Engineering: Parallel Computation of the Number π

6

Chapter Summary
In computing the number π, by simple numerical integration, the focus is in parallel implemention on three different parallel architectures and programming environments: OpenMP on the multicore processor, MPI on a cluster, and OpenCL on a GPU. In all three cases a spatial domain decomposition is used for paralelization, but differences in communication between parallel tasks and in combining the results of these tasks are shown. Measurements of the running time and speed-up are included to assist self-studying readers.

A detailed description of the parallel computation of π is available in Chap. 3 Example 3.4 and in Chap. 4 Example 4.4. The solution methodology relies on a numerical integration of unit circle:

$$\pi = 4 \int_0^1 \sqrt{1 - x^2} dx$$

that is in a direct relation with the value of π. The numerical integration is performed by calculation and summation of all N sub-interval areas. A sequential version of the algorithm in a pseudocode, which results in an approximate value of π, is given below:

R. Trobec et al., *Introduction to Parallel Computing*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-98833-7_6

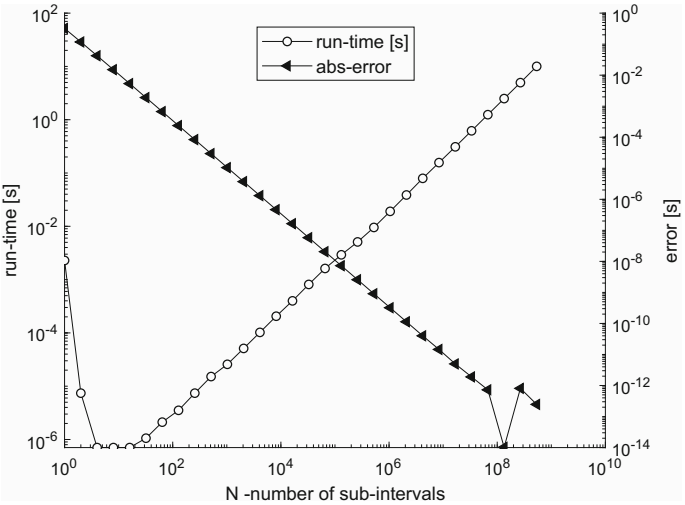

Fig. 6.1 Run-time and absolute error on a single MPI process in computation of π as a function of number of sub-intervals N

Algorithm 1 SEQUENTIAL ALGORITHM: COMPUTE_π

Input: N - number of sub-intervals on interval [0, 1]

1: **for** $i = 1 \ldots N$ **do**
2: $x_i = (1/N)(i - 0.5)$
3: $y_i = \sqrt{1 - x_i^2}$
4: $Pi = Pi + 4(y_i/N)$
5: **end for**

Output: Pi - an approximation for the number π

We validate the Algorithm 1 on a single computer in order to prove its correct behavior. It is expected that with an increased number of sub-intervals N, the approximation of π will become better and better, which should be confirmed by calculated absolute error of approximate π value. This is easy, because we know the π value with arbitrary accuracy. However, with the increased N the run-time will also increase. Embedding the existing MPI program from Listing 4.5 in an additional `for` loop that increases the number of intervals by a factor of two, followed by compiling and running the program:

```
>mpiexec -n 1 MSMPIPi
```

on a HP EliteBook 840 notebook, based on Intel Core 64-bit processor i7-7500U CPU with 2 physical cores and 4 logical processor, on MS Windows 10 operating system with Visual Studio 2017 compiler, we get the results shown in Fig. 6.1. Note, that for all presented MPI experiments in this book, the same notebook was used. We compile in `Release` mode with optimization for maximal speed, e.g., `/O2`.

To see the full response, the results are shown in logarithmic scale on both axes. The run-time mostly increases as expected, except with a few smallest values of sub-

intervals, where the impact of MPI program setup time, cache memory, or interactions with operating system could be present. In the same way, the approximation error becomes smaller and smaller, until the largest number of intervals, where a small jump is presents, possibly because of a limited precision of the floating point arithmetic.

The next step is to find out the most efficient way to parallelize the problem, i.e., to engage a greater number of cooperating processors in order to speed up the program execution. Even though the sequential Algorithm 1 is very simple, it implies most of the problems that arise also in more complex examples. First, the program needs to distribute tasks among cooperating processors in a balanced way. A relatively small portion of data should be communicated to cooperating processes, because the processes will generate their local data by a common equation for a unit circle. All processes have to implement their local computation of partial sums, and finally, the partial results should be assembled, usually by a global communication, in a host process to be available for users.

