Tutorial for using *pocl* on CPU multi-cores

Ross Maloney

August 2, 2023

Portable OpenCL, or *pocl*, is an implementation of OpenCL, or Open Computer Language. Although *pocl* is designed to be easily ported to different target processors, the distributed implementation uses CPUs as it's target. Specifically such CPUs are assumed to be multi-core processors. As such if differs from a large number of OpenCL distributions which are targeted for GPUs.

Multi-core CPUs are different to processors generally encountered with OpenCL. Generally there are fewer compute units. In this context a compute unit corresponds to one of the multi-cores. One OpenCL kernel executes on multiple multi-cores. In contrast to processors generally considered in OpenCL, these multi-cores are more capable processors and thus able to handle more complex kernels. The art of using OpenCL on multi-cores, and thus using pocl, is balancing the reduction in processing elements available by increasing the complexity of the program (kernel) which can execute on such elements.

This tutorial is based on using an Apple Mac Pro 2019 with 28 cores operating under Debian Linux.

1 Installing the *pocl* system on Debian

Assume a standard Debian Linux system has been installed. In addition, a *clang* distribution had to be installed into it. The *clang* package was installed from a Debian archive using `aptitude`. This installed `clang` version 14. Installing `clang` on the system resulted in `clang` able to compile and produce an executable C program, but not an *OpenCL* program.

Then `aptitude` was used to install `pocl-opencl-icd`. This brought in a number of other elements. In `/etc/OpenCL/vendors` (which this installation created) there was a file `pocl.icd` which contained the filename `libpocl.so.2.10.0`. An attempt to install `libpocl-dev` using `aptitude` gave the information that `ocl-icd-opencl-dev` was required instead. It was installed using `aptitude`. This installed the necessary header files and libraries for OpenCL and also for *pocl*.

To use *pocl*, the parameter `-lOpenCL` was added to the standard `clang` command line to link with the *pocl* library.

2 Applying OpenCL to multi-cores

OpenCL is an open source follow on from the proprietary, although freely distributed, CUDA. Both can do the same processing but with different nomenclature, with OpenCL supporting the CPU multi-core environment. Because OpenCL followed CUBA in appearance historically, the CUDA approach is widespread. Table 1 attempts to link those approaches.

Table 1: Correspondence between OpenCL and CUDA parameters

Functional role	OpenCL	CUDA
Major division	work group	block
Minor division	work item	thread
L1 memory	global memory	global memory
L2 memory	local memory	shared memory
L3 memory	private memory	local memory
Read only memory	constant memory	constant memory
Picture frame storage	image	texture
Device function		_device
Constant memory	__constant	__constant
Device variable	__global	__device
Shared memory	__local	__shared
Number of blocks	get_num_groups	gridDim
Size of block	get_local_size()	blockDim
Block index	get_group_id()	blockIdx
Thread index in block	get_local_id()	threadIdx
Kernel identifier	__kernel	__global__
Kernel launch	dimemsion of indexing	number of blocks
	dimension's thread count	threads per block
	dimension's threads per block	

A important difference between OpenCL and CUDA occurs in the parameters used to launch their processing kernel. Although processing in both OpenCL and CUDA is primarily directed at blocks with threads following, OpenCL does not give a block count (work group count) in the launch. Blocks needed in OpenCL are calculated from the thread counts supplied.

Each core in a multi-core OpenCL application is a separate compute unit. Generally such cores are fewer in number than the compute units of a GPU. Thus a modification in approach to kernel programming is warranted. Figure 1 gives the general idea of the differences and similarities between using those two environments. In the multi-core environment `pocl` maps individual compute units, identified as work-groups, to cores in a one-to-one fashion.

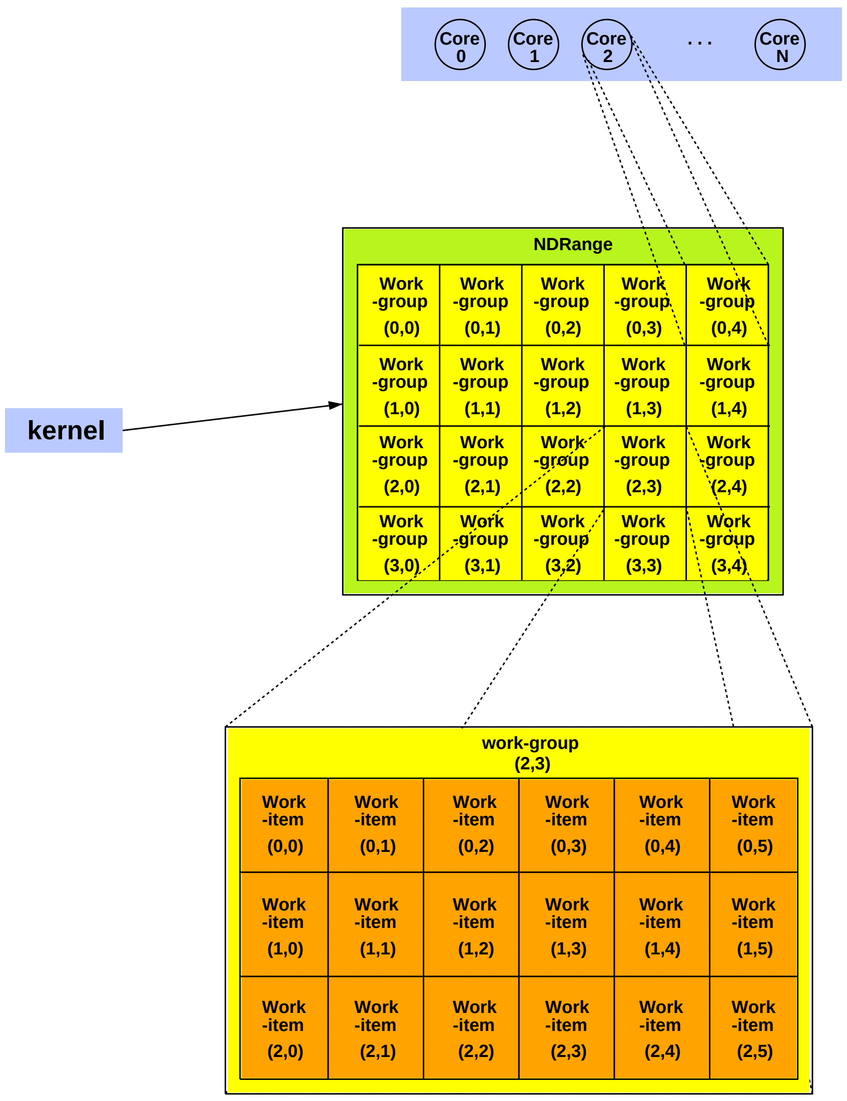

Figure 1: Kernel composition to processor's cores

Familiarity with both OpenCL and CUDA, and programming using the C language in each is assumed here. An overall summary of applying OpenCL to multicore CPUs is:

