *slice;
  double ticks;

#ifdef _OPENMP
  printf("threads = %s  DIM = %d\n", argv[1], DIM);
  omp_set_num_threads(atoi(argv[1]));
#endif

  srand(time(NULL));
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++) {
      a[i][j] = rand() % 100;
      b[j][i] = rand() % 100;
    }
  ticks = omp_get_wtime();
  z = &c[0][0];
  slice = &a[0][0];
  #pragma omp parallel
  #pragma omp for
  for (i=0; i<DIM; i++) {
    y = &b[0][0];
    for (j=0; j<DIM; j++) {
      dot = 0;   x = slice;
      for (k=0; k<DIM; k++) {
        dot += *x * *y;
        x++;  y++;
      }
      *z = dot;   z++;
    }
    slice += DIM;
  }
  return 0;
}
```

Figure 2.13: OpenMP code implementing square matrix multiplication using vector storage

The conclusion to be drawn is the vector method of performing matrix multiplication appears to offer a better parallel processing configuration than row/column and row/row processing. The advantage comes from reduced internal address calculation in locating of values in each matrix to be multiplied. But does this remain the case when more than minimal parallel and for OpenMP directives are used?

Table 2.4: Execution times of vector square integer matrix multiplication using OpenMP directives

Threads		Array Dimension						
		100	500	1000	2000	3000	4000	5000
0	min	0.9	54.2	715.3	7589.4	25297.8	62199.9	121011.5
	mean	0.9	87.3	819.7	7642.2	25835.2	62545.2	121286.5
	max	0.9	103.3	867.9	7671.4	25940.9	62616.0	121323.6
1	min	0.9	39.7	686.3	7923.9	25842.3	64150.5	123460.9
	mean	1.0	81.1	853.3	7947.0	26347.8	64341.8	124401.6
	max	1.0	106.7	910.9	7962.6	26439.9	64642.7	124960.9
2	min	0.6	24.2	384.8	3925.5	12682.6	31997.8	62841.4
	mean	0.6	31.6	429.7	4144.8	13402.7	32688.8	63067.6
	max	0.6	53.1	449.1	4230.4	13857.3	33505.2	63170.4
3	min	0.5	17.8	218.4	2765.0	9035.7	22410.0	42613.2
	mean	0.5	24.9	275.5	2805.7	9313.4	22867.8	44593.0
	max	0.5	33.5	301.8	2833.0	9528.2	23133.4	45278.1
4	min	0.4	12.4	184.0	2063.6	6913.3	17232.6	31678.6
	mean	0.5	15.8	206.0	2095.5	7111.8	17291.3	33666.0
	max	0.5	19.9	227.7	2127.3	7168.8	17400.9	33998.7
5	min	0.4	9.3	133.7	1594.1	5720.8	13880.7	27285.9
	mean	0.5	11.4	156.1	1682.0	5755.4	13932.9	27352.3
	max	0.8	16.5	181.7	1714.8	5788.1	13971.3	27411.9
6	min	0.4	8.3	108.4	1335.0	4527.7	11476.4	23055.6
	mean	0.4	9.7	127.3	1414.9	4825.3	11709.9	23098.9
	max	0.5	15.9	151.9	1440.8	4899.0	11762.6	23208.5
7	min	0.5	13.1	157.6	1650.1	5568.0	13602.5	26807.6
	mean	0.5	16.3	179.8	1689.4	5721.9	14020.9	27021.4
	max	0.6	22.1	202.5	1721.3	5786.9	14185.9	27180.5
8	min	0.6	11.9	128.0	1428.3	4894.0	12296.0	23700.1
	mean	0.6	13.7	162.8	1477.6	5046.3	12383.3	23925.8
	max	0.6	18.3	187.1	1509.5	5110.1	12511.2	24181.7
9	min	0.6	11.0	103.6	1265.4	4419.4	10905.2	21224.5
	mean	0.6	12.4	153.7	1305.7	4520.3	11110.0	21498.1
	max	0.7	16.0	178.0	1345.9	4572.1	11341.6	21651.7
10	min	0.6	9.2	107.5	1264.0	4441.8	11013.1	21218.4
	mean	0.7	11.3	132.5	1299.4	4516.8	11204.4	21431.2
	max	1.5	15.8	168.8	1335.3	4565.5	11500.1	21650.2
11	min	0.6	8.6	75.1	1185.5	4188.3	10455.9	20116.6
	mean	0.7	10.4	124.6	1245.8	4265.9	10653.5	20503.6
	max	0.7	14.2	163.2	1302.0	4483.0	10934.1	20543.2
12	min	0.7	8.0	54.2	427.6	1388.5	3454.5	6333.7
	mean	0.7	8.6	58.5	435.7	1405.2	3522.6	6532.2
	max	1.5	11.7	62.4	463.5	1413.7	3651.3	6758.8

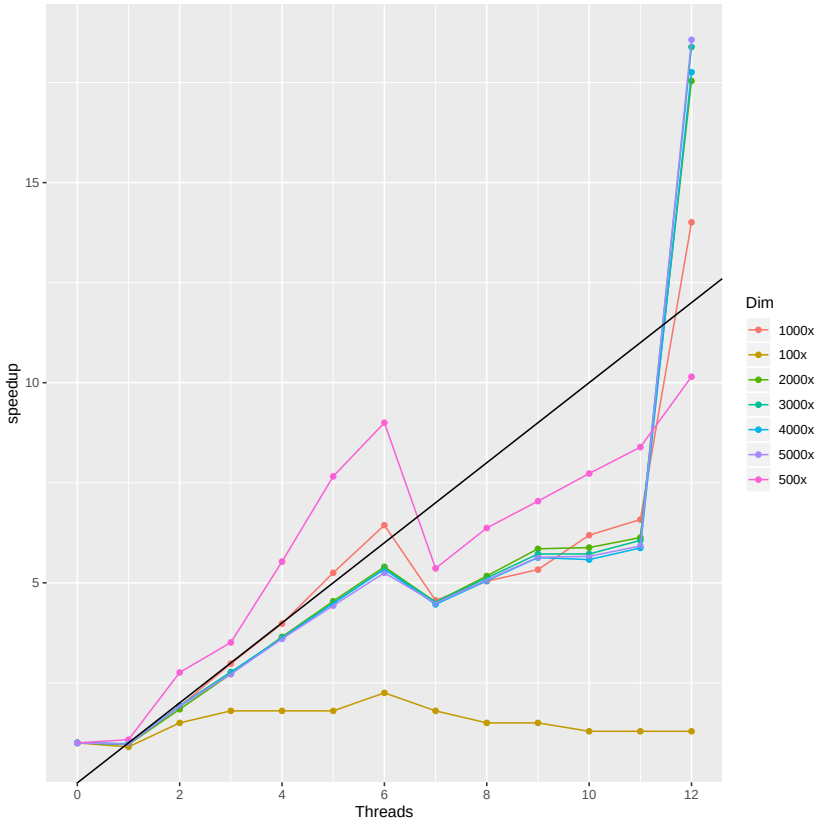

Figure 2.14: Speedup obtained with matrix multiplication vector program using OpenMP primitives alone

2.3 Adding clauses to the OpenMP parallel directive

The OpenMP parallel and for primitives allow a number of clauses to be added so as to modify their behaviour. In this section no clauses are added to the for directive which is used. However, `default`, `private` and `shared` clauses are added to the `parallel` directive. The specific purpose of each of these clauses is:

default() If a variable used in the range of a `parallel` directive is not mentioned in a `private()` or `shared()` clause, it is this clause which handles that variable. If the parameter `none` is given then all variables used withing the `parallel` directive's range must be given in either a `private` or `shared` clause else an error has occurred.

private() A personal copy of each variable mentioned in the parameter of this clause is used within the range of the associated `parallel` directive's range. Any change to the contents of such variables **are not** sent back to where they were obtained.

shared() A personal copy of the variables in the parameter of this clause **are not** obtained for use within the range of the `parallel` directive's range. Any change made to these variables withing the range of the `parallel` directive **go back** to the original variable.

These clauses give some program control over the flow of data into and out of the region of the program under going parallel execution.

The same examples are used here as in Section 2.2.

2.3.1 Finding prime numbers

A modified version of the program of Figure 2.2 was used here. The modification consisted of replacing the:

```
#pragma omp parallel
```

line by the line:

```
#pragma omp parallel default(none) private(k, number, remainder) \
shared(i, a, primes, no_of_primes, no_of_div)
```

Note the back-slash at the end of the line. This indicates the line following is a continuation of the OpenMP directive, in this case, the `parallel` directive.

Table 2.5 contains the execution times measured using the resulting program when applied to six search intervals. Figure 2.3 shows plots of the execution speedup for each of those search intervals computed from the tabled values.

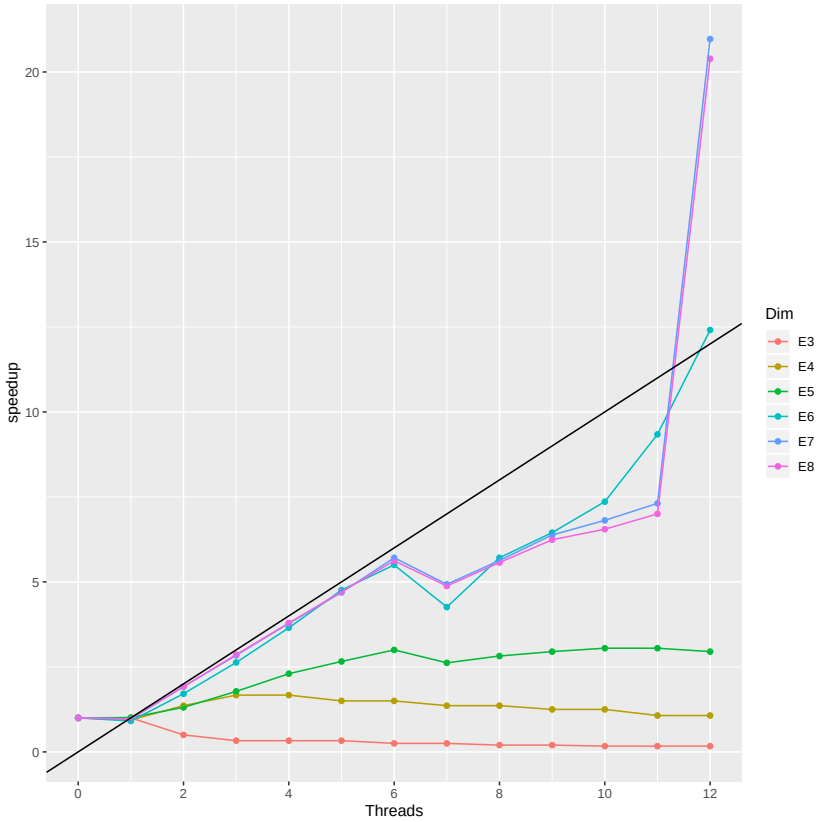

Figure 2.15: Speedup obtained by prime number finding program using OpenMP primitives with simple clauses

By comparing the execution times in Table 2.5 with those in Table 2.1 when no clauses were used with the `parallel` OpenMP directive, and the corresponding speedup plots of Figure 2.15 and Figure 2.3, no substantial change was obtained by using the `default`, `private` and `shared` clauses.

The `default` clause was remove from the code and a number of *spot checks* on the execution time of the resulting executable were performed. All those check times fell within the corresponding `min` to `max` limits of that executable in Table 2.5.

Table 2.5: Execution times of prime number code with default, private and share clauses

Threads		Prime search interval size					
		10^3	10^4	10^5	10^6	10^7	10^8
0	min	0.1	1.5	15.8	236.2	9757.9	223046.6
	mean	0.1	1.5	18.9	392.3	9861.3	223555.5
	max	0.1	1.6	22.0	466.8	9919.5	223650.9
1	min	0.1	1.6	17.6	319.2	9694.0	230410.5
	mean	0.1	1.6	18.8	431.8	10131.6	230663.7
	max	0.1	1.6	26.0	494.5	10258.3	230783.7
2	min	0.2	1.0	12.4	204.5	4852.5	116035.6
	mean	0.2	1.1	14.4	229.4	5148.9	116422.0
	max	0.3	1.2	15.3	253.7	5240.9	116581.4
3	min	0.2	0.8	9.2	124.8	3313.4	77971.8
	mean	0.3	0.9	10.6	148.9	3466.8	78126.3
	max	0.7	1.0	11.0	164.8	3511.1	78245.8
4	min	0.3	0.7	7.8	86.2	2465.8	58705.3
	mean	0.3	0.9	8.2	107.5	2609.7	58929.2
	max	0.3	0.9	8.5	133.2	2661.2	59042.6
5	min	0.3	0.8	6.6	65.5	2078.2	47405.8
	mean	0.3	1.0	7.1	82.4	2103.0	47455.8
	max	0.4	1.7	8.0	104.3	2132.8	47515.7
6	min	0.3	0.9	5.7	44.0	1647.7	39685.1
	mean	0.4	1.0	6.3	71.3	1728.1	39779.8
	max	0.4	1.1	7.2	89.1	1781.5	39838.4
7	min	0.4	0.9	6.9	74.9	1961.9	45522.4
	mean	0.4	1.1	7.2	92.0	2000.7	45780.7
	max	0.4	1.4	8.2	109.1	2032.7	45863.1
8	min	0.4	1.0	5.9	51.9	1708.6	39978.3
	mean	0.5	1.1	6.7	68.7	1750.8	40157.5
	max	0.5	1.2	6.9	88.0	1782.4	40286.3
9	min	0.5	1.1	5.8	39.4	1476.1	35569.1
	mean	0.5	1.2	6.4	60.8	1545.2	35810.1
	max	0.5	2.0	6.7	83.1	1589.9	35877.7
10	min	0.5	1.2	5.3	36.6	1377.9	33992.0
	mean	0.6	1.2	6.2	53.3	1447.0	34144.8
	max	1.4	1.3	6.6	73.8	1484.5	35158.8
11	min	0.6	1.2	5.2	38.7	1277.1	31800.1
	mean	0.6	1.4	6.2	42.0	1348.1	31947.2
	max	1.4	2.2	6.6	44.9	1388.1	32951.6
12	min	0.6	1.3	5.2	30.2	466.8	10883.3
	mean	0.6	1.4	6.4	31.6	470.3	10962.6
	max	0.7	1.4	7.4	32.5	475.8	11026.9

2.3.2 Matrix multiplication using row/column matrix storage

A modified version of the program of Figure 2.7 was used here. The modification consisted of replacing the:

```
#pragma omp parallel
```

line by the line:

```
#pragma omp parallel default(none) private(i, j, k, dot) \
shared(a, b, c)
```

Note the back-slash at the end of the line. This indicates the line following is a continuation of the OpenMP directive, in this case, the `parallel` directive.

Table 2.6: Execution times of row/column matrix multiply code using default, private and shared clauses

Threads		Array Dimension						
		100	500	1000	2000	3000	4000	5000
0	min	1.1	63.2	1365.3	15040.0	58905.7	152019.2	331824.3
	mean	1.1	127.7	1423.8	15108.7	61099.9	152155.0	335196.4
	max	1.1	166.7	1483.1	15183.4	68938.1	152329.9	339680.7
1	min	1.1	62.9	1373.4	15032.7	61151.3	151995.9	330873.4
	mean	1.1	123.3	1428.4	15075.2	61646.9	152196.2	331867.8
	max	1.1	164.2	1485.0	15109.4	62020.6	152363.3	332979.8
2	min	0.7	40.4	674.4	7387.7	28993.9	75886.2	161934.8
	mean	0.7	57.5	701.4	7424.6	30437.8	76744.2	166463.8
	max	0.7	86.1	730.1	7458.9	30844.8	76953.3	169107.5
3	min	0.5	28.2	430.8	4896.1	20232.7	50410.8	110502.4
	mean	0.5	38.4	460.4	4941.9	20444.4	50891.4	111070.3
	max	0.6	44.4	479.5	4973.2	20536.6	51115.4	111433.4
4	min	0.5	21.8	320.5	3535.7	14965.6	38246.2	83726.6
	mean	0.5	27.6	339.5	3672.9	15284.1	38348.8	84240.3
	max	0.5	46.7	369.1	3733.6	15442.4	38386.1	84488.8
5	min	0.5	18.2	242.4	2829.2	11917.6	30762.9	67209.9
	mean	0.5	21.1	261.6	2977.1	12213.4	30987.0	67622.8
	max	0.5	30.8	283.7	3011.4	12376.4	31102.5	68036.9
6	min	0.5	13.6	192.9	2419.4	9922.3	25735.8	54762.7
	mean	0.5	15.8	223.3	2503.1	10232.6	25921.8	56498.0
	max	0.5	19.5	236.8	2525.7	10329.0	25962.3	56865.3
7	min	0.6	18.2	323.0	2859.2	11213.4	29488.9	59713.9
	mean	0.6	22.4	353.3	2897.5	11725.9	29681.9	60728.4
	max	0.9	36.4	397.2	2932.8	11930.9	29788.5	62076.9
8	min	0.6	15.7	294.4	2441.8	9984.0	25459.9	52011.3
	mean	0.6	18.3	319.4	2530.1	10186.4	25666.6	52558.2
	max	0.7	21.6	346.5	2584.6	10393.6	25783.3	53249.8
9	min	0.6	12.1	238.2	2217.2	8891.7	22876.6	46236.0
	mean	0.6	16.5	282.7	2258.2	9167.1	23675.9	46643.5
	max	1.4	24.2	330.5	2307.9	9263.8	25511.9	47008.7
10	min	0.6	12.7	262.2	2159.3	8651.5	21854.0	43203.3
	mean	0.7	14.8	332.1	2217.5	8922.7	23197.8	43885.0
	max	0.7	18.0	373.6	2299.2	9028.9	24847.3	44369.4
11	min	0.6	11.1	323.8	2025.8	8274.3	20419.1	40054.3
	mean	0.7	13.8	369.5	2129.6	8388.4	21388.6	40420.6
	max	0.8	24.3	436.0	2195.0	8535.0	23000.9	40842.2
12	min	0.7	10.3	178.9	683.8	2654.6	6711.2	12469.3
	mean	0.7	11.6	183.1	691.4	2718.1	6758.9	12616.8
	max	1.6	14.4	187.7	721.0	2790.0	6782.5	12716.3

Table 2.6 contains the execution times measured when the resulting program was applied to seven matrix sizes. Figure 2.16 shows plots of the execution speedup for each of those matrix sizes computed from the tabled values.

Comparing the execution times in Table 2.6 with those in Table 2.2 when no clauses were used with the `parallel` OpenMP directive, noticeably smaller execution times were produced in the clause use case of Table 2.6. By comparing the corresponding speedup plots of Figure 2.16 and Figure 2.9, greater speedup was also obtained by using the `default`, `private` and `shared` clauses.

The `default` clause was remove from the code and a number of *spot checks* on the execution time of the resulting executable were performed. All those check times fell within the corresponding `min` to `max` limits of that executable in Table 2.6.

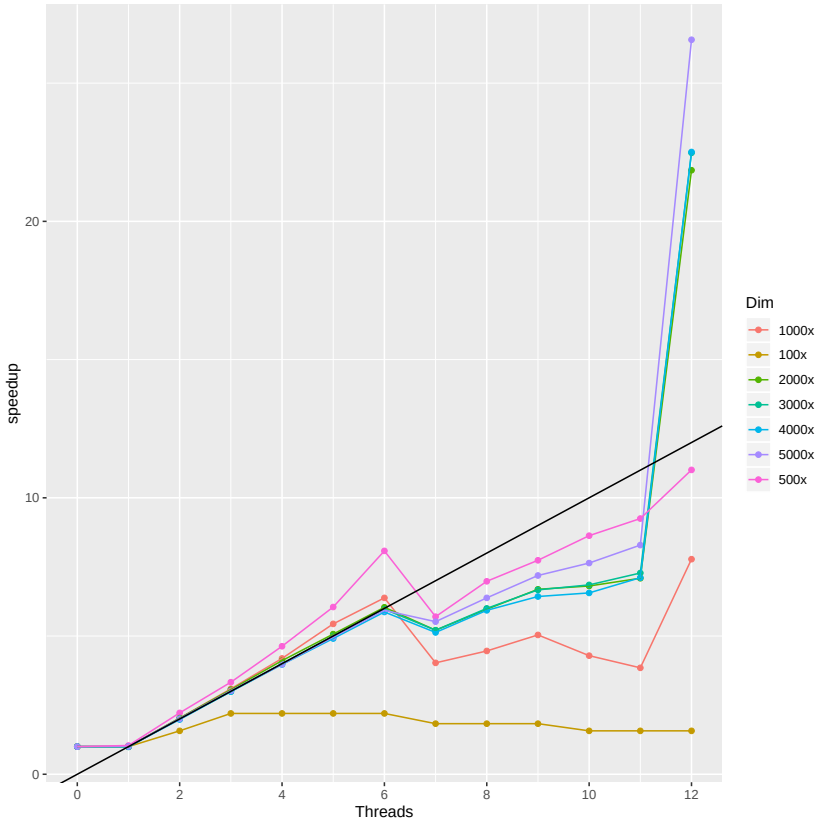

Figure 2.16: Speedup obtained with matrix multiplication using row/column program using OpenMP default, private and shared clauses

2.3.3 Matrix multiplication using row/row matrix storage

Table 2.7 contains the execution times measured when the `default`, `private`, and `shared` clauses of Section 2.3.2 were used in the matrix multiplication program of Figure 2.10 for seven matrix sizes. Figure 2.16 shows plots of the execution speedup for each of those search intervals computed from those tabled values.

Comparing the execution times in Table 2.7 with those in Table 2.3 when no clauses were used with the `parallel` OpenMP directive, and the corresponding speedup plots of Figure 2.17 and Figure 2.11. In both comparison pairs no substantial change was obtained by using the `default`, `private` and `shared` clauses.

The `default` clause was remove from the code and a number of *spot checks* on the execution time of the resulting executable were performed. All those check times fell within the corresponding `min` to `max` limits of that executable in Table 2.7.

Table 2.7: Execution times of row/row matrix multiply code using default, private and shared clauses

Threads		Array Dimension						
		100	500	1000	2000	3000	4000	5000
0	min	0.9	60.7	787.9	7668.6	26636.4	64508.9	123613.6
	mean	0.9	85.3	850.1	7850.5	26685.7	64791.0	124426.1
	max	1.0	103.3	907.5	7927.8	26711.9	64908.8	125187.9
1	min	1.0	39.3	723.5	7874.6	26265.2	63957.9	123357.8
	mean	1.0	73.2	853.0	7922.3	26361.3	64355.6	123864.7
	max	1.0	100.4	912.2	7949.6	26400.2	64548.2	124887.4
2	min	0.6	21.4	394.8	3922.1	13200.2	32370.2	62982.1
	mean	0.6	34.0	418.1	4051.7	13487.0	32501.5	63040.9
	max	0.7	52.8	455.4	4200.1	13895.4	33655.1	63064.8
3	min	0.5	17.3	228.7	2767.3	9078.4	22559.7	44836.8
	mean	0.5	19.9	277.3	2797.5	9388.3	22887.1	44879.7
	max	0.5	28.5	303.9	2835.6	9505.8	23214.9	45025.6
4	min	0.4	13.3	145.9	2039.9	7085.1	17259.8	33789.5
	mean	0.5	15.8	201.3	2098.8	7137.0	17316.2	33878.0
	max	1.2	20.1	226.0	2133.6	7168.7	17439.2	33999.1
5	min	0.4	9.9	128.2	1649.8	5719.8	13801.4	27288.0
	mean	0.5	12.4	155.5	1688.3	5760.8	13946.5	27372.6
	max	0.5	18.3	181.3	1721.1	5799.5	14012.3	27455.4
6	min	0.4	8.4	95.7	1399.3	4834.4	11669.5	22971.0
	mean	0.5	9.8	123.0	1428.5	4874.0	11726.4	23045.3
	max	0.5	14.4	139.9	1453.7	4901.9	11763.0	23118.2
7	min	0.6	13.5	128.9	1658.0	5715.7	13996.4	26871.5
	mean	0.6	16.8	179.1	1696.5	5789.1	14154.2	27286.7
	max	0.6	24.5	215.2	1737.1	5838.7	14330.6	27482.0
8	min	0.6	11.8	128.8	1451.1	5029.8	12309.2	23810.4
	mean	0.6	14.8	158.0	1486.7	5107.5	12474.3	24070.9
	max	0.6	18.3	187.4	1524.0	5160.1	12568.0	24277.3
9	min	0.6	10.3	108.8	1291.9	4485.5	11060.7	21554.9
	mean	0.6	13.1	148.9	1324.0	4561.6	11165.8	21680.8
	max	0.6	15.5	181.8	1360.8	4629.3	11221.5	21796.6
10	min	0.6	9.7	109.6	1250.8	4502.2	11226.0	21312.5
	mean	0.6	11.6	145.5	1324.7	4570.8	11305.2	21628.3
	max	0.7	16.5	164.9	1380.9	4724.7	11615.1	21983.4
11	min	0.6	8.9	100.0	1221.7	4257.5	10681.8	20352.5
	mean	0.7	10.5	129.2	1264.0	4328.6	10805.6	20497.7
	max	1.6	15.4	166.3	1342.9	4395.5	11037.6	20696.0
12	min	0.7	8.3	55.7	438.0	1417.4	3454.3	6505.9
	mean	0.7	8.9	59.6	446.5	1427.8	3492.9	6517.1
	max	1.6	11.9	63.6	476.2	1437.7	3550.5	6546.8

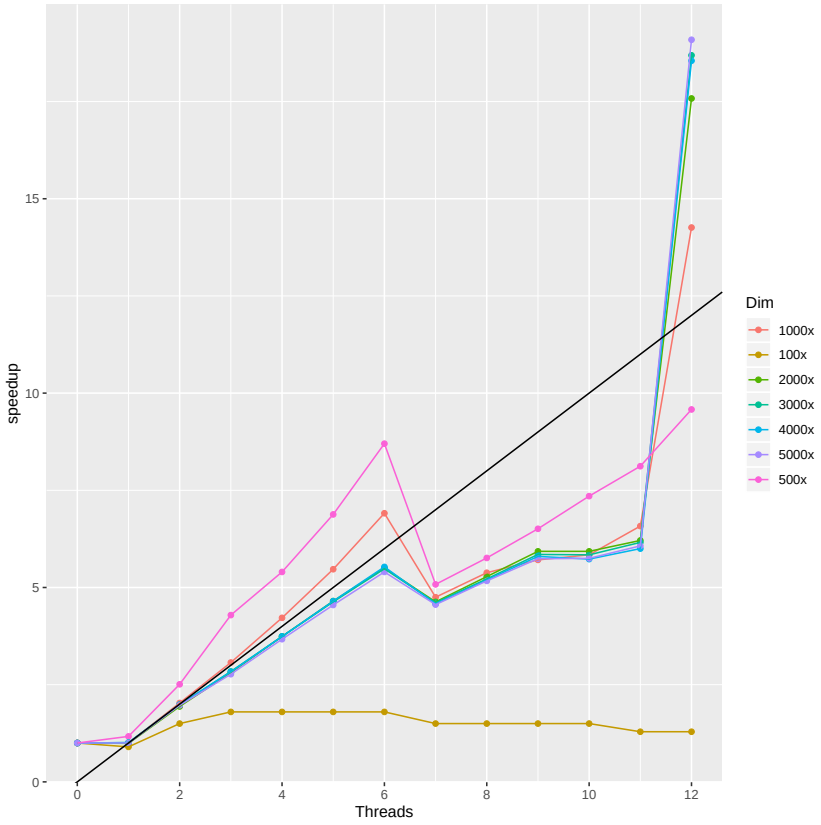

Figure 2.17: Speedup obtained with matrix multiplication using row/row program using OpenMP default, private and shared clauses

2.3.4 Matrix multiplication using vector matrix storage

A further modification of the matrix multiplication was to use the row/row storage of the matrices but address the matrices as vectors as in the program of Figure 2.13. However, to do this the two lines:

```
#pragma omp parallel default(none) private(i, j, k, dot)
shared(a, b, c)
```

were replace by:

```
#pragma omp parallel default(none) private(i, j, k, dot, x, y)
shared(b, z, slice)
```

in the code in Figure 2.13. In this construction `x, y` and `z` replace the matrices `a, b` and `c` which they point to. The matrix `b` is shared due to it's use in the assignment to the variable `y` within the parallel directive.

Table 2.8 contains the execution times measured when the `default`, `private`, and `shared` clauses of Section 2.3.2 were used in the matrix multiplication program of Figure 2.13 for seven matrix sizes. Figure 2.18 shows plots of the execution speedup for each of those search intervals computed from those tabled values.

Table 2.8: Execution times of vector square integer matrix multiplication using OpenMP directives with clauses

Threads		Array Dimension						
		100	500	1000	2000	3000	4000	5000
0	min	0.9	48.4	720.7	7350.6	25565.0	62254.6	121066.8
	mean	0.9	86.1	834.6	7615.6	26125.3	62955.7	122411.9
	max	1.0	99.0	881.8	7672.8	26235.0	63174.6	122825.2
1	min	1.0	41.6	778.7	7536.0	26432.6	64568.5	123727.8
	mean	1.1	66.1	876.7	7899.7	26704.8	64786.6	124039.0
	max	1.1	112.4	916.6	7987.3	26739.4	66849.2	124816.0
2	min	0.6	27.9	382.4	3971.7	13438.9	32510.5	62633.4
	mean	0.7	41.0	424.2	4192.3	13658.9	32784.0	63102.4
	max	1.0	53.5	453.8	4267.6	13832.8	33666.1	63933.3
3	min	0.5	21.3	227.9	2753.2	9092.7	22879.1	44661.0
	mean	0.5	25.8	283.3	2803.6	9405.8	22951.8	44773.0
	max	0.6	36.9	304.4	2837.5	9563.8	23154.6	45104.7
4	min	0.5	10.3	174.7	1986.6	7060.5	17206.1	33823.7
	mean	0.5	14.8	199.6	2082.6	7126.2	17267.3	33917.7
	max	0.5	18.7	229.3	2137.2	7186.9	17368.5	34077.2
5	min	0.4	10.3	136.8	1578.5	5667.8	13875.8	27107.1
	mean	0.5	11.6	165.3	1672.3	5765.3	13956.6	27349.4
	max	0.5	18.5	183.2	1720.0	5793.9	14010.3	27493.2
6	min	0.4	8.4	96.2	1375.7	4830.8	11684.3	23010.9
	mean	0.5	10.8	130.0	1413.9	4868.8	11722.8	23090.0
	max	0.5	16.1	152.5	1450.7	4908.9	11765.6	23120.1
7	min	0.5	13.1	133.7	1629.7	5590.2	13925.6	26860.0
	mean	0.6	16.3	172.7	1684.0	5716.6	14056.6	27074.9
	max	0.6	20.0	197.9	1735.5	5785.2	14135.1	27217.4
8	min	0.6	11.9	114.4	1447.9	4875.3	12355.4	23740.0
	mean	0.6	14.0	154.6	1481.3	5050.3	12522.1	23873.0
	max	0.6	17.2	180.0	1517.4	5124.1	12441.3	24040.1
9	min	0.6	10.2	125.6	1252.1	4472.4	11020.8	21265.9
	mean	0.6	12.2	148.4	1309.2	4529.3	11127.0	21499.8
	max	1.5	16.9	172.3	1362.8	4579.8	11340.0	21661.7
10	min	0.6	9.3	110.6	1271.5	4422.2	11016.6	21121.8
	mean	0.6	12.6	134.6	1309.5	4524.3	11204.2	21443.2
	max	0.7	15.8	168.4	1343.4	4587.1	11526.1	21575.1
11	min	0.6	8.7	97.9	1161.6	4110.7	10566.7	20159.7
	mean	0.7	10.1	125.7	1228.3	4284.3	10625.3	20339.6
	max	1.1	13.8	150.6	1273.8	4367.7	10697.4	20557.3
12	min	0.7	8.1	56.6	430.3	1382.2	3403.2	6335.8
	mean	0.7	9.4	59.3	433.9	1406.5	3469.1	6486.3
	max	1.5	12.4	62.6	437.3	1446.9	3662.7	6759.5

Comparing the execution times in Table 2.8 with those in Table 2.4 when no clauses were used with the `parallel` OpenMP directive, and the corresponding speedup plots of Figure 2.18 and Figure 2.14. In both comparison pairs no substantial change was obtained by using the `default`, `private` and `shared` clauses.

The `default` clause was remove from the code and a number of *spot checks* on the execution time of the resulting executable were performed. All those check times fell within the corresponding `min` to `max` limits of that executable in Table 2.8.

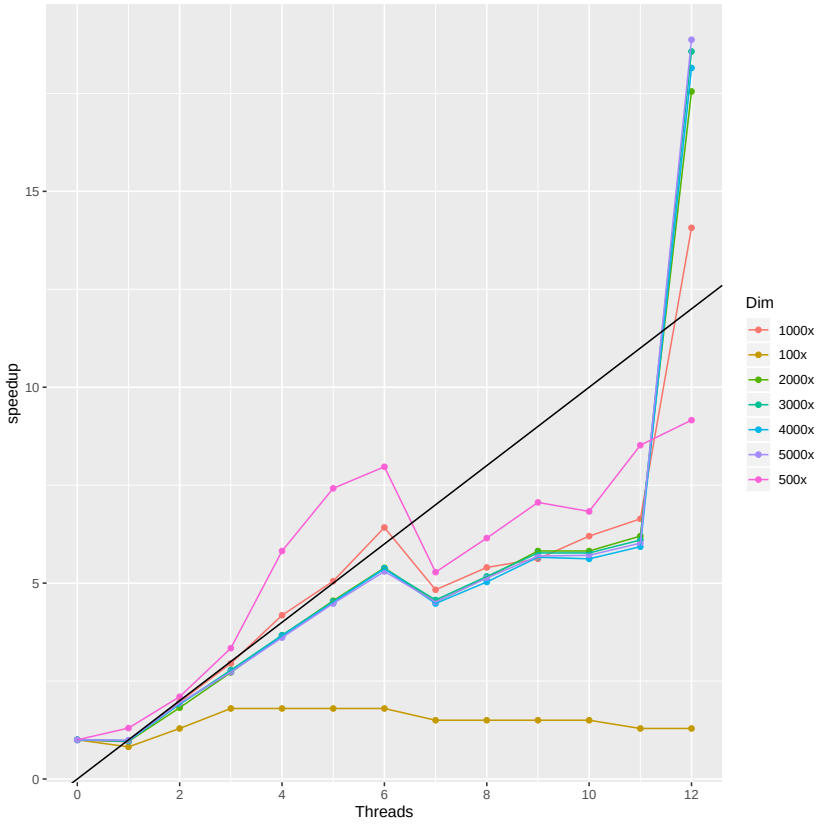

Figure 2.18: Speedup obtained with matrix multiplication vector program using OpenMP default, private and shared clauses

2.3.5 Overview of adding simple clauses to `parallel` directive

For all examples considered, four showed no substantial change to execution time nor speedup as a result of adding `default`, `private`, and `shared` clauses to the OpenMP `parallel` directive.. In the remaining example both the execution time and speedup improved. This example used row/column storage of the matrix being multiplied. Is this an indication of a data handling affect?

In all cases, the `default` clause appeared to have no effect.

Chapter 3

Vectorization

Vectorization is also know as SIMD (Single Instruction Multiple Data). It is a property of the hardware. A vector, or collection, of data containing multiple individual scalar values is processed simultaneously by a single instruction. Like in the case of multi-core processing, vectorization requires a capacity in the compiler to generate appropriate executable code and hardware present to execute such code. Vectorization is aimed at reducing the execution time of programs in which it is used. As such it is a form of optimization.

This chapter considers how the programmer approaches vectorization making the assumption the compiler is able to handle vectorization.

Table 3.1: Mac Pro computers used

	trash can	cheese grater
year	2013	2019
processor	Xeon E5-1650	Xeon W
clock	3.5 GHz	2.5 GHz
turbo	3.9 GHz	4.4 GHz
memory	16 GB	32 GB
cores	6	28
L3 cache	12288 KB	66.5 MB
graphics	FirePro D500 3 GB	Radeon Pro 580X 8 GB
storage	256 GB	256 GB
hi Intrinsic	AVX	AVX-512_VNNI
OS	deb bulleye/sid	deb bulleye/sid
compiler	gcc 9.2.0	gcc 9.2.0

The compiler used was gcc version 9.2.0. Two Mac Pro computers running Debian Bullseye/sid Linux were used the details of which are in Table 3.1. Two computers demonstrate those aspect of the study which have changed due to advancement in technology.

3.1 Automatic optimization/vectorization via compiler options

The simplest way of approaching optimization is to use the compiler. What is achieved depends on the capability of that compiler. This is a good first approach. This section looks at that approach.

By different combinations of command line options the compiler can be asked to optimize the executable code generated from the source code presented. Optimization is generally aimed at producing executable code which run faster. Some optimization can be achieved through better arrangement of code, but optimization using vectorization requires appropriate hardware to be available on the executing computer. But some source code can be vectorized and others cannot.

Table 3.2 list the command line options considered here. These options are proposed to influence the

execution time of the executable code produced. Handling of OpenMP directives inserted in the source code by use of the `-fopenmp` directive is included. Vectorization is one means by which OpenMP can optimize executable code. In gcc handling of OpenMP, the `-fopt-info-vec` option gives information on vectorization at compile time. The `-lgomp` option causes linking of the executable to the `gomp` library which provides OpenMP functions when OpenMP is not directly included, for example, the `omp_get_wtime()` used to measure the execution time of the code. The `Key` given links a command line to a result. The resulting executables were executed by the command line:

```
./abc x
```

where `x` is a command line option used only by an executing OpenMP program. The executions times were recorded correct to one decimal place. The purpose of this decimal place figure was to indicate the *stability* of the last digit before the decimal, i.e. the unit value of the result. The `-march` option indicates the computer architecture for which the executable code is to be optimized for use.

Table 3.2: Command line optimization/vectorization options

Key	gcc command
A	gcc -O2 -lgomp -o abc abc.c
B	gcc -O3 -lgomp -fopt-info-vec -march=native -o abc abc.c
C	gcc -O3 -lgomp -fopt-info-vec -o abc abc.c
D	gcc -O3 -lgomp -fopt-info-vec -fno-tree-vectorize -o abc abc.c
E	gcc -O2 -fopenmp -fopt-info-vec -o abc abc.c
F	gcc -O2 -fopenmp -fopt-info-vec -march=native -o abc abc.c
G	gcc -03 -fopenmp -fopt-info-vec -fno-tree-vectorize -march=native -o abc abc.c
H	gcc -O3 -fopenmp -fopt-info-vec -march=native -o abc abc.c

Messages resulting from the `-fopt-info-vec` compiler option contains useful information. Such messages relate to automatic detected vectorization. An example of such a message is:

```
37:7: optimized: loop vectorized using 16 byte vectors
```

where 37 refers to the line number of the source code containing the loop which was implemented using machine vector instructions of the computer.

Table 3.3 shows the number of messages produced by `gcc` for the compiler options with the `Key` given in Table 3.2 when used on the codes in Figures 3.1 and 3.2. The same results came from the two Mac Pros used. Under the heading `Compiler opt messages` are the message numbers produced at compile relating to optimization which the compiler had performed. In the right hand columns number the messages produced by the executing program resulting from that compile. Using different compiler options is seen to have different outcomes.

Table 3.3: Indicators of compiler and executable performance by messages produced

	Compiler opt message		Output lines produced	
Key	primes	matrix	primes	matrix
A	none	none	2	1
B	2	1	2	1
C	1	1	2	1
D	none	none	2	1
E	none	none	3	2
F	none	none	3	2
G	none	none	3	2
H	2	1	3	2

Two programs, each of a different type, were used. The sieve program of Figure 3.1 is meant as an example related more to data handling as it determines the prime numbers in the interval 2 to 10^8. The program of Figure 3.2 calculates the matrix product of two square matrices of dimension 1000 and is an example of intensive arithmetic.

Notice that the print output of the programs in Figures 3.1 and 3.2 is not included in the execution timing of the program. In each program the OpenMP function `omp_get_wtime()` was used for timing. When the code was not compiled with OpenMP, this function was linked via the `-lgomp` compiler library option.

In both Table 3.4 and 3.5 the column headings 1, 12, and 56 denote the number of threads in use to produce the runtime data in the corresponding column. The Key corresponds to that of Table 3.2. So Tables 3.4 and 3.5 show the influence of compiler options on the subsequent runtime of the executable code resulting. Execution was measured on both the *trash can* (Mac Pro 2013) and *cheese grater* (Mac Pro 2019). The threads in use indicate the influence they had with respect to the compiler options used.

3.1.1 Automatic vectorization of prime number finding

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

#define N 100000000
#define S (int)sqrt(N)

long int primes[N];

int main(int argc, char** argv)
{
  long int i, j;
  int count;
  double ticks;

#ifdef _OPENMP
  printf("threads = %s  N = %d\n", argv[1], N);
  omp_set_num_threads(atoi(argv[1]));
#endif
  ticks = omp_get_wtime();

#pragma omp parallel
#pragma omp for
  for (i=2; i<N; i++) primes[i] = 1;

#pragma omp parallel
#pragma omp for
  for (i=2; i<N; i++)
    if ( primes[i] == 1 )
      for (j=i*i; j<N; j+=i)
        primes[j] = 0;

  ticks = omp_get_wtime() - ticks;
  printf("Elapse time: %lf milli-sec\n", ticks*1000.0);

  count = 0;
#pragma omp for
  for (i=2; i<=N; i++) if ( primes[i] == 1 ) count++;
  printf("prime count = %d\n", count);

  return(0);
}
```

Figure 3.1: Code used to find prime numbers

The data of Table 3.4 was obtained using the code of Figure 3.1. This data shows little effect of compiler option (Key) and a small effect of the number of threads used, where appropriate. OpenMP directives were include in all code compiled, but Key A to D the compiler was not directed to consider such directives. So no thread use effect would be expected. However, Table 3.3 indicates the compiler when using Key B, optimized two loops (indicated by two optimization messages). But the corresponding execution times show no effect. With Key E to G, OpenMP was brought into the compiler's consideration, but Table 3.3 indicates no optimization was performed. The corresponding results in Table 3.4 are consistent with those directives. With Key H optimization was performed consistent with the OpenMP compiler option. Table 3.4 show a slight decrease in execution time. But in the case of the *cheese grater* Mac Pro that reduction did not continue when 56 threads were in use.

Table 3.4: Behaviour of the prime number program on *trash can* and *cheese grater*

		trash can		cheese grater		
		for		for		
Key		1	12	1	12	56
A	min	2515.4	2480.6	1355.5	1356.1	1354.9
	mean	2584.9	2605.6	1368.1	1368.8	1366.2
	max	2648.5	2658.5	1391.4	1388.8	1373.1
B	min	2510.4	2541.1	1353.6	1355.1	1354.2
	mean	2591.5	2601.3	1364.8	1364.9	1363.3
	max	2627.0	2651.3	1378.2	1381.7	1375.9
C	min	2520.3	2503.9	1343.0	1343.9	1343.5
	mean	2600.3	2574.3	1354.0	1356.5	1354.8
	max	2651.1	2653.4	1372.3	1374.0	1374.1
D	min	2515.5	2466.7	1357.3	1353.5	1355.6
	mean	2587.7	2558.6	1370.4	1368.9	1368.9
	max	2662.3	2605.5	1408.6	1389.6	1390.7
E	min	2498.4	2355.8	1351.1	1202.7	1210.2
	mean	2599.9	2384.3	1361.1	1210.1	1221.2
	max	2666.6	2411.6	1385.6	1221.3	1236.8
F	min	2455.2	2331.0	1340.4	1199.7	1208.5
	mean	2551.6	2372.7	1355.8	1211.6	1218.4
	max	2663.3	2394.3	1376.5	1225.6	1232.9
G	min	2496.0	2351.4	1343.1	1198.0	1207.8
	mean	2556.4	2387.3	1358.4	1208.8	1218.3
	max	2614.1	2422.6	1393.1	1226.4	1234.9
H	min	2476.3	2332.7	1346.4	1199.4	1208.1
	mean	2560.7	2358.9	1363.6	1212.0	1220.2
	max	2626.3	2409.5	1386.8	1228.7	1239.8

Overall the data obtained can be summarized as:

- The number of thread used had no significant effect on the execution time.
- The compile options options had no significant effect on the execution time.
- The *cheese grater* had approximately half the execution time of the *trash can*.
- Behaviour of both Mac Pros was the same across thread usage and compiler options used.

The general lack of improvement in execution times for the code of Figure 3.1 suggests the portion of the code requiring the most processing time was not optimized for parallel execution. Although some parts were executed using multiple threads, those parts represented only a small part of the processing effort. Those parts were those the compiler could easily identify. How can the compiler be assisted in dividing the other parts of the program into parallel execution portions?

3.1.2 Automatic vectorization of matrix multiplication

The matrix multiplication code of Figure 3.2 behaves differently to the finding prime number code of Figure 3.1. This is shown in the runtime data of Table 3.5. Matrix multiplication is arithmetic intensive.

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

#define DIM 1000

int a[DIM][DIM], b[DIM][DIM], c[DIM][DIM];

int main(int argc, char * argv[])
{
  int i, j, k;
  int dot;
  int threads;
  double ticks;

#ifdef _OPENMP
  printf("threads = %s  DIM = %d\n", argv[1], DIM);
  omp_set_num_threads(atoi(argv[1]));
#endif

  srand(time(NULL));
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++) {
      a[i][j] = rand() % 100;
      b[j][i] = rand() % 100;
    }