Regarding sequential Algorithm 1, we see that the calculation of each sub-interval area is independent, and consequently, the algorithm has a potential to be parallelized. In order to make the calculation parallel, we will use domain decomposition approach and master–slave implementation. Because all values of y_i can be calculated locally and because the domain decomposition is known explicitly, there is no need for a massive data transfer between the master process and slave processes. The master process will just broadcast the number of intervals. Then, the local integration will run in parallel on all processes. Finally, the master process reduces the partial sums into the final approximation of π. The parallelized algorithm is shown below:

Algorithm 2 PARALLEL ALGORITHM: COMPUTE_π

Input: N - number of sub-intervals on interval [0, 1]

1: Get $myID$ and the number of cooperating processes p
2: Master broadcast N to all processes
3: Compute a shorter **for** loop:
4: **for** $j = 1 \ldots N/p$ **do**
5: $\quad x_j = (1/N)(j - 0.5)$
6: $\quad y_j = \sqrt{1 - x_j^2}$
7: $\quad Pj = Pj + 4(y_j/N)$
8: **end for**
9: Master reduce partial sums Pj to the final result Pi

Output: Pi - approximate value of π

We have learned from this simple example that, besides the calculation, there are other tasks to be done (i) domain decomposition, (ii) their distribution, and (iii) assembling of the final result, which are inherently sequential, and therefore limit the final speedup. We further see that all processes are not identical. Some of the processes are slaves because they just calculate their portion of data. The master process has to distribute the number of intervals and to gather and sum up the

local results. Parallel implementation approaches on different computing platforms differ significantly, and therefore their results are presented in the following sections, separately for: OpenMP, MPI, and OpenCL.

6.1 OpenMP

Computing π on a multicore processor has been covered in Chap. 3.

The numerical integration of a unit square (the part of it that lies in the first quadrant) has been explained in Example 3.4, where the program for computing π is shown Listing 3.15. To analyze the performance of the program, it has been run on a quad-core processor with hyperthreading (Intel Core i7 6700HQ). For 10^9 subintervals of the interval $[0, 1]$ (when the error is approximately 10^{-8}), the results are shown in Fig. 6.2: the bars show the measured wall clock time and the dashed curve illustrates the expected wall clock time in case of the ideal speedup in regard to the number of threads used.

The wall clock time decreases when the number of threads increases, but only up to the number of logical cores the processor can provide. Once the number of threads exceeds the number of logical cores, the program (its OpenMP run-time component, to be precise) places multiple threads on the same core and no reduction of wall clock time can be gained.

In fact, one must observe that up to the number of logical cores, almost ideal speedup is achieved. This is not to be expected very often. In this case, however, it is a consequence of the fact that the entire computation is almost perfectly parallelizable, with the exception of the final reduction. But if 10^9 intervals are divided among 8 threads, the time of the reduction becomes insignificant if compared to the computation of the local sums.

As shown in Example 3.5, π can also be computed by random shooting into the square $[0, 1] \times [0, 1]$ and count the number of shots that hit inside the unit square. The program for computing π using this method is shown in Listing 3.18. As with the numerical integration, the program has been tested on a quad-core processor with hyperthreading (Intel Core i7 6700HQ). For 10^8 shots, the measured wall clock time is shown in Fig. 6.3. Again, the dashed line illustrates the expected wall clock time in case of the ideal speedup in regard to the number of threads used.

As can be seen in Fig. 6.3, (almost) ideal speedup is achieved only for up to 4 threads, i.e., for one thread per physical, not logical core. That implies that instructions and memory accesses of threads placed on the same physical core result in too many conflicts to sustain the speedup and truly benefit from multithreading. This can happen and it is a lesson not to be forgotten.

Even though the wall clock time is what it matters in the end, the CPU time has been measured as well. In Fig. 6.4 the total amount of CPU time needed for computing π using both methods explained in Chap. 3, namely numerical integration and random shooting, is shown.