- Each multi-core is an OpenCL compute unit
- Each compute unit executes one work-group
- Synchronisation of work-groups cannot occur
- The kernel can be more complex than those used with a GPU
- Kernels can access the L1, L2, and L3 memory of it's execution multi-core
- Only work-items information is passed to the `clEnqueueNDRangeKernel()` call which launches the kernel.

A further assistance is provided in Table 2 with respect to writing code for navigation within a kernel. The parallel executions of the one kernel code are independent of one another. Thus each kernel must determine it's overall part in the total computation by referring back to the OpenCL system by using OCL library calls. In Table 2 the variable used in the following programs for each of such calls is tabulated. In their use, each is prefixed by the direction (eg. x) to denote the direction to which they apply.

Table 2: Useful kernel reference calls

OCL library call	host	variable	value obtained
get_global_size()	global[]	gs	ND range length in work-items
get_global_id()		hi	highest global work-item in group
get_local_size()	local[]	ls	work-items per work-group
get_local_id()		li	local highest work-item in group
get_num_groups()		ng	number of work-groups
get_group_id()		gi	group number

3 Addition of columns of data

Figure 2 shows an OpenCL program which adds 7 columns of data with each column containing 15 integer values. Each column addition is performed by a separate work-group. Each of those work-groups execute on a separate CPU multi-core using a copy of the OpenCL kernel. The arguments to the kernel are a matrix containing the columns of data, the length of the data column, and a vector to contain the results. Each work-group is to get a copy of the data array but only return a single integer containing the sum of the data column of that array it summed. The sum produced by each work-group is returned as a component of the vector which contains the overall 7 sums required.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <CL/cl.h>

#define SIZE 7
#define LENGTH 15

int  A[SIZE][LENGTH], C[SIZE];

const char *programSource =
"__kernel \n"
"void add(__global int *A,  \n"
"         int N,            \n"
"         __global int *C)  \n"
"{                          \n"
"  int  xhi, j;             \n"
"                           \n"
"  xhi = get_global_id(0);  \n"
"  C[xhi] = 0;              \n"
"  for (j = 0; j < N; j++)  \n"
"    C[xhi] += A[N*xhi + j];\n"
"} \n";

int main()
{
  int  i, j;
  int  markerA;
//  long  ticks, my_get_wtime(void);
//  float marker;
  cl_platform_id    platform;
  cl_device_id      device;
  cl_int            ret;
  cl_uint           ret_num_devices, ret_num_platforms;
  cl_context        context;
  cl_mem            a_mem_obj, b_mem_obj, c_mem_obj;
  cl_command_queue  command;
  cl_program        program;
  cl_kernel         kernel;
  size_t            global[3];
  long              dataSizeA, dataSizeC;
  size_t            size_ret;

/* prepare data */
  markerA = 1;
  for ( i = 0; i < SIZE; i++)
    for (j = 0; j < LENGTH; j++)  {
      markerA++;
      A[i][j] = markerA;
    }
```

Figure 2: Program to calculate 7 data vectors in parallel (Continues ...)

```
    dataSizeA = SIZE * LENGTH * sizeof(int);
    dataSizeC = SIZE * sizeof(int);

//  ticks = my_get_wtime();

/* setup CPU */
    ret = clGetPlatformIDs(1, &platform, &ret_num_platforms);
    ret = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &device,
                         &ret_num_devices);

    context = clCreateContext(NULL, 1, &device, NULL, NULL, &ret);
    command = clCreateCommandQueueWithProperties(context, device, 0, &ret);

    a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, dataSizeA,
                               NULL, &ret);
    c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY, dataSizeC,
                               NULL, &ret);

    ret = clEnqueueWriteBuffer(command, a_mem_obj, CL_TRUE, 0, dataSizeA,
                               A, 0, NULL, NULL);

    program = clCreateProgramWithSource(context, 1,
                         (const char **)&programSource, NULL, &ret);
    ret = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
    kernel = clCreateKernel(program, "add", &ret);

    i = LENGTH;
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_mem_obj);
    ret = clSetKernelArg(kernel, 1, sizeof(int), &i);
    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_mem_obj);

    global[0] = SIZE;
    ret = clEnqueueNDRangeKernel(command, kernel, 1, NULL, global,
                                 NULL, 0, NULL, NULL);

    ret = clEnqueueReadBuffer(command, c_mem_obj, CL_TRUE, 0, dataSizeC, C,
                              0, NULL, NULL);

    ret = clFlush(command);
    ret = clFinish(command);
    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseMemObject(a_mem_obj);
    ret = clReleaseMemObject(c_mem_obj);
    ret = clReleaseCommandQueue(command);
    ret = clReleaseContext(context);

//  ticks = my_get_wtime() - ticks;
//  marker = ticks;
//  printf("Elapse time: %.1f milli-sec\n", marker/1000000.0);
```

Figure 2: Program to calculate 7 data vectors in parallel (Continues . . .)