  ticks = omp_get_wtime();
#pragma omp parallel
#pragma omp for
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
    {
      dot = 0;
      for (k=0; k<DIM; k++)
         dot += a[i][k]*b[j][k];
      c[i][j] = dot;
    }
  ticks = omp_get_wtime() - ticks;
  printf("Elapse time: %lf milli-sec\n", 1000.0*ticks);
  return 0;
}
```

Figure 3.2: Code used to multiply two square matrices together

The initialization of the matrices is not included in the timings recorded in Table 3.5. The two matrices are square and of dimension 1000.

In the code of Figure 3.2 both matrices are stored by rows. This assists both serial and parallel processing as the rows which are processed together are easily identified. To use this technique, the second matrix in the multiplication is stores as it's transpose. In the code of Figure 3.2 the storage of the matrices is done in this way using random integer values in the range 0 to 99. As a result all values are positive but positive/negative arithmetic is used which removed the influence that range limitation

might present in the results. The range of random values ensured arithmetic overflow would not occur.

Table 3.5: Behaviour of the matrix multiplication program on *trash can* and *cheese grater*

		trash can			cheese grater		
		for		simd	for		
Key		1	12	1	1	12	56
A	min	1893.9	1850.2	1839.9	346.7	346.4	346.4
	mean	1958.1	1970.7	1948.7	350.5	347.4	348.4
	max	1997.7	2016.2	2025.0	375.5	348.1	375.6
B	min	400.0	345.9	438.6	172.1	173.0	171.9
	mean	463.9	464.5	470.0	174.1	174.6	174.3
	max	499.8	503.3	511.1	175.9	180.2	179.3
C	min	755.6	795.2	736.6	240.5	240.4	240.2
	mean	849.0	862.8	846.9	243.3	241.2	241.9
	max	898.0	907.2	894.2	266.0	242.7	263.5
D	min	1888.6	1869.5	1829.9	346.6	346.6	346.3
	mean	1963.7	1968.9	1952.7	348.2	347.4	348.1
	max	2013.1	2025.0	2013.2	367.1	348.4	370.2
E	min	1867.1	99.6	1894.3	691.3	84.2	51.1
	mean	1971.0	103.4	1939.2	694.0	91.5	52.4
	max	2005.9	107.7	1995.0	717.2	105.9	54.7
F	min	1872.2	99.5	1859.0	691.0	83.0	50.9
	mean	1968.3	103.1	1949.8	693.2	92.4	52.1
	max	2014.8	107.5	2001.5	716.5	103.6	55.2
G	min	1846.6	98.6	1813.0	691.2	85.2	50.7
	mean	1951.3	103.5	1936.9	691.9	97.3	52.2
	max	2020.5	106.9	1999.5	693.0	105.8	55.8
H	min	372.6	30.1	400.7	172.3	26.3	14.1
	mean	488.8	32.2	460.8	174.7	26.9	14.9
	max	524.6	36.3	511.0	181.8	27.8	17.0

The matrix multiplication performed in the code of Figure 3.2 and thus the results in Table 3.5 apply to single precision (32 bit) integer arithmetic.

Overall the data contained in Table 3.5 can be summarized as:

- For Key A, C, and D the number of threads did not have a significant influence on the execution time. The effect in Kay B was marginal. This was the same on both computers.
- Only the use of one thread was measured when the `simd` directive was substituted for the for statement in the code of Figure 3.2. This gave execution times the same as when using one thread alone in a for statement.
- The execution times using Key B, C, and H were significantly better on both computers than using the other Keys.
- Execution times on the *cheese grater* were significant faster than those from the `trash can` by a factors in the range 2.6 to 5.6 depending on the compiler Key used.
- Use of Key B in place of Key A, D, E, F, and G gave reduce execution time.
- Use of Key C gave similar, but slightly reduced, improvement in execution time. However, using Key B or C did not change the execution time as more threads were put into use.
- Key E, F, G, and H led to executable code with reduced execution time with increase in thread use.

These results show better one thread performance for Key B and H compared with Key A, and C to G than for the prime number program, the data for which is in Table 3.4. The influence of thread when

using Key H is also enhances which suggest the loops in the program of Figure 3.2 were handled better than source code of Figure 3.1 for generating parallel execution code by the compiler.

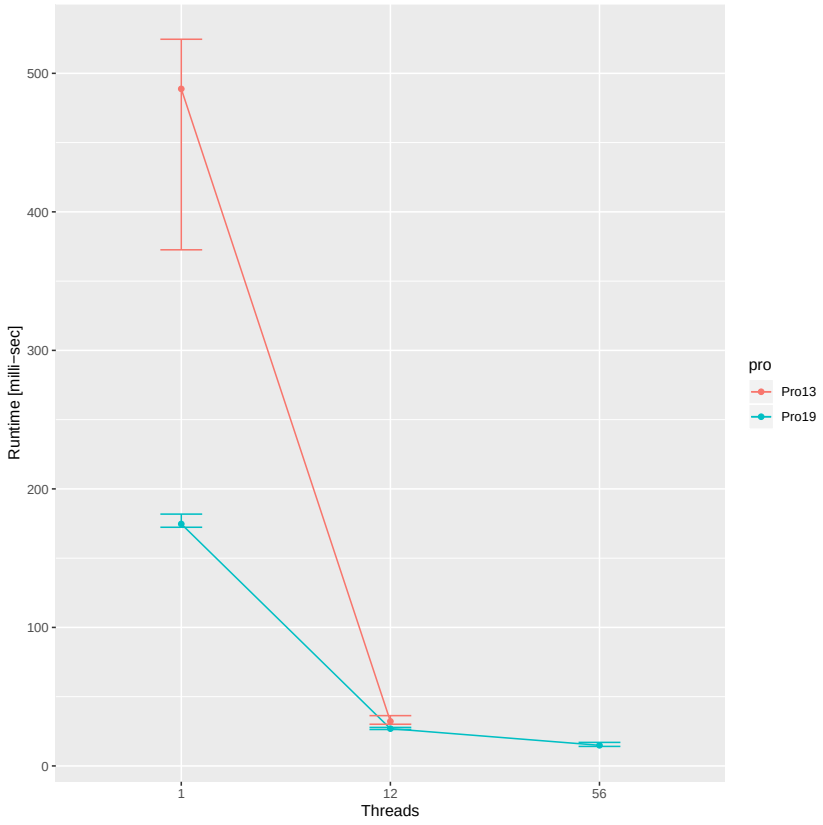

Figure 3.3: Execution time verses thread use on two Mac Pros using Key H compiling

Figure 3.3 is a plot of the execution times for the two Mac Pros using the Key H compiler options. In both Mac Pro cases execution time reduces considerably reaching approximately the same execution time using the same number of threads (12). This is most significant for the Mac Pro 2013 which had a greater execution time with a single thread. The speed up achieved by the Mac Pro 2013 from using 1 thread to using 12 threads was 15.3 while for the Mac Pro 2019 it was 6.5. In the case of the Mac Pro 2019 the execution time further reduces by using the additional threads available, but not at the same rate. For the Mac Pro 2019 the speed up from using 1 thread to using 56 threads was 11.7. For this compiler option (Key) the Mac Pro 2019 achieved a final execution time better than half that obtained by the Mac Pro 2013.

3.1.3 Overview and summary

The compiler optimization messages resulting from the `-fopt-info-vec` option indicate vectorization was performed by the compiler. In the case of Key B this automatic vectorization was without multi-thread execution while with Key H both were present. The results in Table 3.4 and 3.5 indicate the advantage which vectorization can add. The `simd` OpenMP directive used in the prime number code of Figure 3.1 had little effect despite it calling for vectorization to be performed.

3.2 Learning Intel Vectorization

Can manual creation of vectorized code assist in decreasing the overall execution time of a piece of code?

To answer this question requires knowing how vectorization is performed and being able to write vectorized code using such knowledge. Modern Intel hardware can perform vectorization. Since 1997 Intel has provided extensions to hardware they have marketed which provides both registers and instructions which can perform single actions on multiple data (SIMD). However, it is not a straight forward technique to use these enhancements, it being complicated by a number of factors. Each release had deficiencies in parts and scope, with subsequent releases filling in previous deficiencies. Hardware of a particular age will support the releases made when it was marketed, but not latter releases. As shown in Table 3.1 this has an effect on the Mac Pro following from their release.

Released with the vectorization hardware was a library of functions called the *Intrinsics*. These allowed access by, say, a C program to the Intel vectorization capability without using assembler. It is this approach which is used here.

All the processing actions available are performed with respect to registers. Each such register is an array of bits. The number of bits in the register is it's Vector Length. Those bits are subdivided into combinations representing different scalar variable types such as 32 bit integers, 32 bit floating point, 64 bit floating point, etc. At any one time a register can only hold one of the scalar types. The number of registers is more of assembler programming concern, not some much when using Intrinsics.

Learning to use Intrinsics to write Intel vectorization programs requires:

- understanding the registers both how they are packed and how to get data into and out of them,
- Intrinsic functions and to which release they relate,
- the logic behind the naming of the individual Intrinsic functions for this helps understanding of their capability, and
- combining the Intrinsic functions available on the hardware to achieve the required goal.

Writing functing vector code using Intel Intrinsics is the purpose of this section. The goundwork laid down here is used in subsequent sections.

3.2.1 Evolution

In the beginning there was only serial execution hardware on a PC. In 1997 Intel introduced the MMX instruction extension set for the IA-32 and x86-64 hardware to assist multimedia processing. Such processing was also useful for other intensive numerical computation. The extensions used 8 64-bit registers which could be divided into 2 32-bit integers, 4 16-bit integers, or 8 8-bit integers. Notice: integers only. It used the existing floating point hardware registers so using floating point and these extensions together could not happen. Although this hardware has today mostly been replaced, the legacy of MMX continues in subsequent vectorization releases.

Table 3.6 tabulates the evolution history of Intel Intrinsics. In the table the notation 8/16. . . indicates 8-bit, 16-bit, etc. If in the integer column it indicates the size of integer support introduced by the release; if in the float column then it indicates the size of individual floating point value support introduced by the release. The size of the integer/float value divided by the vector length of the register gave the number of that particular scalar housed in the register (vector) and thus processed by the subsequent vector instruction or instructions. After the AVX release, the header file required to use an Intrinsics release remained as `immintrin.h`. Once a feature was introduced in a release it remained active in

subsequent releases. Each release introduced new processing instructions, but not necessarily all parts. The missing parts occurred in subsequent releases.

Table 3.6: Changes introduced by each Intel Intrinsics release

Release	Intrinsics header file	Storage prefix	Function name	Register name	Vector length	Data handled	
						integer	float
MMX	mmintrin.h	_m64	_mm	MMX	64	8/16/32	
SSE	xmmintrin.h	_m128	_mm	XMM	128		32
SSE2	emmintrin.h					8/16/32/64	64
SSE3	pmmintrin.h						
SSSE3	tmmintrin.h						
SSE4.1	smmintrin.h						
SSE4.2	nmmintrin.h						
AVX	immintrin.h	_m256	_mm256	YMM	256	32	32/64
AVX2	immintrin.h					8/16/32/64	
AVX-512F	immintrin.h	_m512	_mm512	ZMM	512	32/64	32/64
AVX-512BW						6/16	
AVX-512CD							
AVX-512DQ							
AVX-512VL							

Except for the MMX release where the MMX registers were the pre-existing floating point registers of the architecture, releases shown in Table 3.6 added purpose built vector registers to the hardware. Significant releases also increased the vector length (the width of the register). This increased length was achieved by adding to the register or the previous release. So the YMM register contained the XMM registers as their low order bits, and the ZMM registers contained the YMM registers as their low order bits. When the XMM registers were introduced in release SSE their low order bits became the MMX registers, replacing their initial cover. Figure 3.4 diagrammatically represents the evolution of Intel vector processing by Intrinsics.

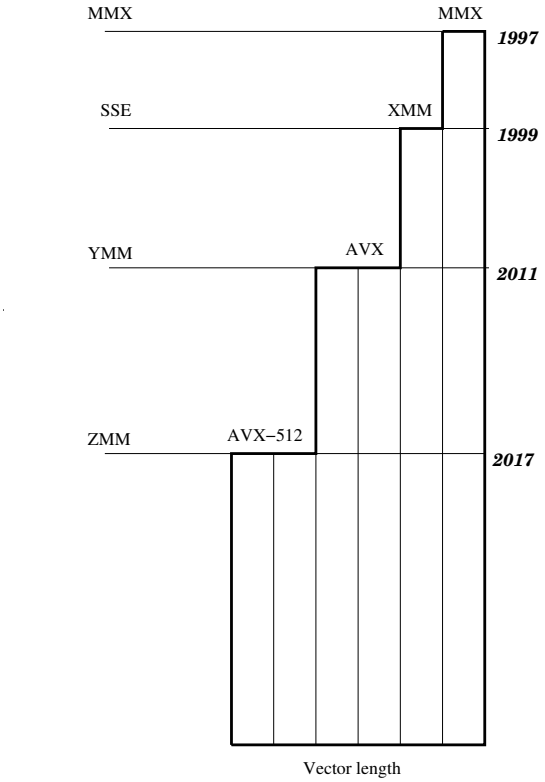

Figure 3.4: Evolution of Intel Intrinsics and their contents

3.2.2 A primer on Intrinsics functions

The general form of an Intrinsics function call is:

__m<**return type**> _mm<**vector length**>_<**operation**>_<**operation data type**> (**arguments**) >

which consists of two parts. On the left of the space is the type of the data to be returned. On the right is the Intrinsics function call itself which is to produce that return. Note in this general form there is highlighted and non-highlighted portions. The highlighted parts build up what the function is to perform. The non-highlighted parts are character which appear in the call. The ;ess than sign (<) and greater than sign (>) are not part of the syntax but are used here to delimit the different parameters in the form. By contrast, the underscore character (_) is part on the syntax.

The retunrn data types are:

parameter	meaning
64	64 bit integer
128	128 bit float
128d	128 bit double precision float
128i	128 bit integer
256	256 bit single precision float
256i	256 bit integer
256d	256 bit double precision float
512	512 bit single precision float
512i	512 bit integer
512d	512 bit double precision float
ask8	8 bit mask vector

In most cases this is the data type of the `destination` of the Intrinsics function. In a lot of cases it corresponds to the type of the arguments of the Intrinsics call. **Note** there are two underscore characters (_) in front of the single `m` character. A variable of the required type is assigned in the program.

Construction of the Intrinsics function name is from three parts. The vector length is taken from the set:

parameter	meaning
	64-bits or 128-bits
256	256-bits
512	512-bits

and indicates the vector length (or width) of the register, or registers, upon which the operation of the function is performed. There are many operations defined, here are several examples:

parameter	meaning
extractf64x4	get 4 off 64-bit floats
loadu	load from memory maybe not on correct boundaries
add	add two vectors
reduce_sum	sum all elements in the vector
mul	multiply two vectors
setz	set all elements of a vector to zero
storeu	store vector into memory maybe not on correct boundaries
set	set all elements of a vector to a particular value
sub	subtract two vectors
castpd512	get first 4 64-bit floats from a vector
cmp	compare two vectors

Finally, there is the data type contained in the register which is to be operated upon. These are"

parameter	meaning
ps	32 bit float
dp	64 bit float
pu8	unsigned 8 bit integer
pu16	unsigned 16 bit integer
pi8	8 bit signed integer
pi16	16 bit signed integer
pi32	32 bit signed integer
epi8	8 bit integer
epi16	16 bit integer
epi32	32 bit integer
epi64	64 bit integer

Note the name of the Intrinsics function (as shown in the above general form) has one underscore character ((_)and two m characters at the head of it's name. Once an Intrinsics function is formulated the argument to be passed to it follow.

One of the problems in using the Intrinsics functions is their evolution depicted in Table 3.6 and Figure 3.4. Each release was not always complete with missing functionality being added in later releases. But the hardware available determines what Intrinsics functions could be used. Figure 3.7 gives an indication of the result. The table also shows complete examples of Intrinsics function names, but without the arguments.

With respect to Table 3.7 notice the use of _mullo_ in place of _mul_ for the latter is used with 64-bit integers while the former stores only the low 32-bits of each product. If the intent is to multiply 32-bit integers and get 32-bit integer results, the former is used.

In Table 3.7 it is also important to note the following. The distribution shown of Intrinsics relates to genuine Intel hardware. At the time of this work Intel led the introduction of Intrinsics and their related hardware over that from AMD. Each operation entry has three parts; the first inline relates to use of a 128-bit register, the second line to a 256-bit register, and the third line to a 512-bit register. An * indicates the corresponding Intrinsics does not (currently) exist.

There are three steps involved in using Intrinsics function is a program:

- Getting the data into the Intrinsics registers;
- Processing that data; and
- Extracting the results.

Implementing each of these steps may require one or more Intrinsics function calls.

With the early Intrinsic function releases it was necessary to align memory addresses on scalar boundaries for transfer to or from the Intrinsics registers. For example, a 32-bit float address had to be a multiple of 32, while a 64-bit float had to have an address which was a multiple of 64. This requirement applied for taking data to, and from memory and the registers. The functions of recent releases have relaxed those constraints. However, Intrinsics aligned to a hardware release can also use Intrinsics of earlier releases. The boundary alignment may not be required on the latest releases, but care is needed if earlier release Intrinsics are used.

Programming with Intrinsics is like assembler programming, but it is also different. Both deal with registers. With assembler, use of registers is controlled by the programmer. For example, the contents of register 2 is going to be used later, so registers 5 and 12 will be used for performing the next addition operation, leaving the data in register 2 untouched. The programmer provides that insight. With

Table 3.7: A distillation of a selection of Intel intrinsics

Operation performed	32-bit floating point		32-bit integer	
	Intrinsic	Release	Intrinsic	Release
load array	_mm_loadu_ps	SSE	_mm_loadu_epi32	AVX-512VL
	_mm256_loadu_ps	AVX	_mm256_loadu_si256	AVX
	_mm512_loadu_ps	AVX-512F	_mm512_loadu_si256	AVX-512F
load values	_mm_setr_ps	SSE	_mm_setr_epi32	SSE2
	_mm256_setr_ps	AVX	_mm256_setr_epi32	AVX
	_mm512_setr_ps	AVX-512F	_mm512_setr_epi32	AVX-512F
load selective	_mm_maskz_loadu_ps	AVX-512VL	_mm_maskz_loadu_epi32	AXV-512VL
	_mm256_extractf128_ps	AVX	_mm256_extracti128_si256	AVX2
	_mm512_maskz_loadu_ps	AVX-512	_mm512_maskz_loadu_epi32	AVX-512F
add	_mm_add_ps	SSE	_mm_add_epi32	SSE2
	_mm256_add_ps	AVX	_mm256_add_epi32	AVX2
	_mm512_add_ps	AVX-512F	_mm512_add_epi32	AVX-512F
multiply	_mm_mul_ps	SSE	_mm_mullo_epi32	SSE4.1
	_mm256_mul_ps	AVX	_mm256_mullo_epi32	AVX2
	_mm512_mul_ps	AVX-512F	_mm512_mullo_epi32	AVX-512
multiply & add	_mm_fmadd_ps	FMA	*	
	_mm256_fmadd_ps	FMA	*	
	_mm512_fmadd_ps	AVX-512F	_mm512_fmadd_epi32	KNC
horizontal add	_mm_hadd_ps	SSE3	_mm_hadd_epi32	SSSE3
	_mm256_hadd_ps	AVX	_mm256_hadd_epi32	AVX2
	*		*	
sum all	*		*	
	*		*	
	_mm512_reduce_add_ps	AVX-512F	_mm512_add_reduce_epi32	AVX-512F
zeroize	_mm_setzero_ps	SSE	_mm_setzero_si128	SSE2
	_mm256_setzero_ps	AVX	_mm256_setzero_si256	AVX
	_mm512_setzero_ps	AVX-512	_mm512_setzero_epi32	AVX-512
initial pattern	_mm_set_ps	SSE	_mm_set_epi32	SSE2
	_mm254_set_ps	AVX	_mm256_set_epi32	AVX
	_mm512_set_ps	AVX-512F	_mm512_set_epi32	AVX512F
initial setting	_mm_set1_ps	SSE	_mm_set1_epi32	SSE2
	_mm256_set1_ps	AVX	_mm256_set1_epi32	AVX
	_mm512_set1_epi32	AVX-512F	_mm512_set1_epi32	AVX-512F
copy lower	_mm_extract_ps	SSE4.1	_mm_extract_epi32	SSE4.1
	*		_mm256_extract_epi32	AVX
	*		_mm512_cvtsi512_si32	AVX512
selective out	*		*	
	_mm256_extractf32x4_ps	AVX-512F	_mm256_extracti32x4_epi32	AVX-512F
	_mm512_extractf32x8_ps	AVX-512DQ	_mm512_extracti32x8_epi32	AVX-512DQ
store register	_mm_storeu_ps	SSE	_mm_storeu_epi32	AVX-512F
	_mm256_storeu_ps	AVX	_mm256_storeu_epi32	AVX-512F
	_mm512_storeu_ps	AVX-512F	_mm512_storeu_si512	AVX-512F

Intrinsics there is no control over which register is used. Not even the type of register (MMX, XMM, YMM or ZMM) is selected. These are chosen by the Intrinsics function which the programmer selects. But in that selection the programmer must know which register type is available on the hardware being used, which is demonstrated by the examples in Table'3.7. There are multiple registers of each type on the hardware, but the Intrinsics programmer need not be concerned with this limit. At a superficial level, Intrinsics programming could be likened to inline assembler programming.

There are no Intrinsics function for performing i/o. This must be performed by the C program containing the Intrinsics calls.

3.2.3 Code examples

The `Intrinsics` is a library of function call for handling vector programming. This enables one instruction, or Intrinsics function call, to process multiple data values stored in a register. Real program are need to show use of the Intrinsics library. Here two small, but complete C programs are used as demonstration. Despite their size, each program has multiple parts which demonstrate different aspects of using the Intrinsics library.

Figures 3.5 and 3.6 use Intrinsics in C coding. In Figure 3.5 single precision (32-bit) floating point values are handled. In Figure 3.6 single precision integer values are handled.A Each code is divided into two parts. The top par uses Intrinsics up to and including the AVX release, while the bottom part uses Intrinsics up to and including the AVX-512 release. For the Intrinsics hardware available on the Mac Pros given in table 3.1, the top part of each code executes on the Mac Pro 2013 only, while the whole of the coed executes on the Mac Pro 2019. The Intel Intrinsics release to which each Intrinsics function belongs is shown as a comment on the right-hand side of the statement.

A purpose of having the codes of Figures 3.5 and 3.6 producing the same result is to show the similarity in handling floating point and integer data using Intrinsics. Similar, but not the same.

Each of the codes in Figures 3.5 and 3.6 perform the same foundation operations needed in all uses of Intrinsics functions. Each moves data values between memory and the Intrinsics registers embodied in the Intrinsics data type. Values are printed from memory. Movement of data between these registers and memory is not I/O but internal movement within the computer. This is done by Intrinsics functions. Two methods of moving data into the registers is given in the top part of the code while the analogue of one of those methods is used in the bottom part of the code. The sum of all data in the register is calculated in the bottom part, while the sum of corresponding values in two vectors is calculated in the top part.

A problem occurs when there are fewer values to be processed than positions in the register. This is the *end game*. Handling this situation is made possible using Intrinsics only after release AVX-512. The *end game* occurs in practical programming after all multiples of the Intrinsics register size have been processed and there is data left over. For instance, if there were 19 32-bit integers to be processed using Intrinsics and there is 16 integers which fit into a `_m512i` register, the remaining 3 integers are the subject of the end play. This is a common occurrence in practise.

Two methods of putting data into the Intrinsics registers are used in Figures 3.5 and 3.6. The `_setr` Intrinsics function contains the data as arguments in the call, while the `_loadu` Intrinsics function loads from an array in memory.

In the top part of the Figure 3.5 code there are three pairs of Intrinsics functions putting data into the 256-bit wide registers, thus moving 8 32-bit floating point values. The `_setr` has the data as arguments. There are two pairs each using the `_loadu` Intrinsics function. Each of these two pair addresses the array of data in a different manner. Since all pairs produce the same result of loading floating point values in arrays `af` and `bf` into vectors `av` and bv, any of the pairs could be used.

The vectors are then added together and the result stored in the vector `result`. To check the addition, the first two values in the `result` vector are printed out, Since this vector is not stored in memory, special handling is required to print those values.

The Intrinsics function calls while assign values to `hold` vectors are calculating the sum of all the floating point elements stored in the `av` vector. The total which ends up in the `hold2` vector is move to the memory location `sum` from where the required total is printed.

Then the advantage offered by release AVX-512 of the Intrinsics functions is demonstrated. A 16 element vector is loaded into the `temp1` vector. The first 4 elements of that vector are then printed. That vector is then over-written by a vector with three elements only. The first 4 elements of this vector are printed to indicate that trailing elements beyond the third are zero. The sum total of all (three) elements in this vector are then calculated by a single Intrinsics function call, and the result printed.

```c
/*  setting up floating point avx, performing addition on the
 *  vector, then printing out some results.
 *
 *  Coded by:  Ross Maloney
 *  Date:      February 2020
 *
 *  gcc -mavx512bw quickf.c
 *
 *  -mavx512bw for vector Intrinsics
 */

#include <stdio.h>
#include <immintrin.h>


int main(int argc, char** argv)
{
  float     af[] = {2.3, 5.6, 10.6, -12.6, 10.1, -7.8, 4.5, 12.3}; // 25.0
  float     bf[] = {13.4, 2.3, -12.3, 4.6, 8.9, -90.7, 23.4, 4.9};
  float     big[] = {1.0, 12.0, -33.0, -4.0, 15.0, 26.0, 27.0, 18.0,
                     -1.3, 4.4, 34.2, 6.7, 12.8, 8.2, 56.4, 7.9}; // 191.3
  float     small[] = {17.0, -3.0, -5.6};
  __m512    temp1;
  __m256    result, av, bv;
  __m128    hold1, hold2, hold3;
  float     sum;

  av = _mm256_loadu_ps(af);                                   // AVX
  bv = _mm256_loadu_ps(bf);                                   // AVX
  av = _mm256_loadu_ps( (const float *)&af[0]);
  bv = _mm256_loadu_ps( (const float *)&bf[0]);
  av = _mm256_setr_ps(2.3, 5.6, 10.6, -12.6, 10.1, -7.8, 4.5, 12.3);
  bv = _mm256_setr_ps(13.4, 2.3, -12.3, 4.6, 8.9, -90.7, 23.4, 4.9);
  result = _mm256_add_ps(av, bv);                             // AVX
  float *f = (float *)&result;
  printf("result = %f %f\n", f[0], f[1]);

  hold1 = _mm256_extractf128_ps(av, 0);                       // AVX
  hold2 = _mm256_extractf128_ps(av, 1);                       // AVX
  hold3 = _mm_add_ps(hold1, hold2);                           // SSE
  hold1 = _mm_hadd_ps(hold3, hold3);                          // SSE3
  hold2 = _mm_hadd_ps(hold1, hold1);                          // SSE3
  sum = _mm_cvtss_f32(hold2);                                 // SSE
  printf("1. sum = %f\n", sum);

  temp1 = _mm512_loadu_ps( (const __m512i *)&big[0]);         // AVX-512
  float *ff = (float *)&temp1;
  printf("1. result = %f %f %f %f %f\n", ff[0], ff[1], ff[2], ff[3]);
  temp1 = _mm512_maskz_loadu_ps(7, (const __m512 *)&small[0]); // AVX-512
  float *fff = (float *)&temp1;
  printf("2. result = %f %f %f %f %f\n", fff[0], fff[1], fff[2], fff[3]);
  sum = _mm512_reduce_add_ps(temp1);                          // AVX-512
  printf("1. sum = %f\n", sum);

  return(0);
}
```

Figure 3.5: A C program using Intel Intrinsics to perform simple floating point operations

```c
/*  setting up integer avx, performing addition on the
 *  vector, then printing out some results.
 *
 *  Coded by:  Ross Maloney
 *  Date:      February 2020
 *
 *  gcc -mavx512bw quicki.c
 *
 *  -mavx512bw for vector Intrinsics
 */

#include <stdio.h>
#include <immintrin.h>


int  main(int argc, char** argv)
{
  int       ai[] = {2, 4, 6, 8, 10, 12, 14, 16}; // 72
  int       bi[] = {1, 3, 5, 7, 9, 11, 13, 15};
  int       big[] = {2, 11, 6, -90, 78, 23, -4, 5,
                     12, -56, 34, 23, 9, 14, 98, 7}; // 172
  int       small[] = {45, -12, 7};
  __m512i   temp1;
  __m256i   evens, odds, result;
  __m128i   hold1, hold2, hold3;
  int       sum;

  evens = _mm256_setr_epi32(2, 4, 6, 8, 10, 12, 14, 16);        // AVX
  odds = _mm256_setr_epi32(1, 3, 5, 7, 9, 11, 13, 15);          // AVX
  evens = _mm256_loadu_si256( (const __m256i *)&ai[0]);         // AVX
  odds = _mm256_loadu_si256( (const __m256i *)&bi[0]);          // AVX
  result = _mm256_add_epi32(evens, odds);
  int *i = (int *)&result;
  printf("result = %d %d\n", i[0], i[1]);

  hold1 = _mm256_extracti128_si256(evens, 0);                   // AVX
  hold2 = _mm256_extracti128_si256(evens, 1);                   // AVX
  hold3 = _mm_add_epi32(hold1, hold2);                          // SSE2
  hold1 = _mm_hadd_epi32(hold3, hold3);                         // SSSE3
  hold2 = _mm_hadd_epi32(hold1, hold1);                         // SSSE3
  sum = _mm_cvtsi128_si32(hold2);                               // SSE2
  printf("sum = %d\n", sum);

  temp1 = _mm512_loadu_si512( (const __m512i *)&big[0]);        // AVX-512
  int *ii = (int *)&temp1;
  printf("1. result = %d %d %d %d\n", ii[0], ii[1], ii[2], ii[3]);
  temp1 = _mm512_maskz_loadu_epi32(15,
                                   (const __m512i *)&small[0]); // AVX-512
  int *iii = (int *)&temp1;
  printf("2. result = %d %d %d %d %d\n",
         iii[0], iii[1], iii[2], iii[3], iii[4]);
  sum = _mm512_reduce_add_epi32(temp1);                         // AVX-512
  printf("1. sum = %d\n", sum);

  return(0);
}
```

Figure 3.6: A C program using Intel Intrinsics to perform simple integer operations

These calculations contrast to finding the sum using the `hold` vectors in the code above. The output produced by the code of Figure 3.6 was:

The output produced by the code of Figure 3.5 was:

```
result = 15.700000 7.900000
sum = 25.000000
1. result = 1.000000 12.000000 33.000000 -4.000000
2. result = 17.000000 -3.000000 -5.000000 0.000000
1. sum = 8.400000
```

The same output is produced by each of the three pairs of setting up the vectors.

The code in Figure 3.6 performs the same processing as that in Figure 3.5 but applied to integer data. The top part of the code in Figure 3.6 handles the two arrays `ai` and `bi` which contain 8 32-bit integers. The contents of these arrays are moved into vectors `evens` and `odds` by two pairs of Intrinsics functions. Both pair have the same result. One pair use the `_setr` Intrinsics function with the array values as arguments to the call. The other pair use the `_loadu` Intrinsics call, the argument of which is the address in memory of the array of values. The result from either pair is the same.

The two vectors were then added and the first two elements of the `result` vector in which the sums had been stored were printed to verify correctness.

Using the `hold` vectors the sum of all the values in the `evens` vector is calculated. The result is store in the memory location `sum` by the `_cvtsi128` Intrinsics function. From `sum` the result is printed.

The bottom of the code in Figure 3.6 handles larger vectors. First the `temp1` vector is loaded with 16 32-bit integer values. The first four of the values in the vector are then printed. The vector `temp1` is then loaded with 4 32-bit integers which is less than it's capacity. Printing of the first 5 members of this vector indicates zero have filled to unused places in the vector. All the values in this vector were added and stored in memory location `sum` from where the value was printed.

```
result = 3 7
sum = 72
1. result = 2 11 6 -90
2. result = 45 -12 7 98 0
1. sum = 138
```

With respect to the codes in Figures 3.5 and 3.6 variables are used several times. In Figure 3.5 the variables `av` and `bv` acting as pairs are loaded with values from memory in three different manners. For any one run of this program only one of such pairs were needed. In Figure 3.6 the variables `evens` and `odds` are a pair and each pair is used twice for loading from memory. Again, only one of those pairs was used in a run of the program.

3.3 Processing speed using Intel Intrinsics

To test the effect Intel Intrinsics have on writing, processing, and resulting execution time, two program types were used. One program type performed matrix multiplication of two square matrices. Such a program is arithmetic intensive. The other program searched for the occurrence of a sequence a six integers in a column of integers. This program was taken as less arithmetic intensive but more data movement intensive.

3.3.1 Matrix multiplication

Two matrix multiplication programs were used. In one the matrices were of 32-bit integer values and in the other 32-bit floating point values.

Each of these matrix multiplication programs were made up of three sections. The first section performs the multiplication using standard C statements. The next section combines C statements with Intel Intrinsics library functions with those function calls similar to those tabulated in Table 3.7 being executable on a Mac Pro 2019. The final section replaced the Mac Pro 2019 Intrinsics functions with those available on the Mac Pro 2013. All sections were executable on the Mac Pro 2019, while the first and final sections executed on the Mac Pro 2013. Different Intrinsics functions apply to handling of integers and floating point values.

Figures 3.7 and 3.8 contain the code of those programs. Figure 3.7 lists the integer matrix multiplication code, while Figure 3.8 list the floating point matrix multiplication code.

```
/* Compute the product of two square matrices of integer values using the direct
 * method and Intel Intrinsics which are available on the 2019 and 2013 Mac Pro.
 *
 * Coded by:  Ross Maloney
 * Date:      April 2020
 */

#include <omp.h>
#include <stdio.h>
#include <immintrin.h>
#include <time.h>

#define DIM  1000
#define SIZE 16   // AVX512
#define SIZE 4    // SSE4.1

int am[DIM][DIM], bv[DIM][DIM], result[DIM][DIM];

int main(int argc, char** argv)
{
  double    ticks;
  int       i, j, k, gulp, ii;
  int       value;
  int       result[9][9];
  __m512i   row, column, hold, temp;            // for Mac Pro 2019
  __m128i   row, column, hold, temp, runner;    // for Mac Pro 2013
  int       rowleft[SIZE], columnleft[SIZE];    // for Mac Pro 2013

  srand(time(NULL));
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      am[i][j] = rand() % 100;
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      bv[i][j] = rand() % 100;

  ticks = omp_get_wtime();

  /* Normal processing */

  for (k = 0; k < DIM; k++)
    for (j = 0; j < DIM; j++)  {
      value = 0.0;
      for (i = 0; i < DIM; i++)  value += am[k][i] * bv[j][i];
      result[k][j] = value;
    }
```

Figure 3.7: C program using Intel Intrinsics to perform integer matrix multiplication (continues . . .)