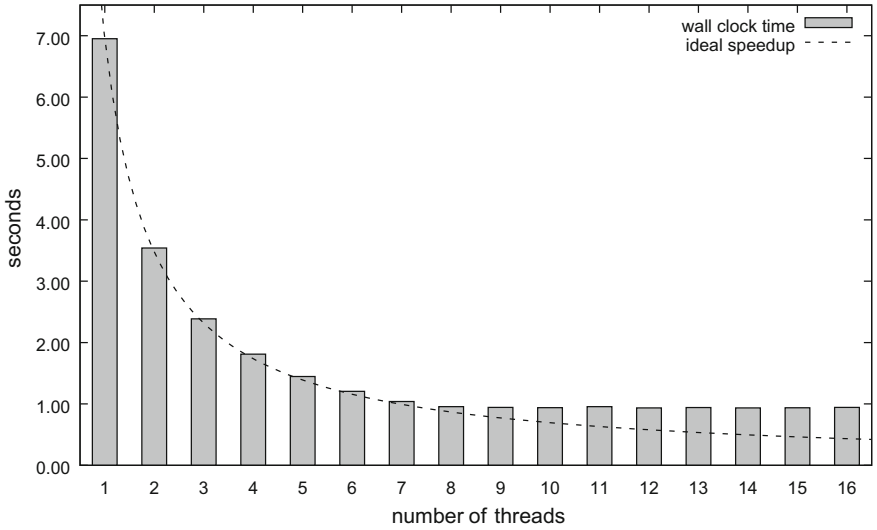

Fig. 6.2 Computing π using the numerical integration of the unit circle using 10^9 intervals on a quad-core processor with hyperthreading

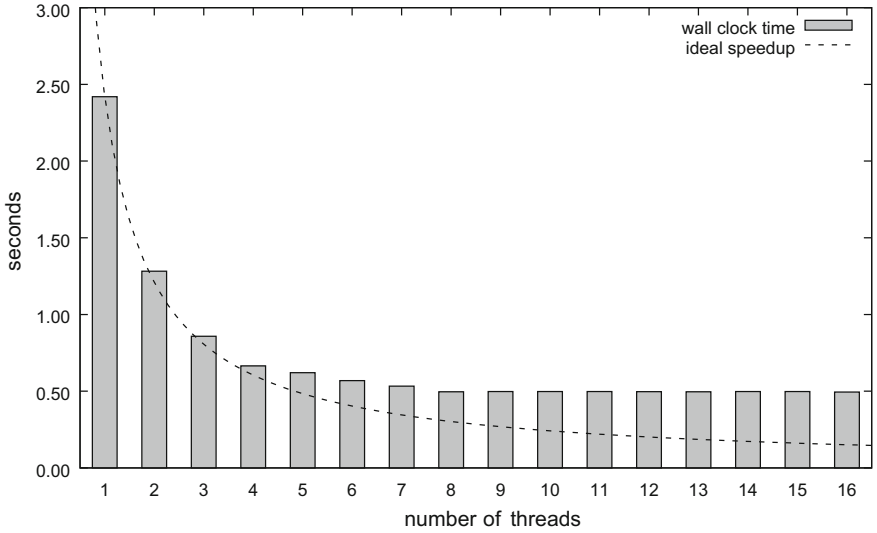

Fig. 6.3 Computing π using random shooting into the square $[0, 1] \times [0, 1]$ using 10^8 shots

Although the wall clock time shown in Fig. 6.2 in Fig. 6.3 decreases with the number of threads the total amount of CPU time increases. These can be expected, since more threads require more administrative tasks from the OpenMP run-time.

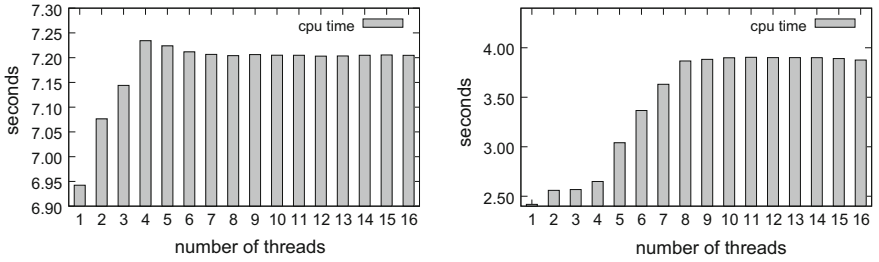

Fig. 6.4 The total CPU time needed to compute π using numerical integration (left) and random shooting (right)

6.2 MPI

An MPI C code for the parallel computation of π, together with some explanation and comments, are provided in Chap. 4 Listing 4.5. We would like to test the behavior of run-time as a function of the number of MPI processes p. On the test notebook computer, two cores are present. Taking into account that four logical processors are available, we could expect some speedup of the execution with up to four MPI processes. With more than four processes, the run-time could start increasing, because of an MPI overhead. We will test our program with up to eight processes. Starting a same program on different number of processes can be accomplished by consecutive `mpiexec` commands with appropriate value of parameter `-n` or by a simple bash file that prepares the execution parameters, which are passed to the `main` program through its `argc` and `argv` arguments.