```
  for (j = 0; j < LENGTH; j++)  {
    for (i = 0; i < SIZE; i++)   printf("%6d ", A[i][j]);
    printf("\n");
  }
  printf("\n");
  for (i = 0; i < SIZE; i++)  printf("%6d ", C[i]);
  printf("\n");

  return(0);
}
```

Figure 2: Program to calculate 7 data vectors in parallel

This program was written in C. C stores a two dimensional array in row order. The program is posed as a one dimensional problem from the standpoint of the host part of the total program. The statements which are commented out perform execution timing which is discussed in the following Section.

The OpenCL technique of placing both the host and kernel program parts in the same file was used here. This enabled one application of `clang` to both compile and link the program. Successfully having done that, there still remained the possibility of a kernel error when processed by the `clBuildProgram()` call. Since the kernel code is contained within a string, `clang` does not look at the contents of that siring. Figure 2 lists the program. Notice the kernel is contained in a multi-line string. The length of the data vector is given by the constant `LENGTH` and the number of data columns, and also the number of work-groups to be used is given by the constant `SIZE`.

```
     2     17     32     47     62     77     92
     3     18     33     48     63     78     93
     4     19     34     49     64     79     94
     5     20     35     50     65     80     95
     6     21     36     51     66     81     96
     7     22     37     52     67     82     97
     8     23     38     53     68     83     98
     9     24     39     54     69     84     99
    10     25     40     55     70     85    100
    11     26     41     56     71     86    101
    12     27     42     57     72     87    102
    13     28     43     58     73     88    103
    14     29     44     59     74     89    104
    15     30     45     60     75     90    105
    16     31     46     61     76     91    106

   135    360    585    810   1035   1260   1485
```

Figure 3: Output produced by the program of Figure 2

Figure 3 shows the Output produced by running the program of Figure 2.

Notice in the listing of Figure 2 a number of lines are commented out. Those lines were associated with timing the execution of the program. Their use is described below.

4 Alternate approaches to calculation by a kernel

There are four manners to approaching coding of a kernel for multi-cores. Each uses memory of the core differently. The kernel used in Figure 2 is one approach. It uses global memory only. This is the memory which links the kernel to the host; the memory addressed by the `clCreateBuffer()` and `clEnqueueReadBuffer()` OpenCL calls of the host program. It is the slowest execution memory. Getting data moving between the host and kernel requires using this memory. Three alternatives are considered here which involve using private memory available on the multi-cores.

Private memory on a multi-core is a cache memory. A multi-core has three cache memories; L1, L2, and L3. The data shown in Table 3 shows the size of such cache, where their speed ranges from L1 to L3, in descending order. The amount of each type of cache memory also varies. Each multi-core has its own L1 and L2 cache, but L3 is shared by all the multi-cores present. The L1 cache is the best to exploit. It being the fastest but it is only small in size. Putting those cache sizes into different terms, the size of 896 KiB means capable of holding 224,000 32-bit values, or 112,000 64-bit values in total. Cache sizes for the Mac Pro 2019 are large compared with multi-cores generally found currently, as typified by the Mac Pro 2013. Here the Mac Pro 2019 was used.

Table 3: Apple Intel Mac core data shown by `lscpu`

	Mac Pro 2019	Mac Pro 2013
CPU model	Intel Xeon W-3275M	Intel Xeon E5-1650
cores	28	6
threads per core	2	2
cpu GHz	2.5	3.5
L3 cache MiB	38.5 (1 instance)	12 (1 instance)
L2 cache MiB	28 (28 instances)	1.5 (6 instances)
L1 cache KiB	896 (28 instances)	192 (6 instances)

Figure 4 shows three kernels each as a replacement to that used in Figure 2. Nothing else was changed. All the variables defined in the kernel code are stored in the private (L1 cache) of each multi-core which executes this kernel code. This is the

```
const char *programSource =
"__kernel \n"                               kernel1
"void add(__global int *A, \n"
"         int  N,          \n"
"         __global int *C) \n"
"{                         \n"
"  int  xhi, j;            \n"
"  int  tmp;               \n"
"                          \n"
"  xhi = get_global_id(0); \n"
"  tmp = 0;                \n"
"  for (j = 0; j < N; j++) \n"
"    tmp += A[N*xhi + j];  \n"
"  C[xhi] = tmp;           \n"
"                          \n"
"} \n";
```

```
const char *programSource =
"__kernel \n"                               kernel2
"void add(__global int *A, \n"
"         int  N,          \n"
"         __global int *C) \n"
"{                         \n"
"  int  xhi, j;            \n"
"  int  fastA[2000];       \n"
"                          \n"
"  xhi = get_global_id(0); \n"
"  C[xhi] = 0;             \n"
"  for (j = 0; j < N; j++) \n"
"    fastA[j] = A[N*i + j];\n"
"  for (j = 0; j < N; j++) \n"
"    C[xhi] += fastA[j];   \n"
"                          \n"
"} \n";
```