```
 /* Mac Pro 2019 Intrinsics */

for (k = 0; k < DIM; k++) {
  for (j = 0; j < DIM; j++) {
    hold = _mm512_setzero_epi32();                                          // AVX=512F
    gulp  = ( 1 << SIZE ) - 1;
    for (i = 0; i <= DIM - SIZE; i += SIZE) {
      row = _mm512_maskz_loadu_epi32(gulp, (const __m512i *)&am[k][i]);     // AVX-512F
      column = _mm512_maskz_loadu_epi32(gulp, (const __m512i *)&bv[j][i]); // AVX-512F
      temp = _mm512_mullo_epi32(row, column);                               // AVX-512F
      hold = _mm512_add_epi32(hold, temp);                                  // AVX-512F
    }
    gulp = ( 1 << (DIM - i) ) - 1;
    row = _mm512_maskz_loadu_epi32(gulp, (const __m512i *)&am[k][i]);       // AVX-512F
    column = _mm512_maskz_loadu_epi32(gulp, (const __m512i *)&bv[j][i]);    // AVX-512F
    temp = _mm512_mullo_epi32(row, column);                                 // AVX-512F
    hold = _mm512_add_epi32(hold, temp);                                    // AVX-512F
    result[k][j] = _mm512_reduce_add_epi32(hold);                           // AVX-512F
  }
}

 /* Mac Pro 2013 Intrinsics */

for (i = 0; i < SIZE; i++) {
  rowleft[i] = 0;
  columnleft[i] = 0;
}
hold = _mm_setzero_si128();                                                 // SSE2
for (k = 0; k < DIM; k++) {
  for (j = 0; j < DIM; j ++) {
    runner = _mm_setzero_si128();                                           // SSE2
    for (i = 0; i <= DIM - SIZE; i += SIZE) {
      row = _mm_loadu_si128( (const __m128i *)&am[k][i]);                   // SSE2
      column = _mm_loadu_si128( (const __m128i *)&bv[j][i]);                // SSE2
      temp = _mm_mullo_epi32(row, column);                                  // SSE4.1
      runner = _mm_add_epi32(runner, temp);                                 // SSE2
    }
    ii = 0;
    for ( ; i < DIM; i++) {
      rowleft[ii] = am[k][i];
      columnleft[ii] = bv[j][i];
      ii++;
    }
    row = _mm_loadu_si128( (const __m128i *)&rowleft[0]);                   // SSE2
    column = _mm_loadu_si128( (const __m128i *)&columnleft[0]);             // SSE2
    temp = _mm_mullo_epi32(row, column);                                    // SSE4.1
    runner = _mm_add_epi32(runner, temp);                                   // SSE2
    hold = _mm_setzero_si128();                                             // SSE2
    runner = _mm_hadd_epi32(runner, hold);                                  // SSSE3
    runner = _mm_hadd_epi32(runner, hold);                                  // SSSE3
    result[k][j] = _mm_extract_epi32(runner, 0);                            // SSE4.1
  }
}


ticks = omp_get_wtime() - ticks;

printf("Elapse time: %lf milli-sec\n", ticks*1000.0);

return(0);
}
```

Figure 3.7: C program using Intel Intrinsics to perform integer matrix multiplication

```
/* Compute the product of two square matrices of floating point values using the
 * direct method and Intel Intrinsics which are available on the 2019 and 2013
 * Mac Pro.
 *
 * Coded by:  Ross Maloney
 * Date:      April 2020
 */

#include <omp.h>
#include <stdio.h>
#include <immintrin.h>
#include <time.h>

#define DIM  1000
#define SIZE 16     // AVX512
#define SIZE 8      // AVX

float am[DIM][DIM], bv[DIM][DIM], result[DIM][DIM];

int main(int argc, char** argv)
{
  double    ticks;
  int       i, j, k, gulp, ii;
  float     value, result[9][9];
  __m512    row, column, hold, temp;                  // for Mac Pro 2019
  __m256    row, column, hold, temp, runner;          // for Mac Pro 2013
  __m128    hold1, hold2, hold3;                      // for Mac Pro 2013
  float     rowleft[SIZE], columnleft[SIZE];          // for Mac Pro 2013

  srand(time(NULL));
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      am[i][j] = rand() % 100;
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      bv[i][j] = rand() % 100;

  ticks = omp_get_wtime();

  /* Normal processing */

  for (k = 0; k < DIM; k++)
    for (j = 0; j < DIM; j++)  {
      value = 0.0;
      for (i = 0; i < DIM; i++)  value += am[k][i] * bv[j][i];
      result[k][j] = value;
    }

   /* Mac Pro 2019 Intrinsics */

  for (k = 0; k < DIM; k++)  {
    for (j = 0; j < DIM; j++)  {
      hold = _mm512_setzero_ps();                                          // AVX=512F
      gulp  = ( 1 << SIZE ) - 1;
      for (i = 0; i <= DIM - SIZE; i += SIZE)  {
        row = _mm512_maskz_loadu_ps(gulp, (const __m512 *)&am[k][i]);      // AVX-512F'
        column = _mm512_maskz_loadu_ps(gulp, (const __m512 *)&bv[j][i]);   // AVX-512F
        temp = _mm512_mul_ps(row, column);                                 // AVX-512F
        hold = _mm512_add_ps(hold, temp);                                  // AVX-512F
      }
      gulp = ( 1 << (DIM - i) ) - 1;
      row = _mm512_maskz_loadu_ps(gulp, (const __m512 *)&am[k][i]);        // AVX-512F
      column = _mm512_maskz_loadu_ps(gulp, (const __m512 *)&bv[j][i]);     // AVX-512F
      temp = _mm512_mul_ps(row, column);                                   // AVX-512F
      hold = _mm512_add_ps(hold, temp);                                    // AVX-512F
      result[k][j] = _mm512_reduce_add_ps(hold);                           // AVX-512F
    }
  }
```

Figure 3.8: C program using Intel Intrinsics to perform floating matrix multiplication (continues ...)

```
    /* Mac Pro 2013 Intrinsics */

  for (i = 0; i < SIZE; i++)  {
    rowleft[i] = 0.0;
    columnleft[i] = 0.0;
  }
  hold = _mm256_setzero_ps();                                          // AVX
  for (k = 0; k < DIM; k++)  {
    for (j = 0; j < DIM; j ++)  {
      runner = _mm256_setzero_ps();                                    // AVX
      for (i = 0; i <= DIM - SIZE; i += SIZE)  {
        row = _mm256_loadu_ps( (const float *)&am[k][i]);              // AVX
        column = _mm256_loadu_ps( (const float *)&bv[j][i]);           // AVX
        temp = _mm256_mul_ps(row, column);                             // AVX
        runner = _mm256_add_ps(runner, temp);                          // AVX
      }
      ii = 0;
      for ( ; i < DIM; i++)  {
        rowleft[ii] = am[k][i];
        columnleft[ii] = bv[j][i];
        ii++;
      }
      row = _mm256_loadu_ps( (const float *)&rowleft[0]);              // AVX
      column = _mm256_loadu_ps( (const float *)&columnleft[0]);        // AVX
      temp = _mm256_mul_ps(row, column);                               // AVX
      runner = _mm256_add_ps(runner, temp);                            // AVX
      hold1 = _mm256_extractf128_ps(runner, 0);                        // AVX
      hold2 = _mm256_extractf128_ps(runner, 1);                        // AVX
      hold3 = _mm_add_ps(hold1, hold2);                                // SSE
      hold1 = _mm_hadd_ps(hold3, hold3);                               // SSE
      hold2 = _mm_hadd_ps(hold1, hold1);                               // SSE
      result[k][j] = _mm_cvtss_f32(hold2);
    }
  }

  ticks = omp_get_wtime() - ticks;

  printf("Elapse time: %lf milli-sec\n", ticks*1000.0);

  return(0);
}
```

Figure 3.8: C program using Intel Intrinsics to perform floating matrix multiplication

Each of the matrix multiplication programs was developed using a set of fixed values for the two matrices. The same values were used for the integer and floating point versions with a decimal point added to each value in the floating point case. These matrices were 9x9. The value 9 for the matrix dimension was chosen as it did not fit into any standard Intrinsics vector length. Correct bahaviour over coming this situation needed to be allowed for in the program code. With different vector lengths, the SIZE parameter needed to be changed to 16 for a Mac Pro 2019, 8 for floating point on the Mac Pro 2013, and 4 for integers on the Mac Pro 2013. Note: there is not always a corresponding Intrinsic function for handling integer values and floating point values in a given release of the Intrinsics. In some instances, the corresponding Intrinsics can occur in Intrinsics releases apart.

In developing all parts of each program matrices of positive and negative values were used. By contrast, timing runs of the programs used matrices only of positive values.

Transcribing Intrinsic code written for Mac Pro 2020 to Mac Pro 2013 was not straight forward. From Table 3.7 Intrinsic releases from AVX512_vnni applied to Mac Pro 2019 while Intrinsic release up to AVX applied to Mac Pro 2013. Only after Intrinsic release AVC512f was the vector length 16 for 32-bit values in contrast to AVX where the vector length was 8. Also there was minimal support for integers in AVX, this being added in AVX2, which was not supported by the Mac Pro 2013. Partial loading, that is when loading fewer than the capacity of the register, of integer or floating point data was also not provided. So if 10 data points were to be processed, and Intrinsic function call loaded 8 data values

and then special code was needed to handle the left over, in this example 2 values. Also addition and multiplication of floating point values was available in AVX using a 512-bit register but this was not available for integers. So multiplication of integer matrix data was more difficult using AVX, and thus the Mac Pro 2013.

Handling of values which are fewer than fill a vector often occurs. Processing such values using Intrinsics function has always been available. With Intel Intrinsics release AVX512 there are `_maskz_loadu` calls which take required values into a vector and zero those vector positions which remain unoccupied. A less straight forward approach was needed with Intel Intrinsics release AVX:

- Create a array in memory which with the same number of elements as the Intrinsics register being used;
- Set that array to zero using an Intrinsics function call;
- Process all data elements which fill the vector;
- Copy the data elements remaining into the zeroed array using standard C statements;
- Copy that array into the vector which processed the other data elements;
- Process that vector in the same way as was done in the normal processing case.

In the case of matrix multiplication the matrix needs to be zeroed one time as the remaining values is the same for each row.

Notice in Figure 3.7 the `_mullo` Intrinsics function is used to multiply the integer vectors while in Figure 3.8 the multiplication of the floating point vectors uses the `_mul` Intrinsics function. If the `_mul` Intrinsics function had been used for processing the integers, those integers would have been treated as unsigned integers.

Two types of executables were tested, one using minimal optimization by the compiler and the other using the maximum. The minimal optimized code was compiled by the command:

`gcc –fopenmp –mavx512bw`

and the miximum optimization by the command:

`gcc –O3 –fopenmp –mavx512bw`

These commands appeared to work correctly on both Mac Pro 2019 and 2013. In these commands the `-fopenmp` option of `gcc` was for linking with the `gomp` library to provide the `omp_get_wtime()` function used to obtain the executable time of each test. This is done automatically when OpenMP code processing is requested (via the `-fopenmp` parameter. The `-mavx512bw` option was for generating code for the Intrinsics functions active in the code, and the `-O3` option indicated to use the highest code optimization of the `gcc` compiler. The `-maxv512bw` specified use of Intel Intrinsics release AVX512bw. On the Mac Pro 2013 the executable containing Intel Intrinsics beyond AVX terminated with the message `Illegal instruction` for that hardware could only support AVX and before. Thus, on the Mac Pro 2013 the `-mavx512bw` option was changed to `-mavx`. At the end of each command the name of the source file `matrixf.c` or `matrixi.c` was added. The executable produce in each case was called `a.out`.

The section of the codes in Figures 3.8 and 3.7 which involve Intrinsics functions for execution on Mac Pro 2013 (the third section) produced a compiler warning when the option `-O3` was used. The warning message was `iteration 8 invokes undefined behaviours` and occurred when the matrix `SIZE` parameter was set to 1000 or above. The timing results might be questioned, but appeared consistent with other timing results when this message had not been produced.

3.3.2 Results

Table 3.8 shows the execution times measured for the integer program of Figures 3.7 and Table 3.9 for the floating point program of Figure 3.8. Both Tables cover seven matrix dimensions. In both Tables 3.8 and 3.9 the `Coding` denotes which of the three sections of the respective code was executed, while `OL` denotes the Optimization Level use on the compiler which generating the executable. The program Coding `normal` corresponds to the using standard C statements, *AVX512* corresponds to using Intrinsics functions of Intrinsics release AVX512 and `AVX` to using Intrinsics function of release AVX.. The OL column designates the Optimization Level used when compiling the program with gcc. Minimum and maximum OLs were used. As before, all execution times were measured using the OpenMP library function `omp_get_wtime()`. Each set of values resulted from 25 runs of the test program with the mean of those values calculated and the maximum and mininum of those values determined.

Table 3.8 and Table 3.9 contain data for both Mac Pro 2019 and 2013. This data was obtained by executing the portions of the test programs using Intrinsic functions available on the Mac Pro 2013 on the Mac Pro 2019. These data give an indication of the effect of available Intrinsic functions had on the performance of the two Mac Pros. All Intrinsic functions which executed on the Mac Pro 2013 also executed on the Mac Pro 2019, but not vice versa.

Table 3.8: Execution times for multiplication of two square matrices of integer values

Controlled variables				Dimension of matrix						
Mac	Coding	OL	Mode	100	500	1000	2000	3000	4000	5000
2019	normal	Nil	min	8.2	244.3	1846.1	14720.6	49798.8	118560.2	232164.6
			mean	8.4	247.2	1852.8	14762.5	49940.6	118798.4	232920.1
			max	9.1	250.6	1868.8	14822.7	50141.7	119012.8	234566.4
	AVX512	Nil	min	2.2	73.3	450.3	3659.7	11549.4	29124.0	54347.3
			mean	2.4	76.5	455.6	3683.7	11603.6	29314.0	54692.0
			max	2.8	80.0	472.1	3778.4	11761.6	29992.5	55415.2
	AVX	Nil	min	5.6	168.8	1191.7	9349.1	31466.5	74547.8	145691.2
			mean	5.5	174.1	1198.5	9355.0	31472.3	74572.9	145740.5
			max	5.6	199.9	1222.6	9377.8	31479.2	74746.9	146104.2
2019	normal	-O3	min	0.3	20.2	173.6	1198.8	4162.7	12163.3	28517.7
			mean	0.3	22.6	174.9	1213.1	4194.2	12636.9	29233.5
			max	0.4	25.2	178.7	1226.5	4223.4	13200.1	29762.3
	AVX512	-O3	min	0.3	18.2	113.2	875.3	2769.8	7130.0	13365.1
			mean	0.3	20.1	119.7	901.9	2810.7	7245.8	13886.2
			max	0.4	23.2	133.0	947.4	2984.6	8935.5	15167.7
	AVX	-O3	min	0.8	36.7	163.4	1126.6	3738.0	8950.3	17489.4
			mean	0.8	40.2	169.8	1165.3	3823.9	9134.2	17910.6
			max	0.9	43.1	194.3	1264.8	4169.0	10069.3	19790.2
2013	normal	Nil	min	9.2	718.5	7212.1	58750.9	199549.5	472904.4	925767.9
			mean	9.3	830.2	7327.5	59115.9	199668.9	473147.1	926359.0
			max	9.3	878.1	7403.8	59229.8	199875.3	473549.4	927100.3
	AVX	Nil	min	5.2	415.0	4019.7	34280.3	117275.4	278180.9	544876.9
			mean	5.2	492.6	4259.5	34758.4	117436.5	278460.8	545409.3
			max	5.2	540.4	4373.6	34855.9	117534.0	278616.5	545787.0
2013	normal	-O3	min	0.7	37.0	458.2	4865.7	16766.3	39781.1	83436.4
			mean	0.7	52.6	488.7	4931.9	16936.0	40149.4	83915.7
			max	0.7	63.7	522.7	4973.4	16955.3	40419.0	84737.5
	AVX	-O3	min	0.7	33.5	408.2	3961.1	13487.9	32498.5	68265.9
			mean	0.7	47.9	463.7	4102.0	13643.7	32640.5	68619.9
			max	0.7	60.6	506.0	4153.9	13714.8	32687.3	69097.0

Figures 3.9 and 3.10 are log/log plots of the integer matrix multiplication execution data of Table 3.8 for the Mac Pro 2019 and Mac Pro 2013 respectively.

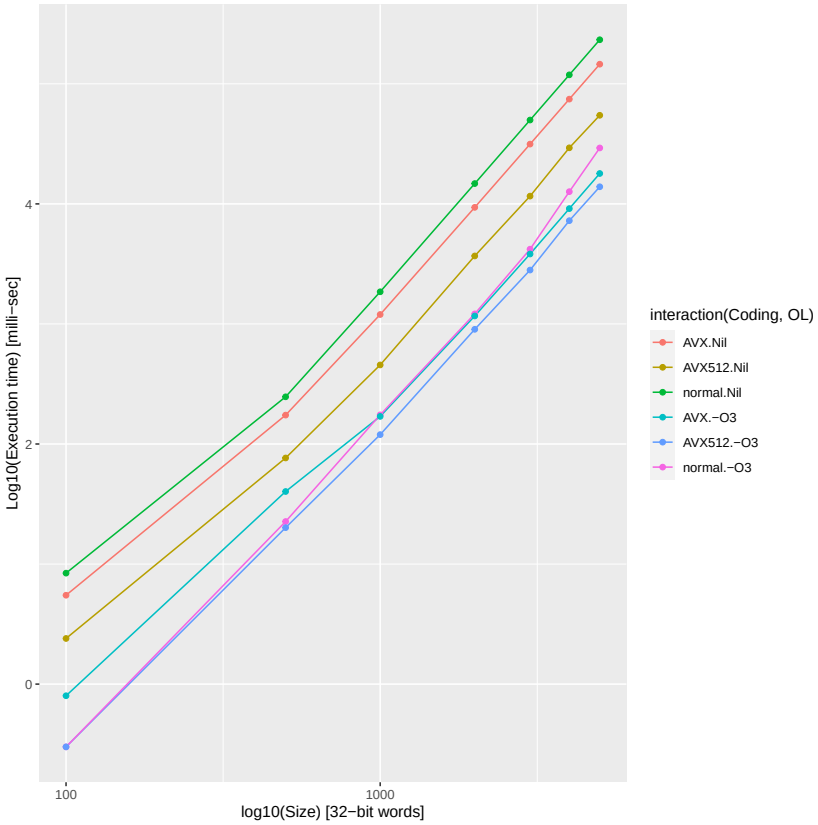

Figure 3.9: MacPro 2019 Intrinsics execution times for multiplication of 32-bit integer square matrices

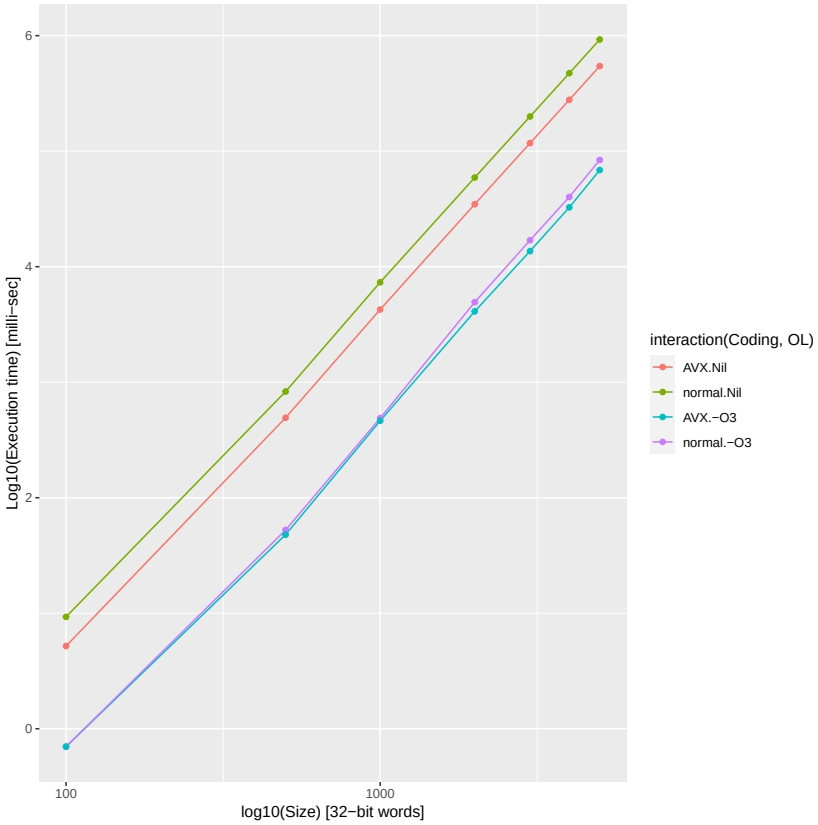

Figure 3.10: MacPro 2013 Intrinsics execution times for multiplication of 32-bit integer square matrices

The log/log plot is used to give equal emphasis to the small and large matrix dimension execution times.

The plots and the data relate to the performance of a single core on both the Mac Pro 2019 and 2013. This is vector performance – SIMP performance – Single Instruction Multiple data. A single instruction, or Intrinsic function, operates on multiple pieces of data, such as integer or floating point values, stored in the Intrinsic regester. In most cases the Intrinsic function is implemented by a single machine instruction in the CPU chip, but in some cases several machine instructions are used for the implementation. So in most instances using Intrinsics functions is SIMD processing where as in other cases it is pseudo-SIMD processing.

Figures 3.9 and 3.10 show using Intel Intrinsics reduces the execution times on both MacPros when using 32-bit integers. The near parallel alignment of each line in these plots indicates the approximate similar execution time verse matrix dimension. The `normal` in each plot is without Intel Intrinsics used and provides a relative measure.

Table 3.9: Execution times for multiplication of two square matrices of floating point values

Controlled variables				Dimension of matrix						
Mac	Coding	OL	Mode	100	500	1000	2000	3000	4000	5000
2019	normal	Nil	min	9.0	270.4	2062.2	16485.7	55663.5	132368.4	258262.4
			mean	9.4	273.5	2063.2	16492.6	55683.0	132568.6	258470.4
			max	9.5	292.5	2065.1	16535.1	55706.7	133685.3	261137.5
	AVX512	Nil	min	2.5	68.2	432.6	3163.6	11648.5	28875.7	60763.1
			mean	2.6	70.7	433.9	3181.9	11747.3	30621.6	63853.2
			max	2.7	83.1	440.3	3205.7	11962.8	31034.9	65387.6
	AVX	Nil	min	4.0	92.9	608.2	4757.2	16121.5	40665.3	79172.3
			mean	4.1	94.9	609.1	4761.7	16174.4	41352.0	81313.0
			max	4.2	97.6	610.0	4780.4	16249.7	41715.6	82561.6
2019	normal	-O3	min	2.3	130.5	1016.7	8322.1	28375.5	68423.6	134822.7
			mean	2.4	134.0	1018.6	8326.7	28404.0	68709.9	135351.7
			max	2.5	157.8	1039.6	8343.9	28421.9	68948.9	135699.2
	AVX512	-O3	min	0.2	19.0	174.2	1197.6	4123.9	11709.6	27269.8
			mean	0.2	21.3	175.2	1212.1	4170.0	12417.9	28598.3
			max	0.4	24.7	176.7	1222.7	4193.1	12952.8	28846.7
	AVX	-O3	min	0.6	29.7	174.7	1229.1	4651.8	14939.5	31800.3
			mean	0.6	32.3	176.0	1233.6	4800.3	15481.3	32783.1
			max	0.8	35.3	178.0	1253.4	4958.7	15793.7	33430.4
2013	normal	Nil	min	5.4	775.0	7418.5	60195.6	204343.6	485979.1	949151.4
			mean	9.1	891.8	7531.4	60581.4	204749.0	486108.8	949284.0
			max	9.4	945.8	7575.6	60790.2	204994.3	486400.1	949782.0
	AVX	Nil	min	4.9	371.5	3520.0	29904.6	100519.3	238789.3	467223.7
			mean	5.0	434.0	3715.2	29995.1	100677.6	239104.9	467404.2
			max	5.0	470.3	3773.2	30035.9	100720.8	239227.8	467581.9
2013	normal	-O3	min	1.7	209.5	2265.0	19664.0	67296.2	159490.8	312744.9
			mean	1.7	258.7	2403.3	19881.9	67380.4	159935.7	312815.6
			max	1.8	287.0	2454.3	19932.3	67413.2	160024.2	312831.8
	AVX	-O3	min	0.3	34.3	380.0	4332.4	15828.3	38477.0	81411.4
			mean	0.3	45.5	441.8	4596.2	16288.8	38556.3	81953.6
			max	0.4	57.8	482.9	4697.5	16599.1	38585.1	83071.7

Table 3.9 contains the equivalent of the Table 3.8 execution times when floating point Intrinsics were used in the implementation program. Figures 3.11 and 3.12 give log/log plots of that data for Mac Pro 2019 and Mac Pro 2013, respectively. The similarity and contrasts between the data of Tables 3.8 and 3.9 are brought into focus by comparing Figure 3.9 with Figure 3.10, and Figure 3.11 with Figure 3.12. The log/log plots of Figures 3.13 and 3.14 show the relative performance of Mac Pros 2019 and 2013 performing 32-bit integer and floating point matrix multiplication, respectively.

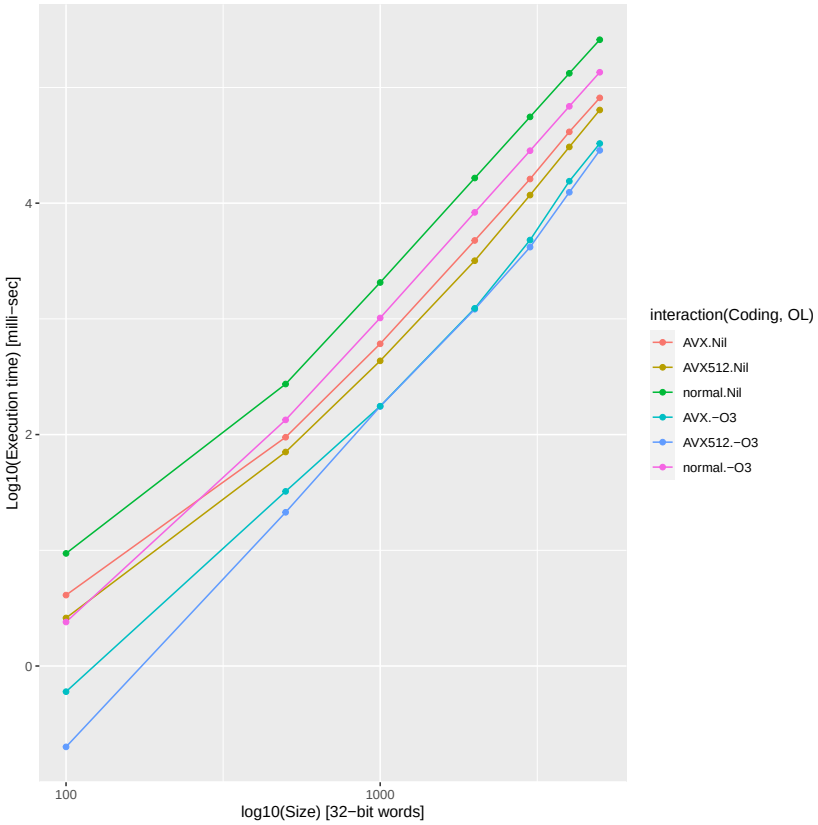

Figure 3.11: MacPro 2019 Intrinsics execution times for multiplication of 32-bit floating square matrices

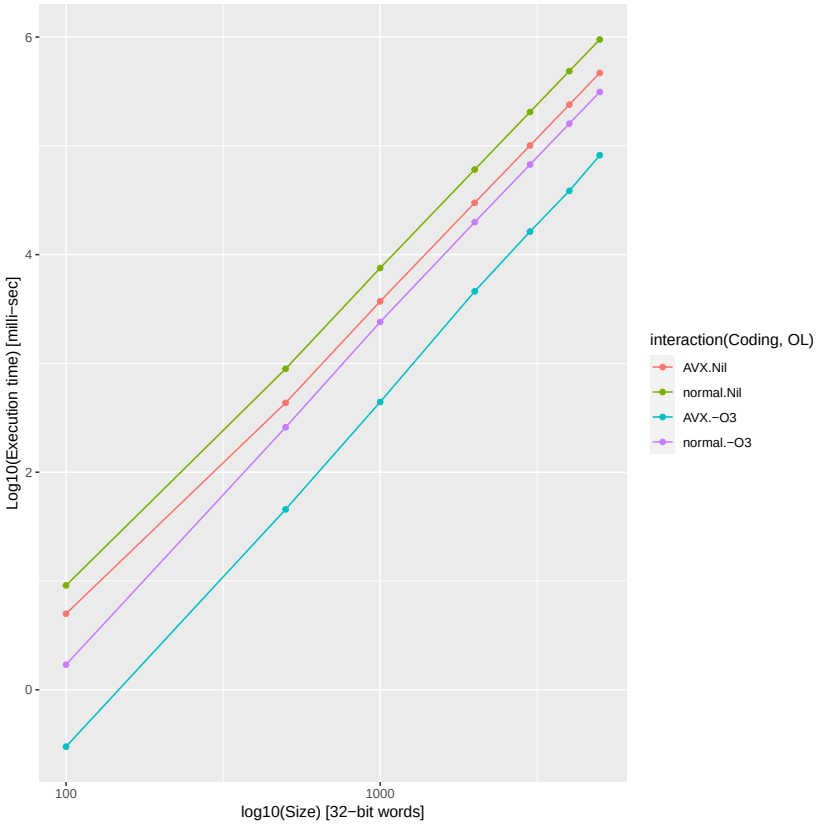

Figure 3.12: MacPro 2013 Intrinsics execution times for multiplication of 32-bit floating square matrices

A consequence of the release scheduled followed for floating point and integer Intel Intrinsics can be seen in this data. Figure 3.9 (or Table 3.8) for integer processing shows a smaller reduction in executions time between AVX-512 and AVX Intel Intrinsics use than in the corresponding Figure 3.11 (or Table 3.9) for floating point use. In the code of Figure 3.8 which produced the data of Figures 3.11 (and 3.12), the `SIZE` constant was 8 instead of 4 in the code of Figure 3.7. So 8 floating point values were used in AVX-512 Intel Intrinsics there as opposed to 4 in the AVX-512 Intel Intrinsics data of Figure 3.9. The Intel Intrinsics support for floating point generally preceded integer support, in Intrinsics releases. However, the data in Table 3.8 and Table 3.9 show 32-bit integer matrix computation was slightly faster than corresponding floating point computation on both the MacPro 2019 and MacPro 2013.

Figures 3.13 and 3.14 compare execution times of Mac Pro 2019 and Mac Pro 2013 for extremes in integer and floating point matrix multiplication performance. From Tables 3.8 and Table 3.9 those extremes are when using normal (or standard) C code for performing matrix multiplications and when using AVX-512 Intrinsics on the Mac Pro 2019 and AVX Intrinsics on the Mac Pro 2013. Those Intrinsics extremes are magnified by using the results when the Intrinsics code was compiled with an Optimization Level (OL) of -O3. In both the integer and floating points case the behaviour was the same. The normal coding executing on the Mac Pro 2013 had the longest execution time while the optimized AVX-512 Intrinsics code executing on the Mac Pro 2019 had the shortest execution time. This behavior remained the same across all matrix dimensions tested. In both the integer and floating point cases, matrix dimensions 500 and above followed similar execution time verses matrix dimension behaviour for both integer and floating point on both Mac Pros.

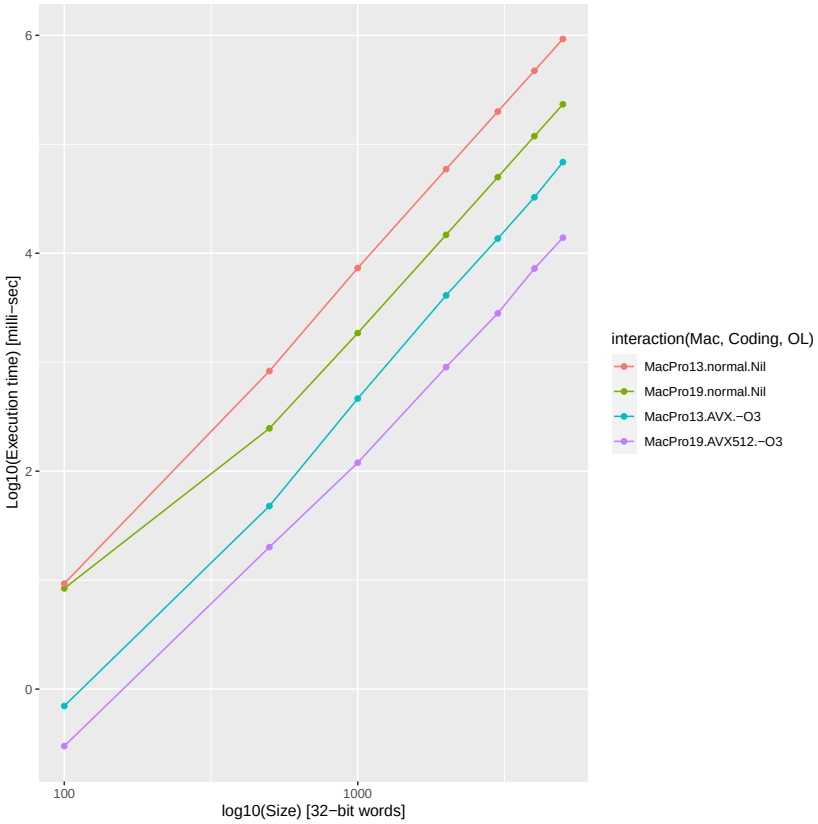

Figure 3.13: MacPro 2019 and 2013 Intrinsics executions times for 32-bit integer square matrix multiply

From Figures 3.9, 3.10, 3.11, and 3.12 using the highest optimization level when compiling standard C code resulted in substantial reduction in execution time than using an Intel Intrinsics implementation of the code. In Figure 3.10 this was reversed in the floating point multiplication on the MacPro 2013. However, the reduction in execution time obtained was less than those in Figures 3.9, 3.10, and3.11. These result suggest the `gcc` compiler can generate optimized code which can run faster than hand written Intel Intrinsics code.

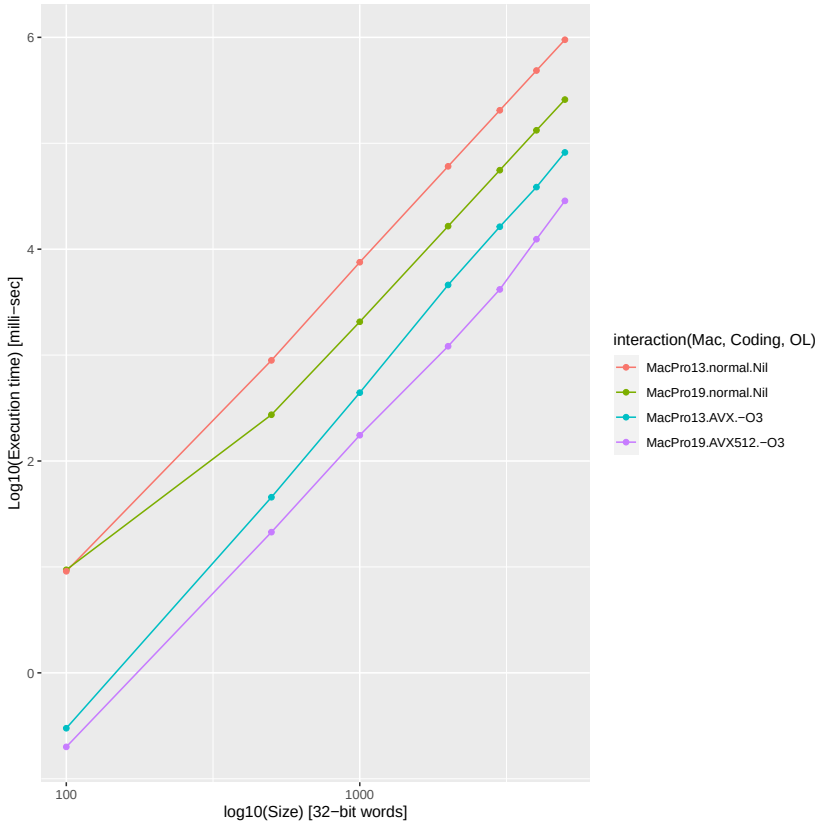

Figure 3.14: MacPro 2019 and 2013 Intrinsics executions times for 32-bit floating square matrix multiply

3.3.3 Searching for a pattern

For a program to implement in Intel Intrinsics which involved moving and manipulating data as opposed to being arithmetic intensive, pattern matching was used. This program used a *linear search* of the data which have no prescribed order; the order was random. The data used was composed of integer values in the range 0 to 9, although the program should perform correctly with negative values whose absolute values were in this range. The search was to locate all occurrences of a fitted sequence of 6 integer values, where both those values and their order in the sequence was matched. A count of the number of matches found was recorded. The question: Can Intrinsics be used to advantage to implement this program?

This program was inspired by a problem in biology. A DNA sequence generally is too long for directly determining the *base pair* occurrence alomg it's length. It is thus cut into smaller lengths which are of a sufficient length upon which such sequencing can be performed. After sequencing, the sequenced base pairs are reassembled to produced the sequence of the original DNA. This cutting is done chemically by reagents which cut DNA at know 4 or 6 sequences. For this program the DNA was simulated by a sequence of small, positive, integers. The sequence where a cut would be placed was simulated by given sequence of 6 positive integers. For a true DNA sequence, the number of fragments which would result from applying a reagent with know sequence cutting would be unknown as the original base pair sequence was unknown. This unknown fragment number, which is direct consequence to the number of sequence matches obtained is also emulated in this program simulation.

For this program the data was a vector of integer values where as with DNA it would be a character sequence. Characters can also be interpreted as integers. The sequence to be found was also in a vector of the same type as the data. The data was compared against this vector. In the normal approach to solving this problem, the sequence being found was compared one data integer after another in sequential order against the data vector. The program would consist of reading a portion of the data vector from memory, performing a bit-for-bit comparison of that portion with the bits of the reference

```c
/*  Code to search for a sequence of 6 short integers in a random array of
 *  short integers.
 *
 *  Coded by: Ross Maloney
 *  Date:     May 2020
 */

#include <omp.h>
#include <stdio.h>
#include <immintrin.h>
#include <time.h>

#define DIM 1000000000

#define SIZE 32    // for Mac Pro 2019
#define SIZE 8     // for Mac Pro 2013

short chain[DIM];

short target[] = {3, 2, 1, 7, 3, 1,};

int main(int argc, char **argv)
{
  double ticks;
  long   i;
  int    j, match, count, n, bitcount;
  __m512i  one, two, three, four, five, six;              // for Mac Pro 2019
  __m512i  v1, v2, v3, v4, v5, v6;                        // for Mac Pro 2019
  __mmask32  mask1, mask2, mask3, mask4, mask5, mask6;    // for Mac Pro 2019
  __m128i  one, two, three, four, five, six;              // for Mac Pro 2013
  __m128i  v1, v2, v3, v4, v5, v6;                        // for Mac Pro 2013
  int  mask1, mask2, mask3, mask4, mask5, mask6;          // for Mac Pro 2019

  srand(time(NULL));
  for (i = 0; i < DIM; i++)  {
    chain[i] = rand() % 10;
  }

  ticks = omp_get_wtime();

  /* Normal processing */

  count = 0;
  for (i = 0; i < DIM-4; i++)  {
    match = 1;
    for (j = 0; j < 6; j++)  {
      if ( target[j] != chain[i+j] )  {
        match = 0;
        break;
      }
    }
    if ( match == 1 )  {
      count++;
      i = i + 5;
    }
  }
  printf("count = %d\n", count);

  /* Mac Pro 2019 Intrinsics */

  count = 0;
  one = _mm512_set1_epi16(target[0]);                     // AVX-512
  two = _mm512_set1_epi16(target[1]);                     // AVX-512
  three = _mm512_set1_epi16(target[2]);                   // AVX-512
  four = _mm512_set1_epi16(target[3]);                    // AVX-512
  five = _mm512_set1_epi16(target[4]);                    // AVX-512
  six = _mm512_set1_epi16(target[5]);                     // AVX-512
  for (i = 0; i < DIM; i+=SIZE)   {
    v1 = _mm512_loadu_si512((const __m512i *)&chain[i]);  // AVX-512
    v2 = _mm512_loadu_si512((const __m512i *)&chain[i+1]);// AVX-512
```

Figure 3.15: C program using Intel Intrinsics to perform linear search for a sequence (continues ...)

```
    v3 = _mm512_loadu_si512((const __m512i *)&chain[i+2]);                    // AVX-512
    v4 = _mm512_loadu_si512((const __m512i *)&chain[i+3]);                    // AVX-512
    v5 = _mm512_loadu_si512((const __m512i *)&chain[i+4]);                    // AVX-512
    v6 = _mm512_loadu_si512((const __m512i *)&chain[i+5]);                    // AVX-512
    mask1 = _mm512_cmpeq_epi16_mask(one, v1);                                 // AVX-512
    mask2 = _mm512_cmpeq_epi16_mask(two, v2);                                 // AVX-512
    mask3 = _mm512_cmpeq_epi16_mask(three, v3);                               // AVX-512
    mask4 = _mm512_cmpeq_epi16_mask(four, v4);                                // AVX-512
    mask5 = _mm512_cmpeq_epi16_mask(five, v5);                                // AVX-512
    mask6 = _mm512_cmpeq_epi16_mask(six, v6);                                 // AVX-512
    if ( mask1 & mask2 & mask3 & mask4 & mask5 & mask6)  {
      bitcount = 0;
      n = mask1 & mask2 & mask3 & mask4 & mask5 & mask6;
      while ( n )  {
        n = n & (n-1);
        bitcount++;
      }
      count += bitcount;
    }
  }


  /* Mac Pro 2013 Intrinsics */

  count = 0;
  one = _mm_set1_epi16(target[0]);                                            // SSE2
  two = _mm_set1_epi16(target[1]);                                            // SSE2
  three = _mm_set1_epi16(target[2]);                                          // SSE2
  four = _mm_set1_epi16(target[3]);                                           // SSE2
  five = _mm_set1_epi16(target[4]);                                           // SSE2
  six = _mm_set1_epi16(target[5]);                                            // SSE2
  for (i = 0; i < DIM; i+=SIZE)  {
    v1 = _mm_loadu_si128((const __m128i *)&chain[i]);                         // SSE2
    v2 = _mm_loadu_si128((const __m128i *)&chain[i+1]);                       // SSE2
    v3 = _mm_loadu_si128((const __m128i *)&chain[i+2]);                       // SSE2
    v4 = _mm_loadu_si128((const __m128i *)&chain[i+3]);                       // SSE2
    v5 = _mm_loadu_si128((const __m128i *)&chain[i+4]);                       // SSE2
    v6 = _mm_loadu_si128((const __m128i *)&chain[i+5]);                       // SSE2
    v1 = _mm_cmpeq_epi16(one, v1);                                            // SSE2
    v2 = _mm_cmpeq_epi16(two, v2);                                            // SSE2
    v3 = _mm_cmpeq_epi16(three, v3);                                          // SSE2
    v4 = _mm_cmpeq_epi16(four, v4);                                           // SSE2
    v5 = _mm_cmpeq_epi16(five, v5);                                           // SSE2
    v6 = _mm_cmpeq_epi16(six, v6);                                            // SSE2
    mask1 = _mm_movemask_epi8(v1);                                            // SSE2
    mask2 = _mm_movemask_epi8(v2);                                            // SSE2
    mask3 = _mm_movemask_epi8(v3);                                            // SSE2
    mask4 = _mm_movemask_epi8(v4);                                            // SSE2
    mask5 = _mm_movemask_epi8(v5);                                            // SSE2
    mask6 = _mm_movemask_epi8(v6);                                            // SSE2
    if ( mask1 & mask2 & mask3 & mask4 & mask5 & mask6)  {
      bitcount = 0;
      n = mask1 & mask2 & mask3 & mask4 & mask5 & mask6;
      while ( n )  {
        n = n & (n-1);
        bitcount++;
      }
      count += bitcount;
    }
  }
  count = count/2;

  ticks = omp_get_wtime() - ticks;

  printf("DIM = %ld\n", DIM);
  printf("count = %d\n", count);
  printf("Elapse time = %lf milli-sec\n", ticks*1000.0);
}
```

Figure 3.15: C program using Intel Intrinsics to perform linear search for a sequence

vector. If all bits in the sequence matched, then the reference sequence had been found in the data stream. A count of such matches was incremented. The data vector would then be read for another comparison. If one mismatch occurred, the comparison would be repeated after advancing the reading of the data vector by one integer. This process terminates when the whole of the data vector had passed through this processing. If the data vector is long, this processing can take considerable time.

An Intrinsics implementation used a pair consisting of a comparison vector and a data chunk vector. There would be one pair for each integer in the target sequence for which the search is to be performed. Since this target contained 6 integers, 6 pairs were used. Each pair was logicly linked to one of the values in the target search sequence and to be ordered in the same manner as such values of the target sequence. Each element of the comparison vector of each pair is filled with one member of target search sequence to which it's pair had been linked. In integer to act as a masked was assigment to each pair.

The algorithm consisted of loading successive fixed size chunks of the data vector being search. The size of each chunk corresponded to the vector length of the Intel Intrinsic being used. These fixed sized chunks were loaded into the data chunk vector of each pair. Each data chunks was compared with the corresponding comparison vector of the pair, resulting in a mask being formed. The masks resulting from each of the pairs were bitwise added together. If this bitwise add was non-zero, one or more of the search sequence existed in the data chucked just processed, and were added to the count of those found. The algorithm termimated when all the data had been passed through this comparison process.

How could Intel Intrinsics and data type be selected to enable processing of the problem. If the problem analogue with DNA was to be maintained, the integers would have been bytes. But Intel Intrinsics handling of bytes is limited as shown in Table 3.10, particularly when comparison is required. The next step up is a 16-bit integer. In that storage a positive integer in the range of 0 to 9 would be stored. Although only positive 16-bit integers were used, the Intel Intrinsics employed handled signed 16-bit integers. Figure 3.15 lists the program which was written to implement the search algorithm and generate the results tabulated in Table 3.11.

Table 3.10: Finding Intel Intrinsics for linear pattern search on Mac Pro 2020 and 2013

		AVX-512						AVX		SSE2	
Operation	Data Size	Vector Length	No.	Vector Length	No.	Vector Length	No.	Vector Length	No.	Vector Length	No.
loadu	8-bits	512	64	256	32	128	16				
loadu	16-bits	512	32	256	16	128	8				
loadu	32-bits	512	16	256	8	128	4	256	8		
compare eq	8-bits	512	64	256	32					128	16
compare eq	16-bits	512	32	256	16					128	8
compare eq	32-bits	512	16	256	8					128	4
const vect	8-bits	512	64					256	32	128	16
const vect	16-bits	512	32					256	16	128	8
const vect	32-bits	512	16					256	8	128	4
onto mask	8-bits	512	64	256	32					128	16
onto mask	16-bits	512	32	256	16	128	8				
onto mask	32-bits	512	16	256	7	128	4				
horz add	8-bits										
horz add	16-bits									128	8
horz add	32-bits					256	8				
reduce add	8-bits										
reduce add	16-bits										
reduce add	32-bits	512	16								
add/sub	8-bits	512	64	256	32					128	32
add/sub	16-bits	512	32	256	16					128	8
add/sub	32-bits	512	16	256	8					128	4
zero check	8-bits	512	64	256	32					128	16
zero check	16-bits	512	32	256	16					128	8
zero check	32-bits	512	16	256	8					128	4

Choices were made in writing the program of Figure 3.15. A 32-bit integer could have been used. This would have enable larger numbers to be used in the search algorithm. It also would have reduced the Intrinsics vector length over which each search step was performed. This would also have increased the number of times the Intrinsics vector registers would have needed refilling from memory. Table 3.10 tabulates the vector length and number of integers in that vector for the Intel Intrinsics which could have been selected for the program. The Intrinsics tabulated are covers, meaning `AVX-512` Intrinsics includes, the Intrinsics of `AVX`, `AVX2`, and those releases before it. Similarly for `AVX` and `SSE2`. Intrinsics release `AVX-512` was implemented on Mac Pro 2019, while release `AVX` was implemented on Mac Pro 2013. Unfortunately Intel Intrinsics `AVX2` was not available on the Mac Pro 2013. The `AVX2` Intel Intrinsics which were pertinent to these considerations are tabulated in Table 3.10 under `AVX-512`. Table 3.10 indicates the limited choice of Intrindics constructs which were available on the Mac Pro 2013. A similar programming approach on the Mac Pro 2019 and 2013 was required. Writing the program of Figure 3.15 was a compromise between utility, providing a solution to the original problem, and speed of operation of the program on both Mac Pros.

Table 3.11: Execution times for linear search for a sequence

Controlled variables				Length of search vector						
Mac	coding	OL	Mode	1×10^5	1×10^6	1×10^7	1×10^8	1×10^9	8×10^9	16×10^9
2019	normal	Nil	min	1.2	5.3	29.6	298.2	2988.1	27410.0	54915.4
			mean	1.2	6.4	29.8	299.1	2994.6	27507.0	55155.0
			max	1.3	7.3	30.0	300.2	3005.4	27744.0	55495.7
			non 0	0	2	11	100	1000	8050	16050
	AVX512	Nil	min	0.2	0.9	4.5	47.4	474.8	3833.7	7582.2
			mean	0.2	1.0	4.5	47.5	478.8	3834.6	7662.6
			max	0.2	1.1	4.7	47.8	479.5	3836.3	7672.7
			non 0	0	2	10	105	1000	8020	16000
	AVX	Nil	min	0.8	3.7	20.3	204.9	2050.3	16393.1	32753.9
			mean	0.8	4.5	20.3	205.1	2057.0	16426.4	32838.9
			max	0.9	5.2	20.5	205.4	2061.1	16457.5	32931.7
			non 0	0	2	11	100	1010	8050	16050
2019	normal	-O3	min	0.7	3.7	16.8	168.1	1682.1	11902.7	23806.4
			mean	0.7	4.2	16.8	168.2	1682.5	11905.6	23812.7
			max	0.8	4.4	16.9	168.5	1683.2	11909.5	23829.6
			non 0	0	1	10	100	1000	8050	16050
	AVX512	-O3	min	0.1	0.2	1.1	15.7	164.5	1322.3	2655.4
			mean	0.1	0.3	1.2	16.3	171.8	1391.2	2765.6
			max	0.1	0.3	1.3	16.9	176.4	1416.9	2832.9
			non 0	0	2	10	110	1000	8050	16000
	AVX	-O3	min	0.1	0.5	2.2	25.3	255.2	2042.7	4098.3
			mean	0.1	0.6	2.2	26.1	265.1	2124.0	4283.1
			max	0.1	0.9	2.5	26.7	273.0	2188.4	4375.3
			non 0	0	1	10	100	1010	8050	16100
‘ 2013	normal	Nil	min	1.2	4.1	90.3	1189.7	11897.1	100877.1	
			mean	1.2	7.2	112.2	1191.2	11906.8	100890.4	
			max	1.4	10.5	119.3	1195.2	11921.1	100920.3	
			non 0	0	1	10	100	1000	7980	
	AVX	Nil	min	0.7	2.3	55.9	673.0	6727.7	53364.9	
			mean	0.7	4.7	64.8	673.3	6732.1	53445.4	
			max	0.7	6.1	67.4	673.7	6736.9	53484.5	
			non 0	0	1	11	100	1010	7975	
2013	normal	-O3	min	0.5	1.8	39.4	631.8	6837.2	41699.8	
			mean	0.5	3.0	64.5	679.6	6838.2	41702.7	
			max	0.5	4.5	68.5	684.2	6839.2	41705.2	
			non 0	0	1	13	100	1000	8010	
	AVX	-O3	min	0.1	0.4	9.6	119.8	1197.9	9577.1	
			mean	0.1	0.7	11.4	119.8	1198.2	9583.4	
			max	0.1	1.1	12.0	119.9	1198.3	9585.8	
			non 0	0	1	10	100	1010	7980	

Table 3.11 shows the results produced by the program of Figure 3.15. Although the same sequence of 6 integer values to be found was used in each run of the program, the sequence being search was different for each run it having been generated as a sequence of random numbers, modulo 10. The generations of that sequence was not part of the execution times shown in Table 3.11. Only one core of the Mac Pro involved in obtaining a set of execution times was used. With each minimum/mean/maximum set of values, the `non 0` number is an indication of how many sequence matches were obtained when that set was obtained. It is only indicative, not rigorous.

Notice the extent of the sequence being searched. At 10^5 integers being searched, zero matches were found. This, together with the trend of matched found, suggested below this length a similar number of matches would be obtained for the set sequence of 6 integers in a random generated sequence. This formed the lower limit of the search. The longest sequence searched had a length of 16×10^9. This corresponded to the 32GB memory size on the Mac Pro 2019 used. The Mac Pro 2013 had a 16GB memory size, and the largest sequence searched on it was 8×10^8. This length was also reached on the Mac Pro 2019. The Mac Pro could not search a 16×10^9 sequence length.

Figures 3.16 and 3.17 are log/log plots of the execution times of the data in Table 3.11 for Mac Pro 2019 and 2013, respectively. In each case the `Size` is the length of the vector searched.

Figure 3.18 is a log/log plot of the maximum and minimum execution times on the Mac Pros 2019 and 2013. The maximum execution time for each Mac Pro occurred when executing the normal (or standard) C code implementation of the algorithm. The minimum execution time occurred when using using the `-O3` optimization level (OL) when compiling an Intrinsics implementation. In the case of the Mac Pro 2019 this was the AVX-512 Intrinsics version, and for the Mac Pro 2013 the AVX Intrinsics version. As in Figures 3.16 and 3.17, the maximum/minimum behaviour in Figure 3.18 remains constant for the length of the sequence being searched.

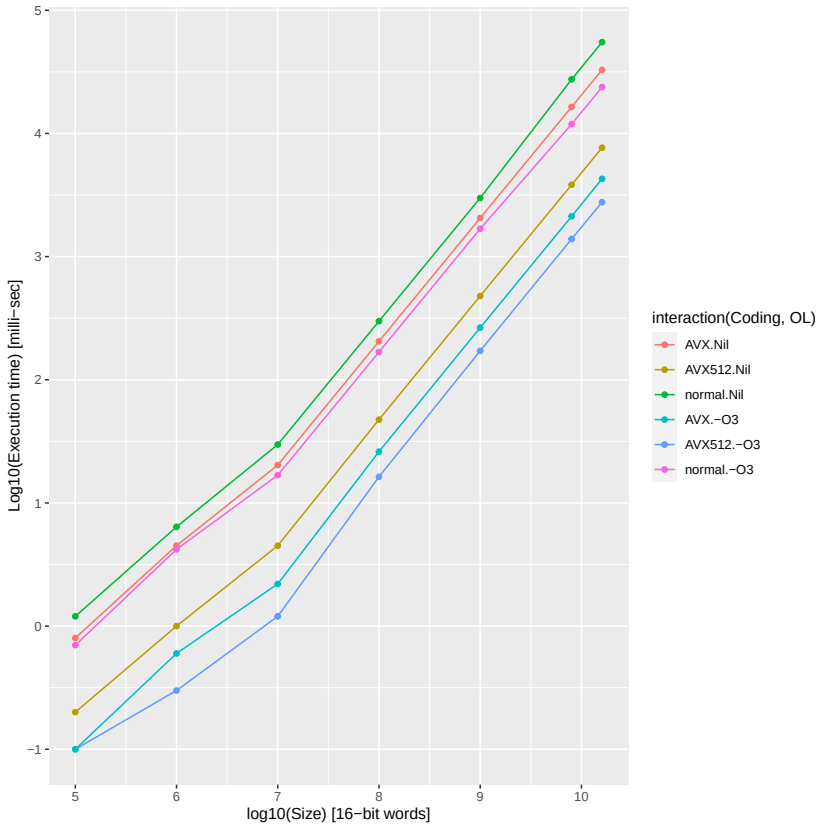

Figure 3.16: MacPro 2019 execution times for Intrinsics search for a 6 16-bit integer sequence

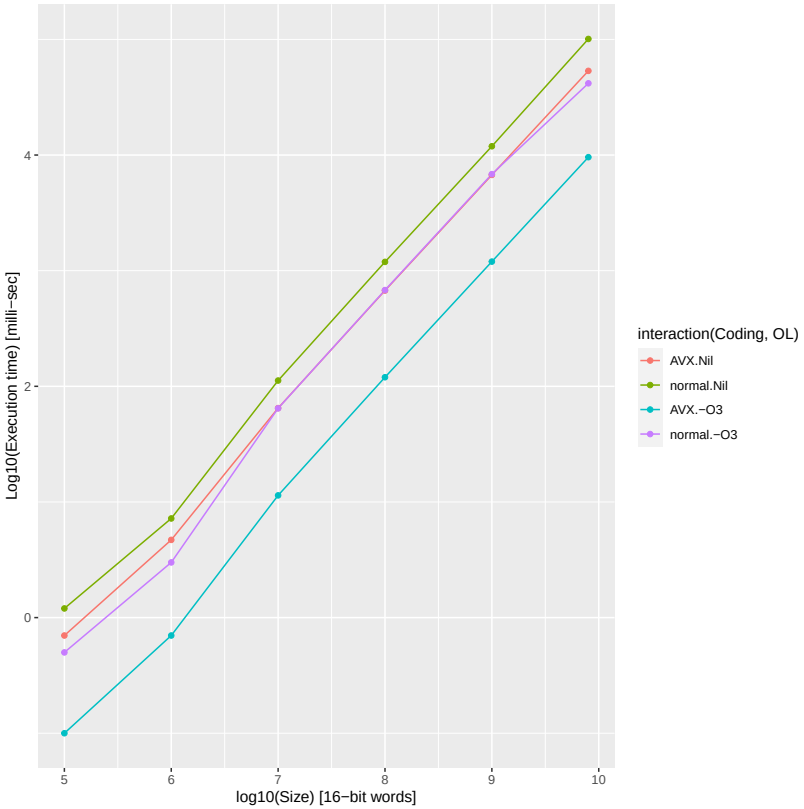

Figure 3.17: MacPro 2013 execution times for Intrinsics search for a 6 16-bit integer sequence

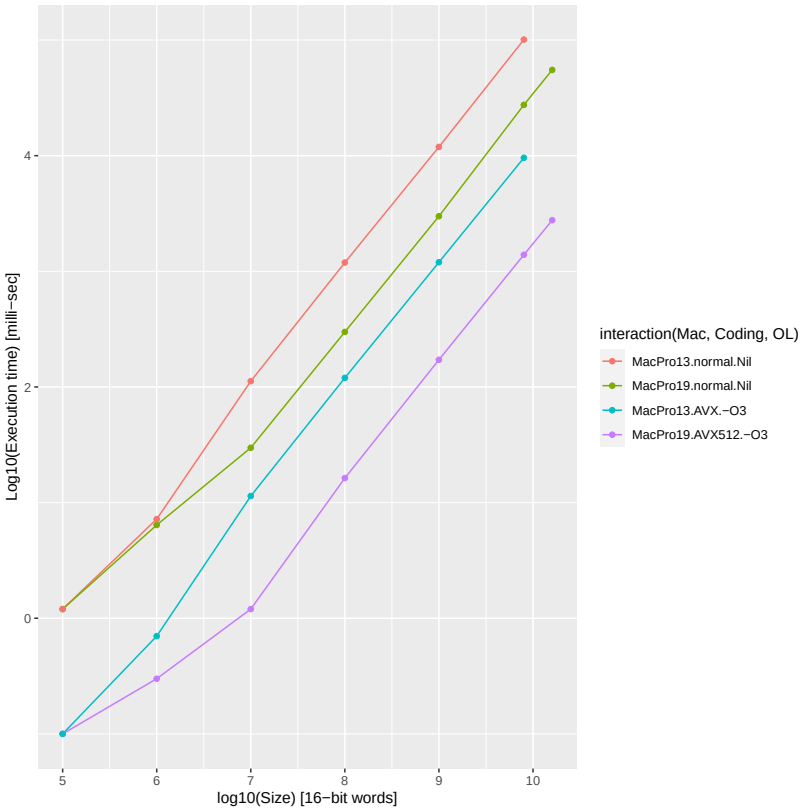

Figure 3.18: Superimposed MacPro 2019 and 2013 Intrinsics execution times for the integer sequence search

3.4 Double precision

There are a large number of numerical problems which need more precision in performing their calculations than 32 bits can provide. Double precision is the next step of greater precision which is implemented in hardware. Intel Intrinsic functions and Nvidia CUDA provide double precision capability. This Section shows the relative behaviour of Intrinsic functions and CUDA of both 32 bit and 64 bit format in relation to corresponding single variable implementation of such arithmetic. Both integer and floating point arithmetic is considered.

3.4.1 By using Intrinsic functions

Table 3.12: A selection of Intel double precision Intrinsics

Operation performed	64-bit floating point		64-bit integer	
	Intrinsic	Release	Intrinsic	Release
load array	_mm_loadu_pd	SSE2	_mm_loadu_epi64	AVX-512F
	_mm256_loadu_pd	AVX	_mm256_loadu_epi64	AVX-512F
	_mm512_loadu_pd	AVX-512F	_mm512_loadu_epi64	AVX-512F
load values	_mm_setr_pd	SSE2	_mm_setr_epi64	SSE2
	_mm256_setr_pd	AVX	_mm256_setr_epi64x	AVX
	_mm512_setr_pd	AVX-512F	_mm512_setr_epi64	AVX-512F
load selective	_mm_maskz_loadu_pd	AVX-512F	_mm_maskz_loadu_epi64	AXV-512F
	_mm256_extractf128_pd	AVX-512F	_mm256_maskz_loadu_epi64	AVX-512F
	_mm512_maskz_loadu_pd	AVX-512	_mm512_maskz_loadu_epi62	AVX-512F
add	_mm_add_pd	SSE2	_mm_add_epi64	SSE2
	_mm256_add_pd	AVX	_mm256_add_epi64	AVX2
	_mm512_add_pd	AVX-512F	_mm512_add_epi64	AVX-512F
multiply	_mm_mul_pd	SSE2	_mm_mullo_epi64	AVX-512VL
	_mm256_mul_pd	AVX	_mm256_mullo_epi64	AVX-512VL
	_mm512_mul_pd	AVX-512F	_mm512_mullo_epi64	AVX-512DQ
multiply & add	_mm_fmadd_pd	FMA	*	
	_mm256_fmadd_pd	FMA	*	
	_mm512_fmadd_pd	AVX-512F	*	
horizontal add	_mm_hadd_pd	SSE3	*	
	_mm256_hadd_pd	AVX	*	
	*		*	
sum all	*		*	
	*		*	
	_mm512_reduce_add_pd	AVX-512F	_mm512_add_reduce_epi64	AVX-512F
zeroize	_mm_setzero_pd	SSE2	_mm_setzero_si128	SSE2
	_mm256_setzero_pd	AVX	_mm256_setzero_si256	AVX
	_mm512_setzero_pd	AVX-512F	_mm512_setzero_si512	AVX-512F
initial pattern	_mm_set_pd	SSE2	_mm_set_epi64	SSE2
	_mm254_set_pd	AVX	_mm256_set_epi64x	AVX
	_mm512_set_pd	AVX-512F	_mm512_set_epi64	AVX512F
initial setting	_mm_set1_pd	SSE2	_mm_set1_epi64	SSE2
	_mm256_set1_pd	AVX	*	
	_mm512_set1_pd	AVX-512F	_mm512_set1_epi64	AVX-512F
copy lower	*		_mm_extract_epi64	SSE4.1
	*		_mm256_extract_epi64	AVX
	*		*	
selective out	*		mm_extract_epi64	SSE4.1
	_mm256_extractf64x2_pd	AVX-512VL	_mm256_extracti62x4_epi64	AVX-512VL
	_mm512_extractf64x4_pd	AVX-512F	_mm512_extracti64x4_epi64	AVX-512F
store register	_mm_storeu_pd	SSE2	_mm_storeu_epi64	AVX-512F
	_mm256_storeu_pd	AVX	_mm256_storeu_epi64	AVX-512F
	_mm512_storeu_pd	AVX-512F	_mm512_storeu_epi64	AVX-512F

Table 3.12 is the 64-bit analogue of the 32-bit Intel Intrinsics library functions given in Table 3.7. Both floating point and integer arithmetic are covered. The name of the Intrinsics release for each function is also tabulated. An asterisk (*) in the Table indicates no 64 bit Intrinsic function exists to perform the operation.

To enable the code in Figure 3.19 to be used in multiple situations a number of statements were changed. These are denoted in Figure 3.19 by `KEY` and `key`. The `KEY` denotes the `VALUE` added to form the sum. Those values being 3 and 3456 in the integer processing version of the program, while 3.0 and 3456.0 were used in the floating point version.. The `key` denote statements which were changed to change the program from performing integer, or alternately, floating point addition. Those statements are contained in Table 3.13. This same `key` is used in Table 3.14 to link to the results obtained from such statements.