```
const char *programSource =
"__kernel \n"                               kernel3
"void add(__global int *A, \n"
"         int  N,          \n"
"         __global int *C) \n"
"{                         \n"
"  int  xhi, j;            \n"
"  int  fastA[2000];       \n"
"  int  tmp;               \n"
"                          \n"
"  xhi = get_global_id(0); \n"
"  for (j = 0; j < N; j++) \n"
"    fastA[j] = A[N*i + j];\n"
"  tmp = 0;                \n"
"  for (j = 0; j < N; j++) \n"
"    tmp += fastA[j];      \n"
"  C[xhi] = tmp;           \n"
"                          \n"
"} \n";
```

Figure 4: Kernels 1, 2, and 3 for summing 7 data columns

important point. The 32-bit storage for array `fastA[]` together with the four other variables are within the 224,000 32-bit limit of the L1 cache available on each core of the Mac Pro used here.

Using the kernels from Figure 4 in the program of Figures 2 produced the same computational sums. Each program was compiled using `clang` with the `-O3` optimization flag and linked with the `OpenCL` library. To compare the behaviour of these programs the statements commented out in Figure 2 were used to generate execution times. Running each such timed execution 25 times, the minimum, maximum, and mean of those 25 repeats are tabulated in Table 4. The time values shown have units of milli-seconds.

Table 4: Program execution times for kernels calculating 7 columns of values

	length	min	mean	max
nullkernel		158.2	166.7	188.7
onerow	15	155.7	163.8	179.0
kernel1	15	157.1	167.2	182.2
kernel2	15	156.8	163.6	173.7
kernel3	15	149.8	167.0	194.1
twoD	15	151.8	166.6	181.1
onerow	1500	156.7	167.7	190.0
kernel1	1500	156.7	168.1	182.8
kernel2	1500	156.7	171.6	195.1
kernel3	1500	157.9	171.5	190.5
twoD	1500	155.3	171.2	183.4

Added to the results in Table 4 is the execution time of a `nullkernel` in which the kernel contained only declaration of variables but no other computation. All the four computational kernels were run using a data length of 15 and 1500. The differences in the mean execution times appear insignificant. This suggests the majority of the execution time was consumed in the host program; the amount of computation was not sufficient to offset this setting up overhead. To reinforce this point a program to produce the same computational results was written in standard C. It gave executions times of 0.001, 0.080, 0.265, and 2.662 milli-sec for data column lengths of 15, 1500, 5000, and 50000 values, respectively. The `twoD` execution times are those produced by the program of Figure 5.

The kernel used in the program of Figure 2 and the three in Figure 4 are called by the same host program. Each produces the same value of each of the seven summations. They differ in the way memory is used. The kernel of Figure 2 uses only the host memory passed as arrays `A` ans `C`. The `kernel1` of Figure 4 uses memory local of each kernel to hold variable `tmp` into which the individual sum of data obtained from the `A` array. As in each of these approaches, each kernel produces a sum of the data associated with the assigned kernel. In the `kernel1` case, the calculated sum in `tmp` is assigned to appropriated indexed position in the array `C` for return to the host. In the `kernel2` approach the data is extracted from the `A` and stored in the array `fastA` which is in the local memory of each kernel. In the `kernel3` the local memories of `tmp` and `fastA` are used to calculate the sum in the kernel. Because each kernel is executed in a separate CPU core, the local memory is local to the core the individual CPU core used by the kernel.

The programming principle used here is memory declared in the body of a kernel is local to the compute device, in this case the CPU core, which executes the kernel.

5 Two dimensional approach to vector summing

Having one work-group for each of the vectors is a one dimensional approach. In the program of Figure 2 this was produced by the 1 in the argument list of the `clEnqueueNDRangeKernel()` call and the single dimension of the `global[]` array which indicated the number of work-item. In that case each work-item was contained in it's own work-group. The work-groups were than launched to execute in parallel. In a two dimensional approach to this summation problem, work-groups are assigned to each vector as in the one dimensional approach. Now the data for each vector are contained in other work-groups. These data work-groups are the second dimension.

The program in Figure 5 is a two dimensional approach to this summation example. In the `clEngueueNDRangeKernel()` call the dimension argument was set as 2. The `global[]` array then used two elements, one for the total number of work-items in the x-direction, and the other for the number in the y-direction. The array `local[]` contains the number of work-item per work-group in the corresponding x and y directions. An array containing the data for each of the work-groups is passed to the kernel together with the array for each work-group to return it's sum. Both those arrays are passed as one dimensional to the kernel. Each work-group kernel only uses the data which is assigned to it. Notice the kernel variables `ny, gx` and `nxg` are assigned a value in each work-group executing the kernel, corresponding to the reference calls listed in Table 2. These variables identify the work-group and thus how the work-group performs in the total computation.

In the case of Figure 5 each work-group is mapped onto a separate core of the computer by the OpenCL (pocl) system. The number of work-groups mapped in the x and y directions to cores is obtained by OpenCL (pocl) system by dividing the work-item entries in the `global[]` array by the corresponding entry in the `local[]` array. Notice in the program of Figure 2 there was no `local[]` array passed to the `clEnqueueNDRangeKernel()`. In that one dimensional case one work-item per work-group was assumed by the OpenCP (pocl) system.

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <time.h>
#include  <CL/cl.h>

#define  SIZE   7
#define  LENGTH   15

int    A[SIZE*LENGTH], C[SIZE];

const char *programSource =
"__kernel \n"
"void add2d(__global int *A, \n"
"           __global int *C) \n"
"{                            \n"
"  int  i, j;                 \n"
"  int  ygs, xgi, xmg;        \n"
"  int  tmp;                  \n"
"                             \n"
"  ygs = get_global_size(1);  \n"
"  xgi = get_group_id(0);     \n"
"  xng = get_num_groups(0);   \n"
"                             \n"
"  for (i = 0; i <= ygs; i++)  tmp += A[i*xng + xgi]; \n"
"  C[xgi] = tmp;              \n"
"} \n";

int main()

  int   i, j, counter;
  int   markerA;
  long   ticks;
  long   my_get_wtime(void);
  float marker;
  cl_platform_id    platform;
  cl_device_id      device;
  cl_int            ret;
  cl_uint           ret_num_devices, ret_num_platforms;
  cl_context        context;
  cl_mem            a_mem_obj, b_mem_obj, c_mem_obj;
  cl_command_queue  command;
  cl_program        program;
  cl_kernel         kernel;
  size_t            global[3], local[3];
  long              dataSizeA, dataSizeC;
  size_t            size_ret;

/* prepare data */
  markerA = 1;
  for ( i = 0; i < SIZE; i++)
    for (j = 0; j < LENGTH; j++)  {
      markerA++;
      A[j*SIZE + i] = markerA;
    }
```

Figure 5: Program to calculate 7 sums in a 2 dimensional ND Range (Continues . . .)

```
  dataSizeA = SIZE * LENGTH * sizeof(int);
  dataSizeC = SIZE * sizeof(int);

//  ticks = my_get_wtime();

/* setup CPU */
  ret = clGetPlatformIDs(1, &platform, &ret_num_platforms);
  ret = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &device,
                       &ret_num_devices);

  context = clCreateContext(NULL, 1, &device, NULL, NULL, &ret);
  command = clCreateCommandQueueWithProperties(context, device, 0, &ret);

  a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, dataSizeA,
                             NULL, &ret);
  c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY, dataSizeC,
                             NULL, &ret);

  ret = clEnqueueWriteBuffer(command, a_mem_obj, CL_TRUE, 0, dataSizeA, A,
                             0, NULL, NULL);

  program = clCreateProgramWithSource(context, 1,
                             (const char **)&programSource, NULL, &ret);
  ret = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
  kernel = clCreateKernel(program, "add2d", &ret);

  ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_mem_obj);
  ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), &c_mem_obj);

  global[0] = SIZE;  global[1] = LENGTH;
  local[0] = 1;  local[1] = 1;
  ret = clEnqueueNDRangeKernel(command, kernel, 2, NULL, global,
                               local, 0, NULL, NULL);

  ret = clEnqueueReadBuffer(command, c_mem_obj, CL_TRUE, 0, dataSizeC, C, 0,
                            NULL, NULL);

  ret = clFlush(command);
  ret = clFinish(command);
  ret = clReleaseKernel(kernel);
  ret = clReleaseProgram(program);
  ret = clReleaseMemObject(a_mem_obj);
  ret = clReleaseMemObject(c_mem_obj);
  ret = clReleaseCommandQueue(command);
  ret = clReleaseContext(context);

  ticks = my_get_wtime() - ticks;
  marker = ticks;
  printf("Elapse time: %.1f milli-sec\n", marker/1000000.0);
```

Figure 5: Program to calculate 7 sums in a 2 dimensional ND Range (Continues ...)

```
  counter = 0;
  for (j = 0; j < LENGTH; j++)  {
    for (i = 0; i < SIZE; i++)  {
      printf("%6d ", A[counter]);
      counter++;
    }
    printf("\n");
  }
  printf("\n");
  for (i = 0; i < SIZE; i++)  printf("%6d ", C[i]);
  printf("\n");

  return(0);
}
```

Figure 5: Program to calculate 7 sums in a 2 dimensional ND Range

The same output as in Figure 3 was produced. The timing of the execution of the program is included in Table 4 under the `twoD` tag.

6 Searching for kernel's position parameters

Some problems are multi-dimensional by their nature. Of those problems some can be decomposed into blocks, the blocks handled as separate problems, and the solution from each block reassembled into the solution to the original problem. OpenCL can assist in obtaining such block solutions. The important part in programming such block solutions is efficiently working with the relationship between the blocks.

OpenCL has this block handling ability extending into multi-dimensions. To make full use of OpenCL, the two (and three) dimensional property of the ND Range should be used. Each OpenCl cell represents a block which needs to know where in the overall structure it is placed, and thus contributing. The OpenCL system provides such information. This information is accessed by a kernel via OpenCL library functions calls, the more useful of which are shown in Table 2.

Given a two dimensional range a program to demonstrate the division of that range into blocks and each block having it's own assigned position is of interest. Such a program is listed in Figure 15. The range is 12 elements along the x axis, and 15 elements along the y axis. Each of thos elements corresponds to an OpenCl work-item. Further, the work-item along the x axis are gathered into clusters of 3 work-items each, and into clusters of 5 work-items along the y axis. Each of those clusters is an OpenCL work-group.

Before considering the output of Figure 15, it's kernel was changed to that of Figure 6. Everything else remained the same. The kernel receives the two dimensional array `A[][]` from the host program as a one dimensional array into which the kernel delivers it's results. In the Figure 6 kernel only the first four entries in this array are used.

```
const char *programSource =
"__kernel \n"
"void map(__global int *A) \n"
"{                         \n"
"  int  xgi, ygi;          \n"
"  int  xgs, ygs;          \n"
"  int  xli, yli;          \n"
"  int  xhi, yhi;          \n"
"                          \n"
"  xgi = get_group_id(0);  \n"
"  ygi = get_group_id(1);  \n"
"  xli = get_local_id(0);  \n"
"  yli = get_local_id(1);  \n"
"  xgs = get_global_size(0); \n"
"  ygs = get_global_size(1); \n"
"  xhi = get_global_id(0); \n"
"  yhi = get_global_id(1); \n"
"                          \n"
"  A[ygi] = ygs;           \n"
"} \n";
```

Figure 6: Kernel used to determine OpenCL worker parameters

Table 5: Kernel parameters extracted using Figure 6

Position	Parameter	Values				Unstable
A[xgi]	xgs	12	12	12	12	
A[xgi]	ygs	15	15	15	15	
A[xgi]	xgi	0	1	2	3	
A[xgi]	ygi	2	2	2	2	*
A[xgi]	xhi	2	5	8	11	
A[xgi]	yhi	14	14	14	14	*
A[xgi]	xli	2	2	2	2	
A[xgi]	yli	4	4	4	4	
A[ygi]	xgs	12	12	12		
A[ygi]	ygs	15	15	15		
A[ygi]	xgi	3	3	3		*
A[ygi]	ygi	0	1	2		
A[ygi]	xhi	11	11	11		*
A[ygi]	yhi	4	9	14		
A[ygi]	xli	2	2	2		
A[ygi]	yli	4	4	4		

The program of Figure 15 using the kernel of Figure 6 was compiled and executed multiple times. The index in array `A[]` was alternately set as `xgi` and `ygi` to get the numbering of the work-groups along the x and y directions, respectively. Each of the four library functions were assigned to that array element for both the x and y directions. Table 5 shows the results obtained. Each value on a line corresponds to a separate work-group.

Table 5 contains all the information needed to write a kernel for the example range. Table 5 is divided into an upper and lower half. The upper half has 4 values for each entry while the lower half has 3. This follows from the `xgi` and `ygi` argument of the `A[]` array used in the respective halves, where the OpenCL function `get_group_id()` obtained the values. The values 4 (12/3) and 3 (15/5) were supplied as corresponding entries in the arrays `global[]` and `local[]` in the host program. The `xhi` and `yhi` values in their corresponding `xgi` and `ygi` associations are the upper limit for indexing work-items in the x and y direction. The values of `xgs` and `ygs` were the size of the overall range (work-item count) in the x and y directions, respectively.