```
/* On the MacPro 2019 measure the execution time of intrinsic functions.
 *
 * Codedy by: Ross Maloney
 * Date:      October 2020
 *
 *  gcc -fopenmp -mavx512bw intrtiming.c
 */

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

#define REPEAT 1000000000
#define VALUE  3456.0                                          // KEY

int main(int argc, char *argv[])
{
  int i;
  __m128d sum, a;                                              // Key
  double *result;                                              // Key
  double ticks;

  ticks = omp_get_wtime();

  sum = _mm_setzero_pd();
  a = _mm_set_pd(VALUE, VALUE);                                // Key
  for (i = 0; i < REPEAT; i++)
    sum = _mm_add_pd(sum, a);                                  // Key


  ticks = omp_get_wtime() - ticks;
  result = (double *)&sum;                                     // Key
  printf("sum = %lf\n", result[0]);                            // Key
  printf("elapse time:; %lf milli-sec\n", 1000.0*ticks);
  return(0);
}
```

Figure 3.19: Code to measure arithmetic behaviour

The program in Figure 3.19 was use to measure the execution of both integer and floating point arithmetic. Performing 10^9 (a billion) additions of the one value to produce a single sum was used to characterize the arithmetic. The measured execution time obtained included the loop controlling the addition process. But this control was common to all measurements so difference in execution times should reflect the relative timing of the summation operator used. The number of loop repartitions (`REPEAT`) was chosen to give a duration measure of adequate length.

To enable the code in Figure 3.19 to be used in multiple situations a number of statements were changed. These are denoted in Figure 3.19 by `KEY` and `key`. The `KEY` denotes the `VALUE` added to form the sum. Those values being 3 and 3456 in the integer processing version of the program, while 3.0 and 3456.0 were used in the floating point version.. The `key` denote statements which were changed to change the program from performing integer, or alternately, floating point addition. Those statements are contained in Table 3.13. This same `key` is used in Table 3.14 to link to the results obtained from such statements.

Table 3.13: MacPro 2019 Intrinsic functions used for determining behaviour

key	vector	variable type	sum initialized by	a =	sum calculated by
AA	int	32bit integer	0	NA	+= VALUE
BB	int64_t	64bit integer	0	NA	+= VALUE
CC	float	32bit float	0.0	NA	+= VALUE
DD	double	64bit float	0.0	NA	+= VALUE
EE	long double	64+bit float	0.0	NA	+= VALUE
A	__m512i	16x32bit integer	_mm512_setzero_epi32()	_mm512_set_epi32()	_mm512_add_epi32()
B	__m512i	8x64bit integer	_mm512_setzero_si512()	_mm512_set_epi64()	_mm512_add_epi64()
C	__m256i	8x32bit integer	_mm256_setzero_si256()	_mm256_set_epi32()	_mm256_add_epi32()
D	__m256i	4x64bit integer	_mm256_setzero_si256()	_mm256_set_epsi64x()	_mm256_add_epi64()
E	__m128i	4x32bit integer	_mm_setzero_si128()	_mm_set_epi32()	_mm_add_epi32()
F	__m128i	2x64bit integer	_mm_setzero_si128()	_mm_set_epi64x()	_mm_add_epi64()
G	__m64	2x32bit integer	_mm_setzero_si64()	_mm_set_pi32()	_mm_add_pi32()
H	__m512	16x32bit float	_mm512_setzero_ps()	_mm512_set_ps()	_mm512_add_ps()
I	__m512d	8x64bit float	_mm512_setzero_pd()	_mm512_set_pd()	_mm512_add_pd()
J	__m256	8x32bit float	_mm256_setzero_ps()	_mm256_set_ps()	_mm256_add_ps()
K	__m256d	4x64bit float	_mm256_setzero_pd()	_mm256_set_pd()	_mm256_add_pd()
L	__m128	4x32bit float	_mm_setzero_ps()	_mm_set_ps()	_mm_add_ps()
M	__m128d	2x64bit float	_mm_setzero_pd()	_mm_set_pd()	_mm_add_pd()

Another use of the number of repetitions (10^9) used in the program of Figure 3.19 followed from the sum produced. This `VALUE` was printed and is shown in the results of Table 3.14. It shows the influence of the number of bits used in the calculation and the result obtained. When the `VALUE` being added was 3, the correct sum is 3000000000. When the `VALUE` was 3456, the correct sum is 3456000000000. If multiplication had been used, a repetition of 31 would have caused problems with single 32 bit arithmetic, and 63 repetitions with 64 bit arithmetic.

From the Intrinsics functions shown in Table 3.12 those appropriate for a particular processing type were selected and used in the program of Figure 3.19. Each selected group was assigned a `key` as an identifier. Those identifiers as used in Tables 3.13 and 3.14. In Table 3.13 the detail of each group of Intrinsics functions used is given. In Table 3.14 the results obtained from each such combination is tabulated from insertion in the program of Figure 3.19. The timing results were obtained from 25 executions of such Intrinsics/program combinations.

In Tables 3.13 and 3.14 the `key` have one or two letters. A single letter key denotes linkage with Intrinsics functions. Two letter keys denote linkage with single variable functions, i.e. a *standard* serial computer instruction. In the contact of multiple values being processed by Intrinsics functions, these serial instructions were considered as parallel processing functions but handling a single value. Such they contrast standard serial processing with parallel processing.

In Table 3.13 note there is no corresponding 32 bit floating point counterpart of the 32 bit integer Intrinsics of `key` G (vector type `_m64`). Key EE, denoted as `long double`, is quadruple double precision which on the MacPro 2019 is implemented in software, not hardware as is the case with all other implementations. This format occupies 16 bytes and is also called `binary128`.

Table 3.14 shows the execution time results. The results are divided into those for integer and floating point processing. The calculated sum is seen to be incorrect in 32 bit processing, for both integer and floating point processing. Although the calculated result was in error, the execution time was taken as correct. In all cases a result had been produced from the correct execution of the computer instruction

producing correct execution time, in contract to the incorrect result arising from the finite nature of the arithmetic being performed.

Table 3.14: MacPro 2019 single and double precision Intrinsic arithmetic behaviour

key	VALUE	sum calculated	min	mean	max
AA	3	-1294967296	1553.6	1560.5	1566.1
AA	3456	-1448673280	1555.3	1561.9	1572.3
BB	3	3000000000	1551.1	1562.6	1587.5
BB	3456	3456000000000	1552.7	1562.0	1586.2
CC	3.0	67108864.000	2070.9	2076.9	2097.5
CC	3456.0	68719476736.000	2073.7	2077.2	2088.8
DD	3.0	3000000000.000	2071.0	2075.6	2100.6
DD	3456.0	3456000000000.000	2071.1	2075.0	2096.3
EE	3.0	3000000000.000	2528.5	2538.3	2558.8
EE	3456.0	3456000000000.000	2529.6	2538.2	2561.5
A	3	-1294967296	4248.7	4254.3	4274.2
A	3456	-1448673280	4247.8	4252.7	4273.3
B	3	3000000000	4249.0	4253.7	4270.6
B	3456	3456000000000	4247.2	4256.0	4273.5
C	3	-1294967296	3295.8	3337.6	3429.7
C	3456	-1448673280	3289.9	3324.4	3435.8
D	3	3000000000	3377.6	3526.5	3696.3
D	3456	3456000000000	3363.5	3542.8	3672.4
E	3	-1294967296	2666.1	3173.5	3618.0
E	3456	-1448673280	2730.3	3047.1	3451.5
F	3	3000000000	2893.7	3030.2	3215.3
F	3456	3456000000000	2932.6	3077.9	3211.8
G	3	-1294967296	3134.6	3185.3	3244.5
G	3456	-1448673280	3120.6	3175.3	3243.5
H	3.0	67108864.000	4515.5	4519.0	4536.8
H	3456.0	68719476736.000	4513.4	4520.1	4540.1
I	3.0	3000000000.000	4513.7	4520.1	4540.1
I	3456.0	3456000000000.000	4515.5	4520.3	4538.2
J	3.0	67108864.000	3674.4	3679.3	3698.8
J	3456.0	68719476736.000	3674.7	3678.0	3681.4
K	3.0	3000000000.000	3674.2	3681.8	3700.9
K	3456.0	3456000000000.000	3674.7	3678.3	3682.6
L	3.0	67108864.000	3216.1	3226.6	3254.2
L	3456.0	68719476736.000	3220.6	3231.2	3251.4
M	3.0	3000000000.000	3228.8	3272.7	3325.7
M	3456.0	3456000000000.000	3226.6	3278.3	3335.5

Figures 3.20 and 3.21 display the integer and floating point mean execution times, respectively, of Table 3.14 when using repeated adding 3 to form the sum. In each case the results are grouped into 32 bit and 64 bit processing. The quadruple precision denoted by `+64bit` is included with the floating point, it being part of *serial* floating point in Table 3.14. These Figures show the similarity of integer and floating point behaviour in the data of Table 3.14. The data in Table 3.14 indicates that using 3 or 3456 for the summing value produced the same behaviour, except when using Intrinsics functions of 4 32 bit integers (key E). Thus Figures 3.20 and 3.21 are representative of integer and floating point arithmetic behaviour with different precisions and Intrinsics groupings.

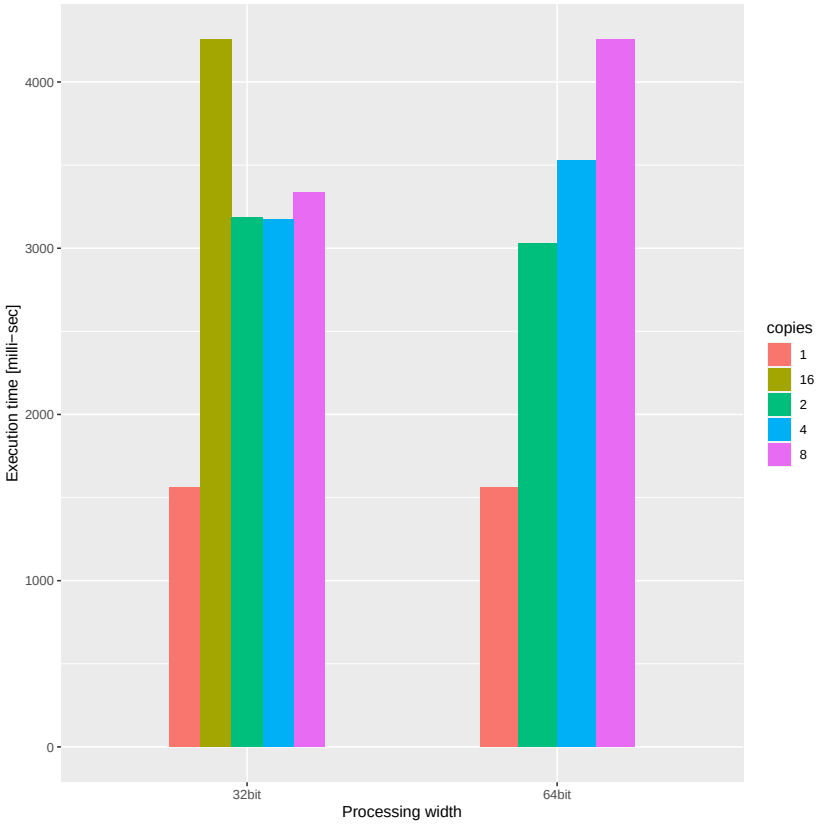

Figure 3.20: Comparison of integer addition processing speeds by repeated adding 3

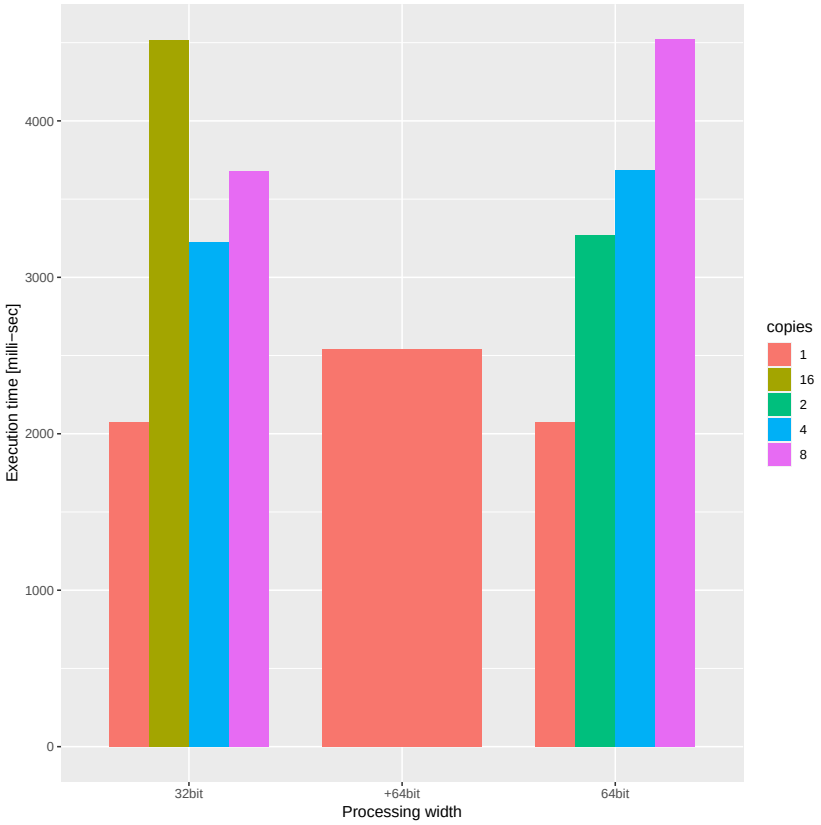

Figure 3.21: Comparison of floating point addition processing speeds by repeated adding 3

From the data in Table 3.14 and Figures 3.20 and 3.21, the following was observed:

- Single and double precision arithmetic for both integer and floating point have the same execution times;
- Integer arithmetic executes faster than floating point;
- Quadruple precision floating point arithmetic is slower to execute than serial floating point instructions by a factor of 25%;
- Execution times for single and double precision arithmetic for both integer and floating point is not affected by the size of the values being processed;
- The execution times using Intel Intrinsics are significantly slower than using serial processing for both integer and floating point arithmetic, per computation;
- For both integer and floating point, as the number of elements in an Intrinsics vector decreased, the execution time also decreased;
- The data suggests Intrinsics are better used when greater than 2 values are stored in the vector which is processed.

Although Intrinsics are slower to execute, there parallel execution of multiple values may compensate for this observation.

3.4.2 By using CUDA

CUDA, which means Compute Unified Device Architecture, is an outgrowth of graphics processing. It enables a computer program to be written which will execute in parallel on a Nvidia graphics card which supports CUDA. In this work a GeForce RTX 2070 super graphics card which supported CUDA was used. This card was mounted internally in a MacPro 2019 which controlled the execution of the program of Figure 3.22.

A CUDA program is written in a language called CUDA C which is like standard C with some additional statements and manners of doing things. In particular the parallel execution part is done using a procedure which is denoted by one such additional statement. Memory has also needs to be allocated on the graphics card and this also involves new, or modification, to standard C statements. CUDA C is not portable to anything other than Nvidia graphics cards. Even with such limitations, CUDA offers a popular manner of performing parallel processing.

Although there is a cost in obtaining the CUDA aware graphics card, the compiler for CUDA C is free together with documentation and tutorials on the language. Most such documentation and tutorials do not mention limitations of CUDA. A large part of graphics processing is handled by 32 bit floating point arithmetic. Integer arithmetic is also used. Double precision floating point arithmetic is also mention in the documentation on the card. But what are the consequences of using integer and double precision floating point processing relative to single precision floating point under CUDA?

The code of Figure 3.22 was used to measure execution times while doing different types and precision of arithmetic using CUDA. As with the code of Figure 3.14, addition was used to obtain those measures. The code of Figure 3.22 was compiled using the command:

```
nvcc -Xcompiler -fopenmp -lgomp cutiming.cu
```

where `cutiming.cu` was the name of the file with the contents of Figure 3.22. In the code of Figure 3.22 the inclusion of the header file `omp.h` was necessary for the timing using the `omp_get_wtime()` OMP library function. The `nvcc` compiler, is associated libraries, and drivers which enables the compiled code to execute on the CUDA enables graphics card installed on the MacPro 2019 were downloaded from Debian `aptitude`.