Note: The values returned for `ygi` and `yhi` in the `xgi` half of Table 5, and the `xgi` and `xhi` values in the `ygi` half of the table are indicated as *unstable*. This follows from repeated runs of the kernel displaying those values producing changing values. The `xli` and `yli` values were respectively the same for both `xgi` and `ygi` index of the `A[]` array.

Each of the 12 work-groups (4 * 3) has access to only one set of numbers from Table 5. It will know the total number of work-items along the x and y axes of the range from the `xgs` and `ygs` respective values. It will know it's x and y position in the range via the `xgi` and `ygi` respectively. it will know the upper limit in the x and y directions of the total work-items which it is to process via the `xhi` and `yhi` values. The `xli` and `yli` values will give the number of work-items below the `xhi` and `yhi` values which that work-group is to process. A numeric value of one (1) has to be added to the `xli` and `yli` values to include the `xhi` and `yhi` work-item to be processed by that work-group.

The problem with the values produced by this technique is their dependency on the `xgi` and `ygi` array index. These indices gave the appropriate 4 and 3 values respectively. But they also had an affect on the values produced; in four cases producing values which changed with subsequent runs of the kernel. Also, although the `xhi` and `yhi` values would be useful, their values were influence by the `xgi` and `ygi` used in their determination.

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>

#define SIZEX 12
#define SIZEY 15

int   A[SIZEY][SIZEX];

const char *programSource =
"__kernel \n"
"void map(__global int *A)   \n"
"{                            \n"
"  int  xgi, ygi, xgs, ygs;  \n"
"  int  xli, yli, xhi, yhi;  \n"
"                             \n"
"  xgi = get_group_id(0);    \n"
"  ygi = get_group_id(1);    \n"
"  xli = get_local_id(0);    \n"
"  yli = get_local_id(1);    \n"
"  xgs = get_global_size(0); \n"
"  ygs = get_global_size(1); \n"
"  xhi = get_global_id(0);   \n"
"  yhi = get_global_id(1);   \n"
"                             \n"

"  A[xgi] = xhi; \n"
"} \n";


int main()
{
  int  i, j;
  cl_platform_id    platform;
  cl_device_id      device;
  cl_int            ret;
  cl_uint           ret_num_devices, ret_num_platforms;
  cl_context        context;
  cl_mem            a_mem_obj, b_mem_obj, c_mem_obj;
  cl_command_queue  command;
  cl_program        program;
  cl_kernel         kernel;
  size_t            global[3];
  size_t            local[3];
  long              dataSizeA, dataSizeC;
  size_t            size_ret;

/* prepare data */

  dataSizeA = SIZEX * SIZEY * sizeof(int);
  for (i = 0; i < SIZEY; i++)
    for (j = 0; j < SIZEX; j++)
      A[i][j] = 0;
```

Figure 7: Program to produce an array of kernel parameters (Continues . . .)

```
/* setup CPU */
  ret = clGetPlatformIDs(1, &platform, &ret_num_platforms);
  ret = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &device,
                       &ret_num_devices);

  context = clCreateContext(NULL, 1, &device, NULL, NULL, &ret);
  command = clCreateCommandQueueWithProperties(context, device, 0, &ret);

  a_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY, dataSizeA, NULL,
                             &ret);

  ret = clEnqueueWriteBuffer(command, a_mem_obj, CL_TRUE, 0, dataSizeA,
                             A, 0, NULL, NULL);

  program = clCreateProgramWithSource(context, 1,
                             (const char **)&programSource, NULL, &ret);
  ret = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
  kernel = clCreateKernel(program, "map", &ret);

  ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_mem_obj);

  global[0] = SIZEX;  global[1] = SIZEY;;
  local[0] = 3;  local[1] = 5;
  ret = clEnqueueNDRangeKernel(command, kernel, 2, NULL, global,
                               local, 0, NULL, NULL);

  ret = clEnqueueReadBuffer(command, a_mem_obj, CL_TRUE, 0, dataSizeA,
                            A, 0, NULL, NULL);

  ret = clFlush(command);
  ret = clFinish(command);
  ret = clReleaseKernel(kernel);
  ret = clReleaseProgram(program);
  ret = clReleaseMemObject(a_mem_obj);
  ret = clReleaseMemObject(c_mem_obj);
  ret = clReleaseCommandQueue(command);
  ret = clReleaseContext(context);

  for (j = 0; j < SIZEY; j++)  {
    for (i = 0; i < SIZEX; i++)   printf("%4d ", A[j][i]);
    printf("\n");
  }

  return(0);
}
```

Figure 7: Program to produce an array of kernel parameters

7 Practical parameters accessible by the kernel

The kernel in Figure 8 is a replacement of that in Figure 6 for use in the program of Figure 15. In this kernel the array index `Idx` is formed from the x and y values returned by the `get_global_id()` OpenCL library function for each work-item which executes the kernel. Those x and y values are mapped into the one dimensional form of the `A[]` array by using the limit of the x dimension returned by the `get_global_id(0)` function.

```
const char *programSource =
"__kernel \n"
"void map(__global int *A)    \n"
"{                            \n"
"  int  xgi, ygi, xgs, ygs;   \n"
"  int  xli, yli, xhi, yhi;   \n"
"  int  Idx;                  \n"
"                             \n"
"  xgi = get_group_id(0);     \n"
"  ygi = get_group_id(1);     \n"
"  xli = get_local_id(0);     \n"
"  yli = get_local_id(1);     \n"
"  xgs = get_global_size(0);  \n"
"  ygs = get_global_size(1);  \n"
"  xhi = get_global_id(0);    \n"
"  yhi = get_global_id(1);    \n"
"                             \n"
"  Idx = yhi*xgs + xhi;       \n"
"  A[Idx] = yli;              \n"
"} \n";
```

Figure 8: Kernel to find usable indexing parameters

From the program of Figure 15 the example problem being considered is centred on a two dimensional array `A[][]` of dimension 15 by 12, i.e. 15 elements (numbers) along the y axis and 12 elements along the x axis. The two dimensional array is passed to the kernel as a one dimensional array. In the host program, the two dimensional array is decomposed into 4 work-groups along x axis and 3 work-groups along the y axis. So the 4 work-groups along the x axis contain 3 work-item and the 3 work-groups along the y axis contain 5 work-items.

The OpenCL (pocl) system is to map the 12 work-groups (4 * 3) to 12 CPU cores – one work-group to each CPU core. Each of those 12 work-groups is to process 15 work-items (3 * 5).

The host considers the array `A[][]` as two dimensional. But the kernel considers the matrix as one dimensional. In processing, the kernel must conform to the dimensionality imposed by the host. This is simplified by the use of OpenCL library functions such as in the kernel of Figure 8. By changing the variable assigned to the

A[Idx] array in the kernel of Figure 8 and processing that kernel with the program of Figure 15 an insight into positional data available to a kernel was sought.

Figure 9 and Figure 10 show the x and y coordinates, respectively, of each work-item viewed by the kernel corresponding to the elements of the two dimensional matrix A[][]. These values cover the who range of the A[][] matrix.

```
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
 0     1     2     3     4     5     6     7     8     9    10    11
```

Figure 9: Kernel's view of get_global_id(0) on 12/3 by 15/5 range

```
 0     0     0     0     0     0     0     0     0     0     0     0
 1     1     1     1     1     1     1     1     1     1     1     1
 2     2     2     2     2     2     2     2     2     2     2     2
 3     3     3     3     3     3     3     3     3     3     3     3
 4     4     4     4     4     4     4     4     4     4     4     4
 5     5     5     5     5     5     5     5     5     5     5     5
 6     6     6     6     6     6     6     6     6     6     6     6
 7     7     7     7     7     7     7     7     7     7     7     7
 8     8     8     8     8     8     8     8     8     8     8     8
 9     9     9     9     9     9     9     9     9     9     9     9
10    10    10    10    10    10    10    10    10    10    10    10
11    11    11    11    11    11    11    11    11    11    11    11
12    12    12    12    12    12    12    12    12    12    12    12
13    13    13    13    13    13    13    13    13    13    13    13
14    14    14    14    14    14    14    14    14    14    14    14
```

Figure 10: Kernel's view of get_global_id(1) on 12/3 by 15/5 range

The function get_global_size(0) would return a value of 12, corresponding to Figure 9. The function `get_global_id(1)` would return a value of 15 corresponding to Figure 10.

Figure 11 gives the work-group numbering of the work-items along the x axis of the range. The numbers are the same within each work-group. Similarly, Figure 12 gives the work-group numbering of work-items along the y axis of the range.

```
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
0   0   0   1   1   1   2   2   2   3   3   3
```

Figure 11: Kernel's view of `get_group_id(0)` on 12/3 by 15/5 range

```
0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0
1   1   1   1   1   1   1   1   1   1   1   1
1   1   1   1   1   1   1   1   1   1   1   1
1   1   1   1   1   1   1   1   1   1   1   1
1   1   1   1   1   1   1   1   1   1   1   1
1   1   1   1   1   1   1   1   1   1   1   1
2   2   2   2   2   2   2   2   2   2   2   2
2   2   2   2   2   2   2   2   2   2   2   2
2   2   2   2   2   2   2   2   2   2   2   2
2   2   2   2   2   2   2   2   2   2   2   2
2   2   2   2   2   2   2   2   2   2   2   2
```

Figure 12: Kernel's view of `get_group_id(1)` on 12/3 by 15/5 range

Figure 13 shows the indices of the work-items along the x axis in the individual x axis work-groups.. Similarly, Figure 14 shows the indices of the work-items along the y axis in the individual y axis work-groups. Withing in each work-group, the work-items are independent of all others although spanning the same values.

```
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
0    1    2    0    1    2    0    1    2    0    1    2
```

Figure 13: Kernel's view of `get_local_id(0)` on 12/3 by 15/5 range

```
0    0    0    0    0    0    0    0    0    0    0    0
1    1    1    1    1    1    1    1    1    1    1    1
2    2    2    2    2    2    2    2    2    2    2    2
3    3    3    3    3    3    3    3    3    3    3    3
4    4    4    4    4    4    4    4    4    4    4    4
0    0    0    0    0    0    0    0    0    0    0    0
1    1    1    1    1    1    1    1    1    1    1    1
2    2    2    2    2    2    2    2    2    2    2    2
3    3    3    3    3    3    3    3    3    3    3    3
4    4    4    4    4    4    4    4    4    4    4    4
0    0    0    0    0    0    0    0    0    0    0    0
1    1    1    1    1    1    1    1    1    1    1    1
2    2    2    2    2    2    2    2    2    2    2    2
3    3    3    3    3    3    3    3    3    3    3    3
4    4    4    4    4    4    4    4    4    4    4    4
```

Figure 14: Kernel's view of `get_local_id(1)` on 12/3 by 15/5 range

8 Matrix multiplication

One practical use of 2D OpenCL (pocl) is multiplication of matrices. The program listed in Figure 15 multiplies two matrices of integers, each of dimension 12 x 12.

Matrix A is multiplied by the multiplication and the required result is returned in matrix C. Each work-group performs the multiplication of one row of matrix A and one column of matrix B, and returns the overall addition of those products as one value in the result matrix C. Finally, the A, B, and C matrices are printed.

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>

#define SIZE 12

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

const char *programSource =
"__kernel \n"
"void matmul(__global int *A, \n"
"            __global int *B, \n"
"            __global int *C, \n"
"            int N)           \n"
"{                            \n"
"  int  i;                    \n"
"  int  acc;                  \n"
"                             \n"
"  const int xhi = get_global_id(0); \n"
"  const int yhi = get_global_id(1); \n"
"                                    \n"
"  acc = 0;                          \n"
"  for (i = 0; i < N; i++)           \n"
"    acc += B[i*N + xhi] * A[yhi*N + i]; \n"
"  C[yhi*N + xhi] = acc;             \n"
"                             \n"
"} \n";


int main()
{
  int  i, j;
  int  markerA, markerB;
  cl_platform_id    platform;
  cl_device_id      device;
  cl_int            ret;
  cl_uint           ret_num_devices, ret_num_platforms;
  cl_context        context;
  cl_mem            a_mem_obj, b_mem_obj, c_mem_obj;
  cl_command_queue  command;
  cl_program        program;
  cl_kernel         kernel;
  size_t            global[3];
  size_t            local[3];
  long              dataSize;
  size_t            size_ret;
```