```
/*  Measurement of execution of CUDA expression.
 *
 *  coded by:  Ross Maloney
 *  Date:      October 2020
 *
 *  nvcc _compiler -fopenmp -lgomp cutiming.cu
 */

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

#define VALUE 3456
#define REPEAT 1000000000

__global__
void self_add(int64_t *out)
{
  int i;

  *out = 0.0;
  for (i = 0; i < REPEAT; i++)
    *out += VALUE;
}

int main()
{
  int64_t    out;
  int64_t    *d_out;
  double   ticks;

  ticks = omp_get_wtime();

  // allocate device memory
  cudaMalloc((void **)&d_out, sizeof(int64_t));

  // main function
  self_add<<<1,1>>>(d_out);

  // get and print results
  cudaMemcpy(&out, d_out, sizeof(int64_t), cudaMemcpyDeviceToHost);

  ticks = omp_get_wtime() - ticks;

  printf("sum = %ld\n", out);
  printf("Elapse time: %lf milli-sec\n", 1000.0*ticks);

  // cleaning up
  cudaFree(d_out);
}
```

Figure 3.22: Computer code used to measure execution timining of CUDA repeated addition

CUDA is based on CUDA blocks and threads. Each block has a single processor and a number of threads. That processor can process in parallel a number of threads contained in it's block. This number is the `warp` size which is generally set on any CUDA enabled graphics card as 32. Each thread contains a program which uses memory om the graphics card. Each CUDA card imposes a *maximum* number of blocks, threads in those blocks, and the amount of memory each thread can access. Scheduling the execution of warps on each block is done by the graphics card itself. Data to be processes, data as results, and the programs for each thread to execute must be passed by communication between the

master processor, in this case the MacPro 2019, and the CUDA enabled graphics card. The time to do such communication must be taken into account in measuring the execution time of the CUDA card.

In the CUDA C program of Figure 3.22 the `__global__` statement signifies the following function (`self_add()`) is for execution on threads of the CUDA card. The `self_add<<<1,1>>>` statement calls for execution of the `self_add()` function. This statement indicates execution is to be done on one block of the CUDA card using one thread. The data memory to be used is allocated on the CUDA card by the `cudaMalloc()` call, and the result, the sum, is retrieved from the CUDA card by the `cudaMemcpy()` call. The value to be added and the number of repartitions of additions to be performed are coded into the `self_add()` function. The storage type `int64_t` and the value of the constant `VALUE` were changed to give the execution timing data.

Table 3.15: CUDA single variable execution behaviour on a MacPro/RTX2070 combination

type	variable type	VALUE	sum calculated	min	mean	max
int	32bit integer	3	-1294967296	121.0	139.9	154.2
int	32bit integer	3456	-1448673280	124.7	138.7	155.8
int64_t	64bit integer	3	3000000000	128.3	139.9	158.8
int64_t	64bit integer	3456	3456000000000	124.3	137.0	151.0
float	32bit float	3.0	67108864.000	2231.9	2247.5	2267.0
float	32bit float	3456.0	68719476736.000	2233.1	2252.8	2263.4
double	64bit float	3.0	3000000000.000	22712.0	22834.8	22929.6
double	64bit float	3456.0	3456000000000.000	22724.4	22849.5	22923.7

Table 3.15 shows the results of executing the program of Figure 3.22 on a MacPro 2019 on which a GeForce RTX 2070 super CUDA enabled graphics card was installed. These results are for 25 successive runs of the compiler program from which the minimum and maximum execution was recorded while calculating the mean execution time printed by the program.

From the data in Table 3.15 the following was observed:

- 32 bit integer execution times were significantly faster than those in Table 3.14;
- 64 bit integer execution times are the same as 32 bit integer with increased precision in the calculation;
- The same wrong values for the sum using 32 bit integer and floating point addition as in Table 3.14 were obtained;
- 32 bit floating point addition is marginally slower than serial 32 bit floating point of Table 3.14;
- 64 bit floating point is significantly slow;
- 128 bit floating point is not currently available.

This data suggests CUDA enabled graphics cards, or at least the RTX 2070 super, is dominant when using 32 bit and 64 bit integer arithmetic, or addition at the least.

3.4.3 Summary of double precision for parallel computation

From the data obtained the following recommendations can be made with respect to use in parallel computation, at least from the perspective of repeated addition:

- For 32 bit and 64 bit integer computing, a CUDA enabled graphics card is best;
- For 64 bit floating point computation MacPro core usage is the best;

- 64 bit floating point and integer computation on MacPro cores has the same computational speed as their respective 32 bit counterpart;
- For 32 bit floating point computation, Intrinsics functions behaviour is same as CUDA enabled graphics cards, i.e. for 32 bit float point computation Intrinsics functions on McPro 2019 cores are the equal of CUDA graphics cards;
- Intrinsics functions should have 4 or more values in their vector.

3.5 Programming to use all computational assets

Parallel programming should to use all computing assets of the available computer hardware. An Intel core has Single Instruction Multiple Data (SIMD) capability via Intrinsics instructions which are embodied in function calls. If there are multiples of such cores: Can each of those multiple cores execute Intrinsic instructions simultaneously? Can the programmer direct the individual core upon which Intrinsics instructions be executed? A CUDA enabled graphics card also has multiple processors, one processor embedded in each CUDA block. How can the programmer control the use of such multiple processors? The questions are addressed in this Section.

Programming languages such as OpenMP and OpenACC do such control at a higher level of abstraction than what is considered here. As such, this Section provides a foundation upon which such programming languages can be assessed. Here again the degree of parallel programming achieved is measured by execution time of programs.

From the data in Table 3.9 a speedup for 32 bit floating point SIMD of approximately 4.7 was obtained. From the data in Table 3.8 a speedup for 32 bit integer SIMD of approximately 4.2 was obtained. These speedups were obtained using a single thread (processor) which is only a partial use of the computational resources of a multi-core computer.

In the following double precision floating point SIMD is used. CUDA can perform double precision floating point operations but this capacity is less impressive than it's single precision processing capacity. However, for a large number of scientific/engineering computations, double precision is required. Here, building on available SIMD capacity through multi-cores is of interest here.

3.5.1 Intrinsics with pthreads

The MacPro 2019 used here had 28 cores. Using the Linux command:

```
less /proc/cpuinfo
```

it was seen there be 56 `processors` present numbered 0 to 55. Each of those processors were indicated as `GenuineIntel` each have `flags` of avx, avx2, avx512f, avx512dq, avx512cd, avx512bw, avx512vl, and avx512_vnni together with sse, sse2, ssse3, sse4_1, and sse4_2. Those flags indicate SIMD processing capacity on that processor. The `flags` with names commencing with avx512 indicate SIMD processing on a vector of 512 bits. Such a SIMD vector would contain 16 32 bit float point or integers values, or 8 64 bit floating point or integer values. All the values in such a vector would be processed together. The same information followed from using the command `lscpu`.

If all 56 processors were used on the MacPro 2019 in a program, that program would execute 896 ($= 56 * 16$) times faster than a program using serial computing of 32 bit floating point or integer values. If using 64 bit floating point or integer values, the speedup would be 448 ($= 56 * 8$). These are theoretical speedups. Contrast these speedups when using a single processor. Using a single processor, the 32 bit speed up is 16 and the 64 bit speedup is 8.

The CPU information uses the word `processor` in place of threads. With Intel hyper-threading,

each core generates 2 threads. The MacPro 2019 was understood to have 28 cores. Thus 56 threads. So each thread can process a SIMD vector.

In the data of Table 3.16 the repetitions started at 56 and then were halved. Then 54 was added to step back from the maximum number of processors available. Then 40 repetitions was added to cover the range between 54 and 28.

The program of Figure 3.23 was used to investigate distributing double precision floating point Intrinsics use across multi-processors. A number of approaches to this distribution across processors were considered. The naming of those approaches shown in Table 3.16 and Figure 3.24 were composed two parts; a prefix and a suffix. The prefix and suffix with their associated meanings are;

	pthread_		CPU_ZERO()	CPU
Prefix	create()	attr_setaffinity_np()	and CPU_SER()	distribution
random	used NULL	not used	not used	random
CPUall	used NULL	not used	used	random/multiple
CPUin	used attr	used	not used	uniform
serial	not used	not used	not used	one CPU only
stdomp	not used	not used	not used	uniform

Suffix	in combine()
No	no intrinsics
Wi	Intrinsics used

The *No* suffix approach enabled measurement of thread creation overhead.

A printf() function (not shown) was used in combine() to determine the CPU distribution generated. It was not included in the program runs used to generate the data of Table 3.16. The pthread_attr_init(&attr) call was included in all program runs.

Figure 3.24 plots the data of Table 3.16 obtained by running the program of Figure 3.23 having the modifications to perform under the different thread/core(processor) distributions considered. From this data the following patterns were observed:

1. The *No* and *Wi* data of each distribution technique need to be regarded as a pair.
2. No distribution technique resulted in less execution time than the serial execution of the combine() function.
3. The *random* and *CPUall* distribution behaved the same.
4. The *CPUin* pair had the greatest execution time.
5. The *stdomp* distribution had an approximate constant time to initiate all thread numbers in use.
6. In each distribution pair, the difference between the *Wi* and *No* execution time for a given thread use (repeats) was approximately the same as that for the *serial* pair. This was not true for the *stdomp* pair.

```
/*  Multiply more than one row/column matrix pairs
 *
 *  gcc -fopenmp -mavx512bw scratch.c -lpthread
 */

#include  <omp.h>
#include  <stdio.h>
#include  <immintrin.h>
#include  <pthread.h>
#include  <time.h>

#define  SIZE 8

double     am[1][9] = { {2.0, 67.0, 12.0, -23.0, 14.0, 7.0, -7.0, 23.0, 12.0} };
double     bv[1][9] = { {23.0, 10.0, 2.0, 6.0, 8.0, 23.0, 12.0, -23.0, 11.0} };
int        gulp;
__m512d    row, column, temp;                    // for Mac Pro 2019
double     hold;

int  main(int argc, char** argv)
{
  double    ticks;
  int       i, repeats;
  pthread_attr_t  attr;
  pthread_t thread[56];
  int       iret[56];
  void      *combine(void *ptr);

  repeats = atoi(argv[1]);
  printf("repeats = %d\n", repeats);
  pthread_attr_init(&attr);

  ticks = omp_get_wtime();

  hold = 0.0;
  gulp = ( 1 << SIZE ) - 1;
  for (i = 0; i < repeats; i++)  {
    iret[i] = pthread_create( &thread[i], NULL, combine, NULL);
    pthread_join (thread[i], NULL);
  }

  ticks = omp_get_wtime() - ticks;
  printf("Elapse time: %lf milli-sec\n", ticks*1000.0);
  return(0);
}

void  *combine(void *ptr)
{
  int     i;
  extern  __m512d    row, column, temp;                  // for Mac Pro 2019
  extern  int        gulp;
  extern  double     am[1][9], bv[1][9];
  extern  double     hold;

      /* Mac Pro 2019 double precision Intrinsics */
  for (i = 0; i < 1000; i++)  {
    row = _mm512_maskz_loadu_pd(gulp, (const __m512 *)&am[0][0]);     // AVX-512F
    column = _mm512_maskz_loadu_pd(gulp, (const __m512 *)&bv[0][0]); // AVX-512F
    temp = _mm512_mul_pd(row, column);                                // AVX-512F
    hold = _mm512_reduce_add_pd(temp);                                // AVX-512F
  }
}
```

Figure 3.23: Program to measure matrix dot product Intrinsics execution using pthreads and others

Table 3.16: Execution times using standard pthread approaches

Approach used	Repeats	Execution time [milli-sec] min	mean	max
randomNo	56	2.62	3.04	3.50
randomNo	54	2.73	3.03	3.18
randomNo	40	2.07	2.27	2.54
randomNo	28	1.47	1.58	1.82
randomNo	14	0.79	0.90	1.07
randomNo	7	0.48	0.55	0.65
randomNo	3	0.29	0.33	0.39
randomNo	1	0.14	0.19	0.21
randomWi	56	4.31	4.60	4.90
randomWi	54	4.27	4.69	9.65
randomWi	40	3.19	3.42	3.67
randomWi	28	2.23	2.39	2.75
randomWi	14	1.14	1.30	1.70
randomWi	7	0.65	0.75	0.85
randomWi	3	0.32	0.47	0.61
randomWi	1	0.21	0.24	0.28
CPUallNo	56	2.80	3.02	3.33
CPUallNo	54	2.70	2.92	3.32
CPUallNo	40	2.08	2.29	2.84
CPUallNo	28	1.42	1.63	1.84
CPUallNo	14	0.82	0.88	1.12
CPUallNo	7	0.48	0.56	0.71
CPUallNo	3	0.22	0.35	0.59
CPUallNo	1	0.13	0.19	0.21
CPUallWi	56	4.25	4.61	4.85
CPUallWi	54	4.40	4.54	4.74
CPUallWi	40	3.11	3.44	3.70
CPUallWi	28	2.20	2.43	2.63
CPUallWi	14	1.14	1.31	1.58
CPUallWi	7	0.64	0.82	1.11
CPUallWi	3	0.38	0.45	0.51
CPUallWi	1	0.17	0.24	0.29
CPUinNo	56	6.73	7.08	7.68
CPUinNo	54	6.62	6.87	7.38
CPUinNo	40	4.87	5.21	5.46
CPUinNo	28	3.46	3.67	3.99
CPUinNo	14	1.83	1.94	2.39
CPUinNo	7	0.87	1.04	1.48
CPUinNo	3	0.36	0.44	0.50
CPUinNo	1	0.14	0.21	0.27
CPUinWi	56	8.64	9.01	9.51
CPUinWi	54	8.40	8.83	13.36
CPUinWi	40	6.25	6.53	6.92
CPUinWi	28	4.45	4.63	5.02
CPUinWi	14	2.15	2.32	2.48
CPUinWi	7	1.18	1.28	1.37
CPUinWi	3	0.53	0.63	0.72
CPUinWi	1	0.21	0.28	0.72
serialNo	56	0.29	0.32	0.34
serialNo	54	0.30	0.31	0.43
serialNo	40	0.22	0.23	0.32
serialNo	28	0.14	0.15	0.16
serialNo	14	0.07	0.08	0.11
serialNo	7	0.04	0.04	0.04
serialNo	3	0.02	0.02	0.02
serialNo	1	0.01	0.01	0.01

Continued on next page

Approach used	Repeats	Execution time [milli-sec] min	mean	max
serialWi	56	1.64	1.77	2.47
serialWi	54	1.74	1.85	2.63
serialWi	40	1.29	1.43	1.62
serialWi	28	0.84	0.95	1.52
serialWi	14	0.42	0.54	0.72
serialWi	7	0.26	0.29	0.50
serialWi	3	0.15	0.15	0.18
serialWi	1	0.03	0.07	0.08
stdompNo	56	1.93	3.42	12.21
stdompNo	54	2.15	2.63	11.59
stdompNo	40	2.11	2.42	4.79
stdompNo	28	1.96	3.04	12.02
stdompNo	14	1.89	2.61	11.72
stdompNo	7	1.90	2.97	12.21
stdompNo	3	1.92	3.54	12.84
stdompNo	1	1.94	2.34	5.96
stdompWi	56	7.43	8.17	13.97
stdompWi	54	7.54	9.47	17.53
stdompWi	40	7.34	8.17	17.15
stdompWi	28	6.93	7.50	12.08
stdompWi	14	4.47	5.39	13.33
stdompWi	7	3.22	3.85	13.48
stdompWi	3	2.47	3.90	12.75
stdompWi	1	2.02	3.11	12.17

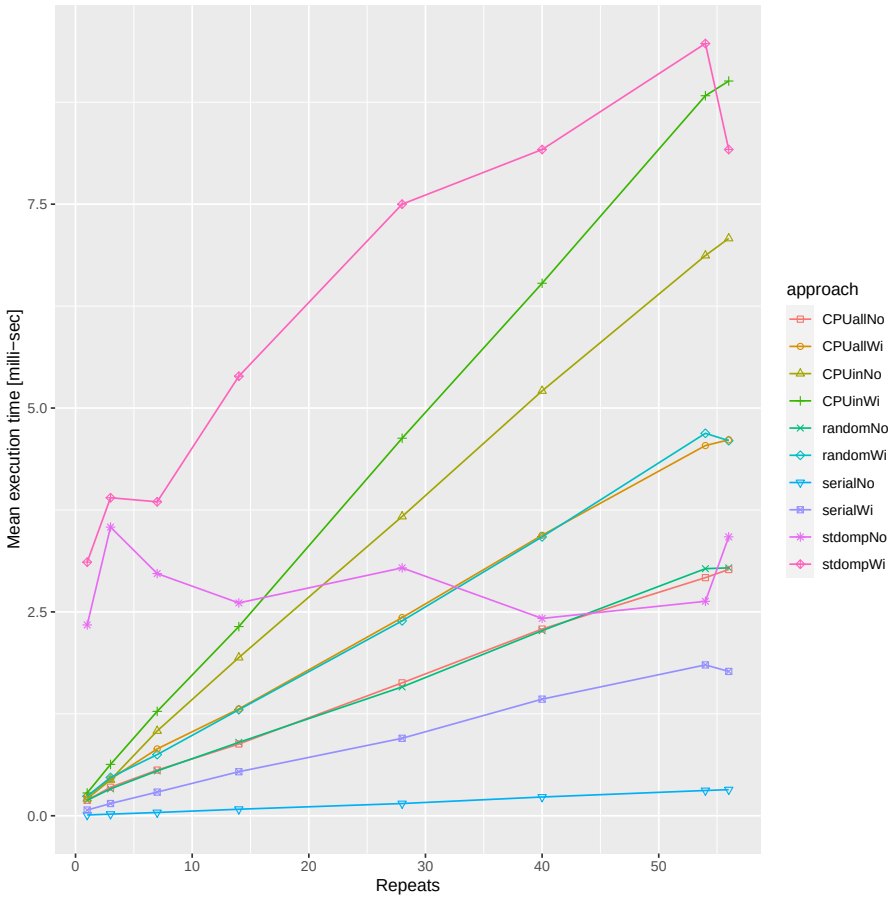

Figure 3.24: Matrix dot product execution times using pthreads and others

From those observations of the data, the following actions were taken:

- Due to the high execution times of the *CPUin* pair, no further consideration was given to using the `pthread_attr_set_affinity_np()` functioni for thread allocation.
- Because the near identical behaviour of the *CPUall* and *random* pairs, the `CPU_ZERO()` and `CPU_SER()` macros received no further consideration for reducing execution time.

The approaches denoted by the *random* prefix using a NULL in the `pthread_create()`, and the `pthread_join()` is shown in Figure 3.23 as having the better performance. However, the increasing execution times with increasing number of threads(processors) in use did not suggest parallel execution was occurring using pthreads in the configurations used. The standard OpenMP with no Intrinsics (stdompNO) was the exception to this behaviour but with little reduction in execution time when using greater numbers of threads.

3.5.2 Intrinsics with Cilk

Initially `Cilk` was obtained from the source distribution `cilk-5.4.6.tar.gz` downloaded from `supertech.csail.mit.edu/cilk`. After unpacking, `./configure` was run successfully but the subsequent `make` resulted in errors relating to a `PrintChar` routine. From the `print-ast.c` file, the `PrintChar` function was copied into the `output.c` file before the `output_constant` function. Then in the `printf-ast.c` file, the two occurrences of `PrintfChar` were changed to `PrintChar1`. Then `make` and `make install` were successful. Despite the errors indicated, the install worked successfully.

The PDF manual for this distribution was downloaded from the same URL. Compiling and running of the `fib.c` program in that manaul, together with exploring the `stat` and `profile` options described in the manual, indicated `Cilk` had potential.

This `Cilk` distribution was posted in 1994. `OpenCilk` had replaced it. Where as the 1994 `Cilk` was mated with `gcc`, `OpenCilk` was linked to `clang`. The distribution `OpenCilk-9.0.1-Linux.sh` of `OpenCilk` was downloaded from `github.com/OpenCilk/opencilk-project/releases/tag/opencilk/beta3`. This was a binary distribution for Intel computers operating under Linux. This file was executed on the `/opt` directory, on which is deposited the urlOpenCilk-9.0.1-Linux/ directory which contained `bin/`, `include/`, `lib/`, `libexec/` and `share/` subdirectories. The directory `/opt/OpenCilk-9.0.1-Linux/bin` was added to the `PATH` environment variable. The `/usr/local/include/cilk` subdirectory, left from the installation of the 1994 `Cilk` distribution, was deleted. `OpenCIlk` was composed as an attachment for the `clang` compiler.

`OpenCIlk` was the developing version of `Cilk` and as such it received the most consideration here. Both the `OpenCilk` and `Cilk` distributions contained a compiler extension for compiling user programs together with a runtime environment in which that compiled code ran. In the case of the 1994 distribution, the package code was written in C, while C++ was extensively use in the `OpenCilk` distribution. The PDF manual of the 1994 distribution indicated parameters which could be used while running user `Cilk` programs. This include specifying the number of processors to use. It was only via a tutorial example given on a slide downloaded from the Internet that the `CILK_NWORKERS` parameter was discovered for specifying that number of processes to use with a `OpenCilk` program.

One difficulty with `OpenCilk` was it did not support an `OpenMP` library. Thus an alternative to the `omp_get_wtime()` function used to measure execution timing was required which had the same behaviour characteristics was required. Inspection of the `GCC` source code which did support and implement `omp_get_wtime()` indicated it was based on the POSIX `timespec` structure. Linux supported such a structure. The code of Figure 3.26 was written and used in place of `omp_get_wtime()`.

```
/*  Cilk Intel Intrinsics multiplication of two square matrices of double
 *  precision floating point values.
 *
 *  clang -fopencilk -O3 cilkfmatavx.c -mavx512dq -o cilkfmatavx
 *  CILK_NWORKERS=2 ./cilkfmatavx
 *
 *  Coded by:  Ross Maloney
 *  Date:      26 November 2020
 */

#include <cilk/cilk.h>
#include <stdio.h>
#include <immintrin.h>
#include <time.h>

#define DIM  2000
#define SIZE 8     // AVX512

double am[DIM][DIM], bv[DIM][DIM], result[DIM][DIM];

int main(int argc, char** argv)
{
  double    ticks;
  int       i, j, gulp, ii;
  __m512    row, column, hold, temp;                  // for Mac Pro 2019
  double    my_get_wtime(void);

  srand(time(NULL));
  cilk_for (int k = 0; k < DIM; k++)
    for (j = 0; j < DIM; j++)  {
      am[k][j] = rand() % 10000;
      am[k][j] = am[k][j] - 5000.0;
      bv[j][k] = rand() % 10000;
      bv[j][k] = bv[j][k] - 5000.0;
    }
  ticks = my_get_wtime();

  cilk_for (int k = 0; k < DIM; k++)  {
    for (j = 0; j < DIM; j++)  {
      hold = _mm512_setzero_pd();
      gulp  = ( 1 << SIZE ) - 1;
      for (i = 0; i <= DIM - SIZE; i += SIZE)  {
        row = _mm512_maskz_loadu_pd(gulp, (const __m512 *)&am[k][i]);
        column = _mm512_maskz_loadu_pd(gulp, (const __m512 *)&bv[j][i]);
        temp = _mm512_mul_pd(row, column);
        hold = _mm512_add_pd(hold, temp);
      }
      gulp = ( 1 << (DIM - i) ) - 1;
      row = _mm512_maskz_loadu_pd(gulp, (const __m512 *)&am[k][i]);
      column = _mm512_maskz_loadu_pd(gulp, (const __m512 *)&bv[j][i]);
      temp = _mm512_mul_pd(row, column);
      hold = _mm512_add_pd(hold, temp);
      result[k][j] = _mm512_reduce_add_pd(hold);
    }
  }

  ticks = my_get_wtime() - ticks;
  printf("DIM = %d   Elapse time: %lf milli-sec\n", DIM, 1000.0*ticks);
  return(0);
}
```

Figure 3.25: Combination of OpenCilk and Intel Intrinsicss in code to multiply two square double precision matrices

Table 3.17: Execution times using OpenCilk and Intrinsics to multiply two double precision matrices

			Execution time [milli-sec]			Speed up	
Program	DIM	Procs	min	mean	max	Serial	Vector
fmatrixrr.c	100	1	3.04	3.12	3.24		
fmatavx.c	100	1	0.39	0.43	0.65	7.3	
cilkfmatavx.c	100	1	0.42	0.43	0.44	7.3	1.0
	100	2	0.25	0.29	0.32	9.8	1.5
	100	3	0.23	0.24	0.28	11.1	1.8
	100	4	0.17	0.21	0.24	14.9	2.1
	100	5	0.16	0.19	0.23	16.4	2.3
	100	10	0.13	0.18	0.21	17.3	2.4
	100	20	0.11	0.17	0.22	18.4	2.5
	100	30	0.13	0.19	0.30	16.4	2.3
	100	40	0.19	0.25	0.32	12.5	1.7
	100	50	0.20	0.27	0.36	11.6	1.6
	100	56	0.20	0.27	0.40	11.6	1.6
	100	60	0.18	0.27	0.40	11.6	1.6
fmatrixrr.c	500	1	119.9	121.7	123.5		
fmatavx.c	500	1	56.8	58.2	61.1	2.1	
cilkfmatavx.c	500	1	55.7	56.8	58.0	2.1	1.0
	500	2	24.8	25.5	26.2	4.8	2.3
	500	3	13.0	15.8	18.8	7.7	3.7
	500	4	9.6	10.5	11.9	11.6	5.5
	500	5	5.7	6.8	10.3	17.9	8.6
	500	10	3.4	3.5	3.8	34.8	16.6
	500	20	2.1	2.2	2.3	55.3	26.5
	500	30	1.7	1.7	1.8	71.6	34.2
	500	40	1.4	1.5	1.5	81.1	38.8
	500	50	1.3	1.3	1.4	93.6	44.8
	500	56	1.2	1.3	1.4	93.6	44.8
	500	60	1.2	1.3	1.4	93.6	44.8
fmatrixrr.c	1000	1	890.5	895.9	916.8		
fmatavx.c	1000	1	317.2	321.9	328.9	2.8	
cilkfmatavx.c	1000	1	315.5	316.8	318.6	2.8	1.0
	1000	2	134.1	136.9	139.0	6.5	2.4
	1000	3	68.1	84.0	93.9	10.7	3.8
	1000	4	52.0	57.2	62.8	15.7	5.6
	1000	5	38.9	45.0	51.8	19.9	7.2
	1000	10	25.9	27.8	31.8	32.2	11.6
	1000	20	16.1	17.6	19.2	50.9	18.3
	1000	30	13.3	14.1	14.7	63.5	22.8
	1000	40	12.7	13.5	14.3	66.4	23.8
	1000	50	12.5	13.8	15.0	64.9	23.3
	1000	56	12.0	13.4	14.4	66.9	24.0
	1000	60	12.8	15.0	23.1	59.7	21.5
fmatrixrr.c	2000	1	7226.6	7232.8	7256.5		
fmatavx.c	2000	1	2407.3	2420.5	2446.0	3.0	
cilkfmatavx.c	2000	1	2410.1	2414.0	2416.8	3.0	1.0
	2000	2	1071.5	1100.5	1141.5	6.6	2.2
	2000	3	539.0	668.6	788.3	10.8	3.6
	2000	4	470.9	502.9	576.6	14.4	4.8
	2000	5	358.1	410.6	527.9	17.6	5.9
	2000	10	194.0	223.3	251.9	32.4	10.8
	2000	20	124.9	135.4	141.8	53.4	17.9
	2000	30	103.2	106.8	117.2	67.7	22.7
	2000	40	86.6	91.7	102.4	78.9	26.4
	2000	50	75.7	82.3	96.1	87.9	29.4
	2000	56	65.5	69.6	79.5	103.9	34.8
	2000	60	71.6	85.0	96.3	85.1	28.5

Continued on next page

Program	DIM	Procs	Execution time [milli-sec] min	mean	max	Speed up Serial	Vector
fmatrixrr.c	4000	1	61890.7	64902.8	65525.5		
fmatavx.c	4000	1	30024.4	31940.4	32842.1	2.0	
cilkfmatavx.c	4000	1	30844.9	32301.6	32725.2	2.0	1.0
	4000	2	15744.7	16356.1	16676.4	4.0	2.0
	4000	3	10828.6	11322.7	11510.0	5.7	2.8
	4000	4	8056.3	8546.3	8640.5	7.6	3.7
	4000	5	6462.6	6798.4	6976.9	9.6	4.7
	4000	10	3562.7	3761.1	4058.5	17.3	8.5
	4000	20	2244.6	2501.9	2836.9	25.9	12.8
	4000	30	2254.3	2322.6	2497.8	27.9	13.8
	4000	40	1501.8	1537.6	1587.5	42.2	20.8
	4000	50	1141.6	1200.2	1338.3	54.1	26.6
	4000	56	1058.6	1106.0	1251.2	58.7	28.9
	4000	60	1143.5	1208.1	1271.2	53.7	26.4
fmatrixrr.c	5000	1	121326,2	127460.5	128710.3		
fmatavx.c	5000	1	61831.0	65984.3	67254.4	1.9	
cilkfmatavx.c	5000	1	63354.6	66404.8	66984.5	1.9	1.0
	5000	2	31870.2	33248.7	33980.4	3.8	2.0
	5000	3	20859.9	21862.1	22292.4	5.8	3.0
	5000	4	15652.5	16528.5	16791.4	7.7	4.0
	5000	5	12819.0	13239.9	13674.2	9.6	5.0
	5000	10	6962.0	7283.7	7506.8	17.5	9.1
	5000	20	4073.6	4895.4	5306.8	26.0	13.5
	5000	30	4636.9	4826.8	5236.5	26.4	13.7
	5000	40	3031.3	3143.1	3280.6	40.6	21.0
	5000	50	2340.7	2481.8	2765.0	51.4	26.6
	5000	56	2210.1	2323.6	2423.7	54.9	28.4
	5000	60	2494.2	2698.6	2972.4	47.2	24.5