Figure 15: Matrix multiplication (Continues ...)

```
/* prepare data */
  markerA = 1;
  markerB = SIZE*SIZE;
  for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++) {
      markerA++;
      markerB--;
      A[i][j] = markerA;
      B[i][j] = markerB;
      C[i][j] = 0;
    }

  dataSize = SIZE * SIZE * sizeof(int);

/* setup CPU */
  ret = clGetPlatformIDs(1, &platform, &ret_num_platforms);
  ret = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &device,
                       &ret_num_devices);

  context = clCreateContext(NULL, 1, &device, NULL, NULL, &ret);
  command = clCreateCommandQueueWithProperties(context, device, 0, &ret);

  a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, dataSize, NULL,
                             &ret);
  b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, dataSize, NULL,
                             &ret);
  c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY, dataSize, NULL,
                             &ret);

  ret = clEnqueueWriteBuffer(command, a_mem_obj, CL_TRUE, 0, dataSize, A,
                             0, NULL, NULL);
  ret = clEnqueueWriteBuffer(command, b_mem_obj, CL_TRUE, 0, dataSize, B,
                             0, NULL, NULL);
  ret = clEnqueueReadBuffer(command, c_mem_obj, CL_TRUE, 0, dataSize,
                             C, 0, NULL, NULL);

  program = clCreateProgramWithSource(context, 1,
                             (const char **)&programSource, NULL, &ret);
  ret = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
  kernel = clCreateKernel(program, "matmul", &ret);

  ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_mem_obj);
  ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_mem_obj);
  ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_mem_obj);
  i = SIZE;
  ret = clSetKernelArg(kernel, 3, sizeof(int), &i);

  global[0] = SIZE;  global[1] = SIZE;
  local[0] = 1;  local[1] = 1;
  ret = clEnqueueNDRangeKernel(command, kernel, 2, NULL, global,
                               local, 0, NULL, NULL);
```

Figure 15: Matrix multiplication (Continues . . .)

```
  ret = clEnqueueReadBuffer(command, c_mem_obj, CL_TRUE, 0, dataSize, C,
                            0, NULL, NULL);

  ret = clFlush(command);
  ret = clFinish(command);
  ret = clReleaseKernel(kernel);
  ret = clReleaseProgram(program);
  ret = clReleaseMemObject(a_mem_obj);
  ret = clReleaseMemObject(c_mem_obj);
  ret = clReleaseCommandQueue(command);
  ret = clReleaseContext(context);

  for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++)   printf("%4d ", A[i][j]);
    printf("\n");
  }
  printf("\n");

  for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++)   printf("%4d ", B[i][j]);
    printf("\n");
  }
  printf("\n");

  for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++)   printf("%8d ", C[i][j]);
    printf("\n");
  }
  printf("\n");


  return(0);
}
```

Figure 15: Matrix multiplication

Using this approach, there are 144 work-groups in total, each containing 1 work-item. The parameters are set in the `global[]` and `local[]` array in Figure 15.

9 Better parallel matrix multiplication processing

Although the program of Figure 15 performs matrix multiplication by parallel processing, the introduction of *tiles* can increase the amount of processing in parallel. From the matrix viewpoint, the rows and columns are broken into pieces. Because the matrices are two dimensional, the breaking of each dimension into pieces enables those pieces to be gathered together into two dimensional tiles. The matrices are then covered by those tiles. Processing is performed using those tiles and the results from each tile combined to produce the required result.

The kernel in Figure 16 replaces the kernel of Figure 15 to produce matrix multiplication. In the Figure 16 kernel, tiles are used to do the parallel processing. To use this kernel in the program of Figure 15 only other change was the `local[]` array values of 1 in the host program, was replace by a tile size of, say, 3.

In the kernel of Figure 16, local memory is used to hold the matrix tiles being processed. Data is copied from global memory holding the whole `A` and `B`, into local memory `subbA` and `subB`. The matrix multiplication is performed in the local memory which results in faster processing. From Table 3, local memory, which corresponds to L3 cache on CPU cores used by `pocl`, is small but common to all cores. A kernel is executed in each CPU core.

```
const char *programSource =
"__kernel \n"
"void matmul(__global int *A, \n"
"         __global int *B, \n"
"         __global int *C, \n"
"         int N)          \n"
"{                        \n"
"  int  i, j;             \n"
"  int  acc;                          \n"
"  __local int Asub[400];  \n"
"  __local int Bsub[400];  \n"
"                          \n"
"  const int  xgi = get_global_id(0);  \n"
"  const int  ygi = get_global_id(1);  \n"
"  const int  xli = get_local_id(0);   \n"
"  const int  yli = get_local_id(1);   \n"
"  const int  xls = get_local_size(0); \n"
"                          \n"
"  acc = 0;                \n"
"  for (i = 0; i < (N/xls); i++)  { \n"
"    Asub[yli*xls + xli] = A[ygi*N + (i*xls + xli)]; \n"
"    Bsub[yli*xls + xli] = B[(i*xls + yli)*N + xgi]; \n"
"    barrier(CLK_LOCAL_MEM_FENCE);  \n"
"                          \n"
"    for (j = 0; j < xls; j++)      \n"
"      acc += Asub[yli*xls + j] * Bsub[j*xls + xli]; \n"
"    barrier(CLK_LOCAL_MEM_FENCE);  \n"
"  } \n"
"  C[ygi*N + xgi] = acc;  \n"
"                         \n"
"} \n";
```

Figure 16: Matrix multiplication OpenCL kernel using tiles