```
double my_get_wtime()
{
  struct timespec  ts;
  double           time, nano;

  clock_gettime(CLOCK_MONOTONIC, &ts);
  time = (double) ts.tv_sec;
  nano = (double) ts.tv_nsec;
  return(time + nano/1000000000.0);
```

Figure 3.26: Code to measure the time of day from a system supplied wall clock

Figure 3.25 contained the source code used to measure the execution behaviour of Intel Intrinsics distributed over multi-cores using `OpenCilk`. The code multiplies two square matrices of randomly created double precision floating point values. Such values are either positive or negative. This is used as representative of scientific computation. Execution time was measured using the `my_get_wtime()` function of Figure 3.26. Table 3.17 shows the measurements obtained.

Speed up is a computed relative measure. Two measures were computed from the data of Table 3.17 and tabulated there. One was computed relative to the mean execution time of a standard serial implementation of the matrix multiplication without either use of Intel Intrinsics or OpenCilk. This value for each size of the matrices, was divided by the mean measured execution time of the subject `Intrinsics/OpenCilk` code of the corresponding matrix size. This reference code was in a file named `fmatricrc.c` and that name was used as the marker in Table 3.17 of the execution times for this code. This speed up represented the performance of the `Intrinsics/OpenCilk` code in relation to standard matrix multiplication code. These values are under the *Serial* heading of Table 3.17. The other speed up value tabulated under the *Vector* heading of Table 3.17 was computed in a similar

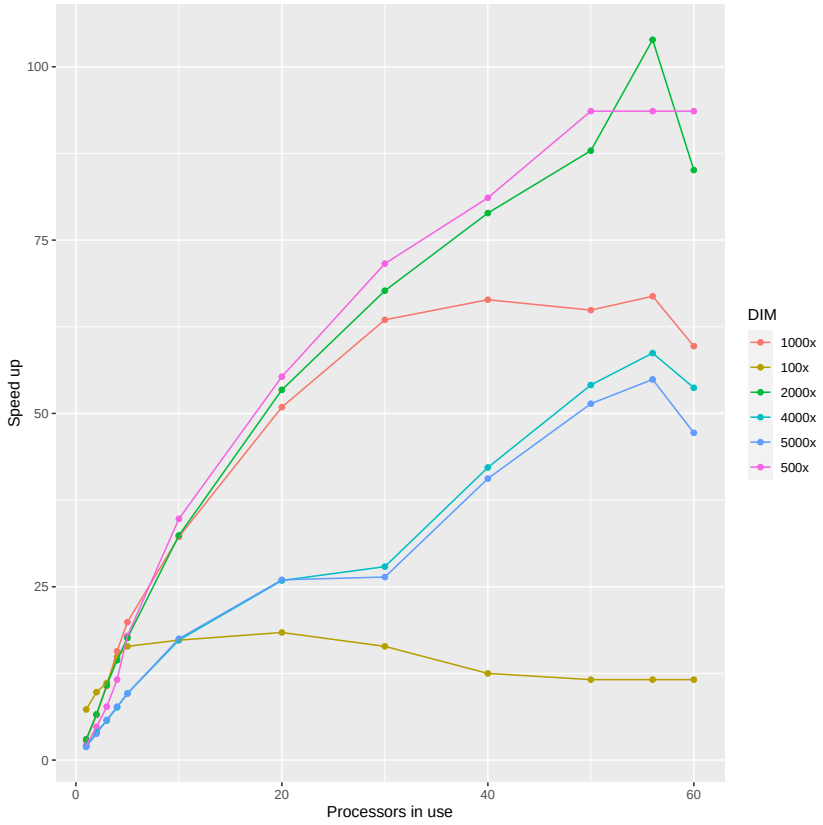

Figure 3.27: Speed up relative to serial of OpenCilk/Intrinsics double precision matrix multiplication

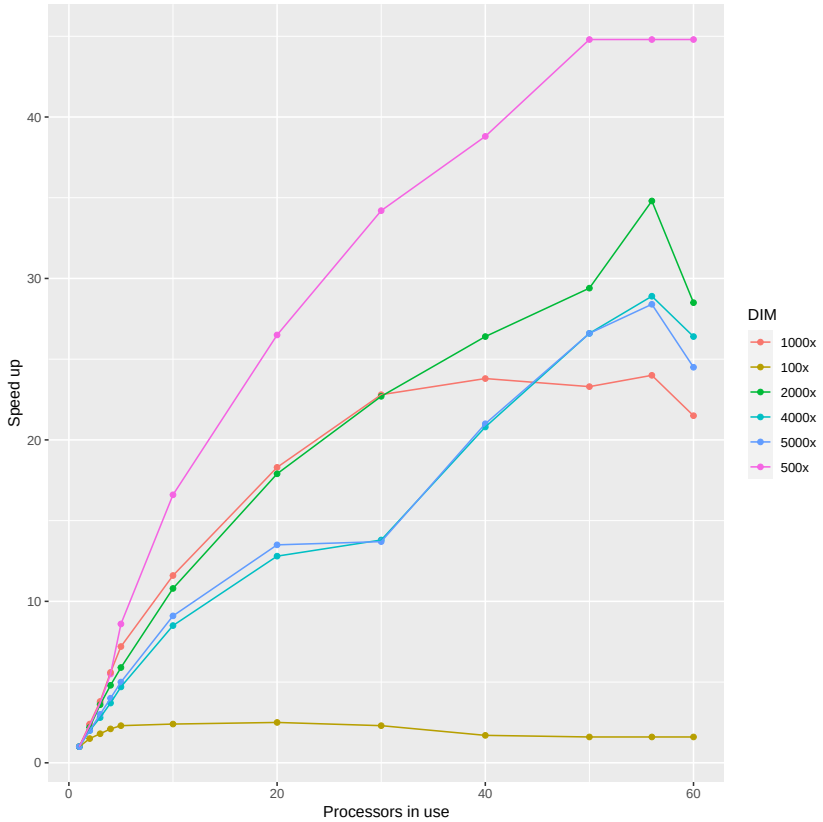

Figure 3.28: Speed up relative to vector of OpenCilk/Intrinsics double precision matrix multiplication

manner. However, the corresponding mean execution time of a code using Intel Intrinsics alone was used as the numerator of of the computed value. The file containing this code was named `fmatavx.c` and used as the marker in Table 3.17 for it's values. This speed up represented how the `Intrinsics/OpenCilk` code performed relative to using the Intel Intrinsics vector processing alone. In both speed up measures, multi-core performance is compared to single core performance.

Figure 3.27 plots the speed up data from under the *Serial* heading of Table 3.17 while Figure 3.28 plots the data under the *Vector* heading. These plots show speed up increases for all matrix sizes when using 2, 3, 4, and 5 processors. Further increases occurred by using more processors except in the case of the 100x100 matrix multiplication; the smallest sized computation considered. This was particularly the case in the *Vector* speed up. In both Figures 3.27 and 3.28 the effect of the limit of 56 processors is reflected in the speed up in value obtained.

Although the shapes of the spreed up curves in Figures 3.27 and 3.28 are similar, their speed up range is significantly different.

Figure 3.29 shows a sample of the consequences of the speed ups of Figure 3.27 on the execution speed of code performing matrix multiplication of double precision values. From Table 3.17, the execution times of the serial (standard) code for performing 2000x2000, 4000x4000 and 5000x5000 matrix multiplication were 7232.8, 64902.8, and 127460.6 milli-sec, respectively.

Only one example has been used here, that of matrix multiplication. But what has been shown is that Intel Intrinsics which are know to implement vector operations on a single core, can also be use across multiple cores. The result is a significant decrease in execution time to achieve the required result. Also, only one of the three `OpenCilk` compiler directives (`cilk_for`) was used.

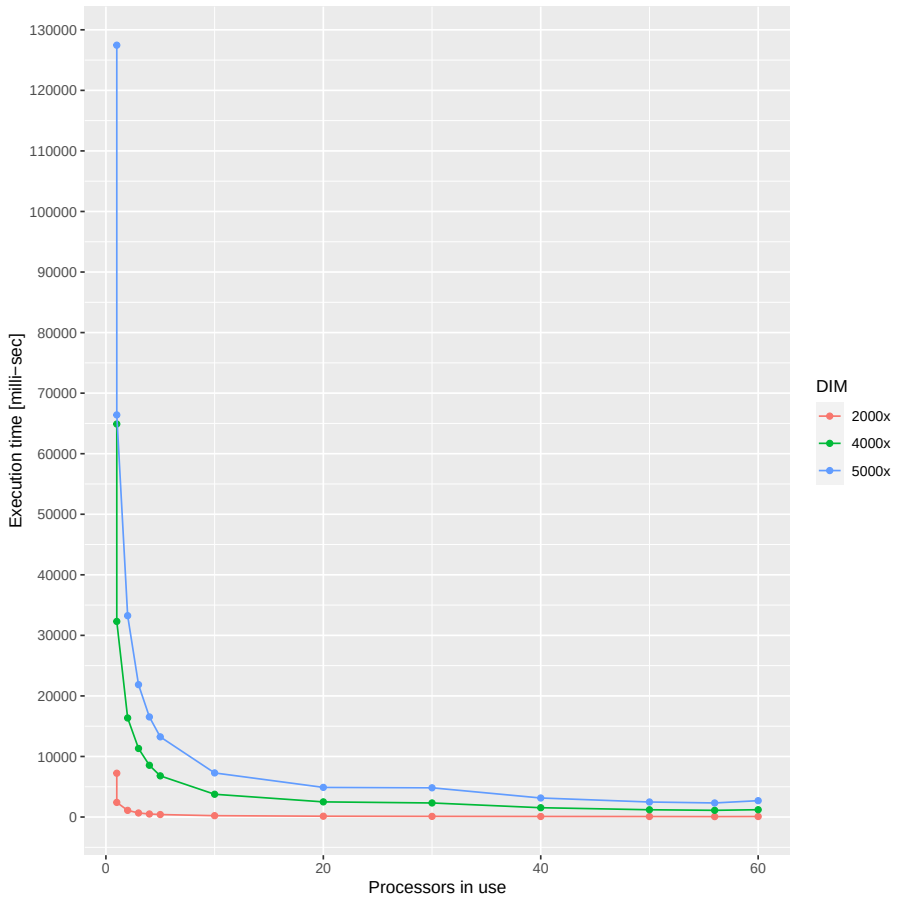

Figure 3.29: Execution times multiplication of double precision square matrices using Intrinsics and OpenCilk

3.5.3 CUDA

From Table 3.15 it is known CUDA can process double precision floating point values. Here CUDA code to perform matrix multiplication of double precision floating point values was executed on a MSI manufactured GeForce RTX 2070 Super graphics card. The card was mounted on a MacPro 2019. From the execution time measured the speed up was computed with respect to serial code for performing the same operation on the MacPro 2019 alone using no Intrinsics.

The CUDA code used here implements the algorithm proposed by **kreutz:2017** for mapping matrix multiplication onto CUDA enabled hardware. The mapping uses a two dimensional arrangement for both grid of CUDA blocks and blocks of threads positioned into that grid. The matrices being multiplied are assumed to be square with dimension $n \times n$. All execution is performed in thread blocks which this algorithm assigns to have dimension $m \times m$. The value of m is available for selection. Having the threads containing all the execution is in line with the CUDA architecture where threads perform the computation.

Each row by column multiplication of the two matrices to produce each value in the result matrix involves n multiplications and additions. These are performed in one or more thread blocks. These $m \times m$ thread blocks are arranged with the grid block configured into a square array. The size of this grid block array is a dimension k where:

$$\begin{array}{ll} k = n/m & \text{if } n \text{ is divisible by } m \\ k = n/m + 1 & \text{if } n \text{ is not divisible by } m \end{array}$$

The $k \times k$ grid array containing $k \times k$ thread blocks of size $m \times m$ calculate the n elements of a row of the result array. So this combination is repeated n times as a CUDA procedure.

Figure 3.30 lists a program which implemented this algorithm together with points for obtaining execution timing of it's operation. Also note in this program, matrix `B` is stored as a transpose with it's columns orientated as rows, in the same manner as matrix `A`. This program needed to be mapped to the hardware upon which it was to operate. Each CUDA enabled GPU card is different. However the values of interest in this adaption process can be obtained by software from the GPU card itself. Of the range of values made available for the MSI GeForce RTX 2070 Super card by this process, the values:

warp size	32
maximum threads per block	1024
maximum grid dimensions	2147483647, 65535, 65535
maximum block dimensions	1024, 1024, 64

were the most significant. The maximum number of threads per block limits the choice of m. The thread block was of dimension $m \times m$, so the maximum value m could take for this card was 32 ($32 \times 32 = 1024$). For this algorithm, this limit on the size of the threads in a thread block over rides the maximum block dimension available on this card. The maximum grid dimension imposed no limitation on the k values following from the $n \times n$ dimension of the matrices used here. The two matrices multiplied were filled with random positive and negative double precision floating point values.

The CUDA code of Figure 3.30 and the serial code for comparison were compiled using the NVIDIA `nvcc` compiler. The command lines used to compile and run the resulting executable was of the form:

```
nvc –O3 cfmatrixrr.cu –o cfmatrixrr
./cfmatrixrr
```

The `nvcc` compiler used the filename extension `.cu` to identify CUDA code being compiled nad `.c` for C code.

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

#define DIM 4000

__global__ void mm_kernel(double *A, double *B, double *C, int n)
{
  int rowA = blockIdx.x*blockDim.x + threadIdx.x;
  int rowB = blockIdx.y*blockDim.y + threadIdx.y;
  int i;

  if (rowA < n && rowB < n)
    for (i = 0; i < n; ++i)
      C[rowA*n + rowB] += A[rowA*n + i]*B[rowB*n + i];
}

double a[DIM][DIM], b[DIM][DIM], c[DIM][DIM];

int main() {
  double *dev_a, *dev_b, *dev_c;
  int i, j;
  double ticks;
  double my_get_wtime(void);

  dim3 blockDim(6,6);
  dim3 gridDim(667,667);

  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      c[i][j] = 0.0;

  srand(time(NULL));
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++) {
      a[i][j] = rand() % 10000;
      a[i][j] = a[i][j] - 5000.0;
      b[j][i] = rand() % 10000;
      b[j][i] = b[j][i] - 5000.0;
    }
  ticks = my_get_wtime();  /* measure the time */

  cudaMalloc((void **)&dev_a, DIM*DIM*sizeof(double));
  cudaMalloc((void **)&dev_b, DIM*DIM*sizeof(double));
  cudaMalloc((void **)&dev_c, DIM*DIM*sizeof(double));

  cudaMemcpy(dev_a, a, DIM*DIM*sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, b, DIM*DIM*sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(dev_c, c, DIM*DIM*sizeof(double), cudaMemcpyHostToDevice);

  mm_kernel<<<gridDim, blockDim>>>(dev_a, dev_b, dev_c, DIM);

  cudaMemcpy(c, dev_c, DIM*DIM*sizeof(double), cudaMemcpyDeviceToHost);

  cudaFree(dev_a);
  cudaFree(dev_b);
  cudaFree(dev_c);

  ticks = my_get_wtime() - ticks;  /* compute running time */
  printf("DIM = %d    Elapse time: %lf milli-sec\n", DIM, 1000.0*ticks);
  return 0;
}
```

Figure 3.30: CUDA program to measure executing two square double precision matrix multiplication

Table 3.18: Execution times for CUDA multiplication of double precision square matrices

DIM	Program	grid	block	worked	Execution time [milli-sec] min	mean	max	Speed up
100	fmatrixrr.c				2.63	2.66	2.79	
	cfmatrixrr.cu	100	1	yes	117.6	134.7	156.7	
	cfmatrixrr.cu	50	2	yes	114.6	136.5	155.2	
	cfmatrixrr.cu	25	4	yes	125.2	136.6	155.1	
	cfmatrixcc.cu	20	5	yes	119.2	137.0	162.8	
	cfmatrixrr.cu	17	6	yes	114.4	133.6	147.0	
	cfmatrixrr.cu	10	10	yes	119.0	135.8	151.2	
	cfmatrixrr.cu	5	20	yes	121.1	138.2	150.5	
	cfmatrixrr.cu	4	30	yes	120.0	134.4	145.5	
	cfmatrixrr.cu	4	31	yes	116.3	133.0	153.6	
	cfmatrixrr.cu	4	32	yes	117.9	132.5	151.3	
	cfmatrixrr.cu	4	33	no	119.1	133.7	153.0	
	cfmatrixrr.cu	3	40	no	122.0	136.6	145.3	
500	fmatrixrr.c				115.8	118.1	121.5	
	cfmatrixrr.cu	500	1	yes	159.4	175.2	193.8	
	cfmatrixrr.cu	250	2	yes	134.8	148.0	161.2	
	cfmatrixrr.cu	125	4	yes	127.9	138.3	146.9	
	cfmatrixrr.cu	100	5	yes	129.8	141.8	159.4	
	cfmatrixrr.cu	84	6	yes	129.2	141.1	158.2	
	cfmatrixrr.cu	50	10	yes	121.2	139.5	150.0	
	cfmatrixrr.cu	25	20	yes	138.4	145.4	154.0	
	cfmatrixrr.cu	17	30	yes	124.5	147.1	158.0	
	cfmatrixrr.cu	17	31	yes	124.8	146.3	165.1	
	cfmatrixrr.cu	16	32	yes	130.5	149.4	162.0	
	cfmatrixrr.cu	16	33	no	124.4	134.5	143.3	
	cfmatrixrr.cu	13	40	no	113.0	134.6	149.5	
1000	fmatrixrr.c				878.6	882.5	902.1	
	cfmatrixrr.cu	1000	1	yes	370.7	385.8	400.5	2.3
	cfmatrixrr.cu	500	2	yes	175.2	202.2	219.3	4.4
	cfmatrixrr.cu	250	4	yes	140.0	152.1	165.5	5.8
	cfmatrixrr.cu	200	5	yes	147.5	173.7	432.7	5.1
	cfmatrixrr.cu	167	6	yes	152.3	164.2	182.8	5.4
	cfmatrixrr.cu	100	10	yes	153.9	167.6	178.2	5.3
	cfmatrixrr.cu	50	20	yes	173.1	185.4	213.0	4.8
	cfmatrixrr.cu	34	30	yes	189.8	196.9	211.8	4.5
	cfmatrixrr.cu	33	31	yes	188.7	201.8	216.9	4.4
	cfmatrixrr.cu	32	32	yes	181.6	198.5	214.2	4.5
	cfmatrixrr.cu	31	33	no	118.8	128.8	150.8	
	cfmatrixrr.cu	25	40	no	117.8	126.8	135.3	
2000	fmatrixrr.c				7167.4	7190.1	7254.8	
	cfmatrixrr.cu	2000	1	yes	1708.8	1940.8	2198.3	3.7
	cfmatrixrr.cu	1000	2	yes	614.4	634.1	652.6	11.3
	cfmatrixrr.cu	500	4	yes	331.1	343.4	360.5	20.9
	cfmatrixrr.cu	400	5	yes	391.4	406.5	420.9	17.7
	cfmatrixrr.cu	334	6	yes	400.3	418.6	437.5	17.2
	cfmatrixrr.cu	200	10	yes	461.7	481.4	506.6	14.9
	cfmatrixrr.cu	100	20	yes	592.3	608.2	629.7	11.8
	cfmatrixrr.cu	67	30	yes	637.3	654.8	671.3	11.0
	cfmatrixrr.cu	65	31	yes	651.4	668.7	691.2	10.8
	cfmatrixrr.cu	63	32	yes	652.6	670.7	690.5	10.7
	cfmatrixrr.cu	61	33	no	133.2	140.8	151.3	
	cfmatrixrr.cu	50	40	no	134.0	141.6	152.5	

Continued on next page

DIM	Program	grid	block	worked	Execution time [milli-sec] min	mean	max	Speed up
4000	fmatrixrr.c				61582.6	64478.4	64923.2	
	cfmatrixrr.cu	4000	1	yes	14227.5	14486.6	14589.6	4.5
	cfmatrixrr.cu	2000	2	yes	3961.1	3979.5	3997.6	16.2
	cfmatrixrr.cu	1000	4	yes	1807.6	1822.7	1842.7	35.4
	cfmatrixrr.cu	800	5	yes	2018.5	2046.4	2065.7	31.5
	cfmatrixrr.cu	667	6	yes	2154.4	2179.2	2196.9	29.6
	cfmatrixrr.cu	400	10	yes	2612.5	2642.9	2667.6	24.4
	cfmatrixrr.cu	200	20	yes	3633.9	3671.7	3699.6	17.6
	cfmatrixrr.cu	134	30	yes	4169.2	4213.0	4238.1	15.3
	cfmatrixrr.cu	130	31	yes	4372.3	4415.2	4449.4	14.6
	cfmatrixrr.cu	125	32	yes	4240.2	4262.5	4282.6	15.1
	cfmatrixrr.cu	122	33	no	188.4	200.0	207.4	
	cfmatrixrr.cu	100	40	no	191.6	201.0	218.6	
5000	fmatrixrr.c				120701.1	126075.0	127621.7	
	cfmatrixrr.cu	5000	1	yes	27648.3	28206.9	28303.9	4.5
	cfmatrixrr.cu	2500	2	yes	7536.6	7617.0	7677.6	16.6
	cfmatrixrr.cu	1250	4	yes	3342.5	3360.7	3378.0	37.5
	cfmatrixrr.cu	1000	5	yes	3079.5	3105.5	3137.0	40.6
	cfmatrixrr.cu	834	6	yes	3494.8	3538.9	3562.9	35.6
	cfmatrixrr.cu	500	10	yes	3812.5	3849.1	3875.3	32.8
	cfmatrixrr.cu	250	20	yes	5780.2	5825.2	5855.8	21.6
	cfmatrixrr.cu	167	30	yes	7750.7	7829.6	7871.2	16.1
	cfmatrixrr.cu	162	31	yes	8253.6	8346.3	8377.4	15.1
	cfmatrixrr.cu	157	32	yes	7894.6	7955.7	8005.8	15.9
	cfmatrixrr.cu	152	33	no	230.0	242.8	252.0	
	cfmatrixrr.cu	125	40	no	229.9	245.1	253.7	

Table 3.19: Execution times for CUDA multiplication of single precision square matrices

DIM	Program	grid	block	worked	Execution time [milli-sec] min	mean	max	Speed up single	double
100	fmatrixrr.c				2.75	2.19	2.68		
	cfmatrixrr.cu	100	1	yes	123.1	137.1	151.9		
	cfmatrixrr.cu	20	5	yes	117.0	135.3	144.9		
	cfmatrixrr.cu	17	6	yes	124.2	136.7	161.1		
	cfmatrixrr.cu	5	20	yes	125.6	133.4	146.6		
	cfmatrixrr.cu	4	32	yes	122.6	135.9	159.6		
	cfmatrixrr.cu	3	40	no	123.8	134.1	142.5		
500	fmatrixrr.c				114.8	119.0	139.3		
	cfmatrixrr.cu	500	1	yes	116.4	136.8	149.1		
	cfmatrixrr.cu	100	5	yes	121.4	134.8	142.0		
	cfmatrixrr.cu	84	6	yes	118.0	135.6	154.1		
	cfmatrixrr.cu	25	20	yes	117.4	137.1	168.3		
	cfmatrixrr.cu	16	32	yes	123.1	137.4	153.4		
	cfmatrixrr.cu	13	40	no	123.5	135.6	146.8		
2000	fmatrixrr.c				7135.4	7139.6	7145.8		
	cfmatrixrr.cu	2000	1	yes	923.0	941.9	955.6	7.6	3.7
	cfmatrixrr.cu	400	5	yes	271.3	281.6	290.4	25.4	17.7
	cfmatrixrr.cu	334	6	yes	283.6	308.7	323.7	23.1	17.2
	cfmatrixrr.cu	100	20	yes	393.5	414.3	448.3	17.2	11.8
	cfmatrixrr.cu	63	32	yes	591.8	609.9	637.2	11.7	10.7
	cfmatrixrr.cu	50	40	no	124.2	132.0	142.2		
5000	fmatrixrr.c				115195.1	117286.7	118421.9		
	cfmatrixrr.cu	5000	1	yes	12338.4	12546.8	12657.2	9.4	4.5
	cfmatrixrr.cu	1000	5	yes	2159.0	2181.7	2208.1	53.8	40.6
	cfmatrixrr.cu	834	6	yes	2612.9	2643.3	2658.3	44.4	35.6
	cfmatrixrr.cu	250	20	yes	4294.3	4329.2	4360.6	27.1	21.6
	cfmatrixrr.cu	157	32	yes	7036.7	7126.5	7241.1	16.5	15.9
	cfmatrixrr.cu	125	40	no	172.5	183.7	195.3		

Table 3.18 shows the execution times for running the program listed in Figure 3.30 using different block sizes (m value) and the resulting grid size (k value) computed for different size (n value) matrices. The minimum, mean, and maximum execution time is recorded for 25 executions of the program with the given settings. Under the `Program` heading, `fmatrixrr.c` indicates the serial program executed on the MacPro 2019, while `cfmatrixrr.cu` indicates execution of the CUDA program corresponding to Figure 3.30.

The results presented in Table 3.18 are divided into groups with each group pertaining to a single size of the two matrices multiplied. Of interest was the speed up observed for each combination of the variables. Within each group, the speed up was calculated by dividing the mean execution time of the serial execution by the mean execution time of the CUDA code of each set of parameters. For dimensions 100 and 500, the CUDA mean execution times were greater than the serial execution time. So speed up was meaningless. In all groups selecting a block (m) parameter greater than 32 resulted in a result matrix containing only zero values. This is denoted in Table 3.18 by `no` under the column headed by `worker`. Again, speed up was meaningless in such cases.

The results in Table 3.18 show the behaviour of the matrix multiplication algorithm with different parameter settings. With smaller matrix sizes, the execution time saving using the GPU card was masked by the overhead. The overhead of allocating memory on the GPU, transferring data to the GPU, initialing execution on the GPU, transferring the results back to the CPU, and removing the allocated memory on the GPU had to be considered. When the size of the thread block was larger than the allowed number of threads, those overheads again dominated the execution times. Only with the larger matrix dimensions (1000, 2000, 4000, and 5000) did the execution advantage of the GPU overcome those overheads. The behaviour of the GPU as a computing resource followed from the algorithm's behaviour.

Values of `Speed up` in Table 3.18 increase with increasing matrix dimension. It is worth reiterating that these are speed up values for double precision operations. NVIDIA GPUs generally are proposed as single precision devices. A selection of the algorithm parameters used to generate the data of Table 3.18 were used to perform single precision equivalent calculations. The results obtained are in Table 3.19. For comparison, the corresponding double precision speed up from Table 3.18 have been transcribed. The single precision speed ups are greater than the corresponding double precision values. The overhead masking executable time savings and exhaustion behaviour which occurred with double precision also occur with single precision.

The following table summarizes the largest speed up results for double precision floating point processing obtained using CUDA relative to those using the OpenCilk and Intel Intrinsics combination.

	OpenCilk		
DIM	Speed up	Processors	CUDA
100	18.4	20	
500	93.6	56	
1000	66.9	56	5.8
2000	103.9	56	20.8
4000	58.7	56	35.4
5000	54.9	56	40.6

Those results are itemized by the size of the computation performed as indicated by the dimension (DIM) of the two matrices being multiplied. The blank entries under `CUDA` indicate no speed up was measured. Both approaches used multiple computing elements and SIMD (Single Instruction Multiple Data).