The behavior of approximation error should be the same as in the case of a single process. In the computation of π, the following number of sub-intervals have been used $N = [5e9, 5e8, 5e7, 5e6]$ (5e9 is a scienific notation for 5.10^9). Note, that such big numbers of sub-intervals were used because we want to have a computationally complex task, even that the computation of sub-interval areas is quite simple. Usually, in realistic tasks, there is much more computation by itself and tasks become complex automatically. Two smaller values of N have been used to test the impact of the ratio calculation/communication complexity on the program execution. The obtained results for parallel run-times (RT) in seconds and speedups (SU), in computation of π, on a notebook computer are shown in Fig. 6.5.

We have first checked that the error in parallel approximation of π is the same as in the case of a single process. The run-time behaves as expected, with the maximum speedup of 2.6 with four processes and large N. With two processes the speedup is almost 2, because the physical cores have been allocated. Up to four processes, the speedup increases but not ideal, because logical processors cannot provide the same performance as the physical cores because of hyperthreading technology. The program is actually executed on a shared memory computer with potentially negligible communication delays. However, if N is decreased, e.g., to 5e6 or more, the

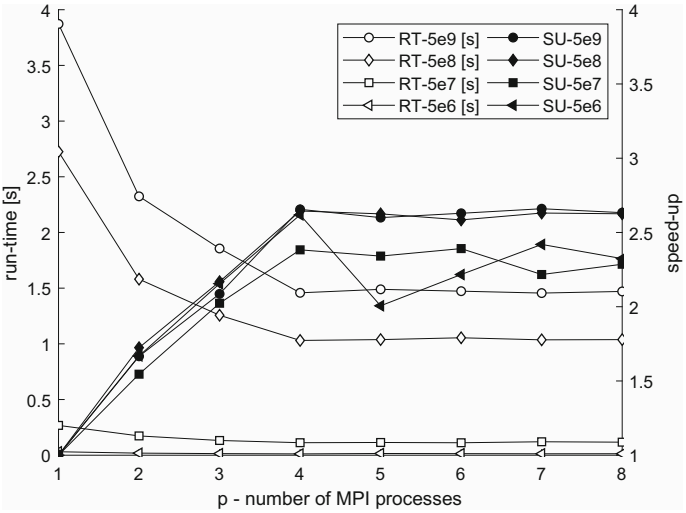

Fig. 6.5 Parallel run-time (RT) and speedup (SU) in computation of π on a notebook computer for $p = [1, \ldots, 8]$ MPI processes and $N = [5e9, 5e8, 5e7, 5e6]$ sub-intervals

speedup is becoming smaller because more processes introduce a larger execution overhead that diminish the speedup.

Let us finally check the behavior of the parallel MPI program on a computing cluster. It is built of 36 computing nodes connected in a 6×6 mesh, each with 6 Gigabit ports to a Gigabit switch. Computing nodes are built as a dual 64-bit CPU Intel Xeon 5520, each of CPUs with 4 physical cores (two threads/core) and 6 GB of local memory. The computing cluster runs under server version of Ubuntu 16.04.3 LTS with GCC Version 7.3 compiler. Only 8 out of 36 interconnected cluster computers (CPUs) have been devoted for our tests, resulting in 32 physical cores. All programs are compiled for maximum speed. Note, that the same computing cluster has been used in all presented MPI tests of this book. The hostfile is:

```
k1:4 k2:4 k3:4 k4:4 k5:4 k6:4 k7:4 k8:4
```

where `k1...k8` are names of 8 cluster computing nodes.

Because, in the π test case program, there is no significant communication load, and because two threads/core are available, we expect practically ideal speedup up to 64 MPI processes. Then, if more processes are generated, the speedup is not so predictable. We will try to explain the results after performing all experiments.

The program is compiled with:

```
>mpicc.mpich -O3 MPIPI.c -o MPIPI
```

Parallel program performances are tested on MPICH MPI with various options for `mpirun.mpich`. First, we run `np = 1...128` experiments, for 1 to 128 MPI processes, with default parameters:

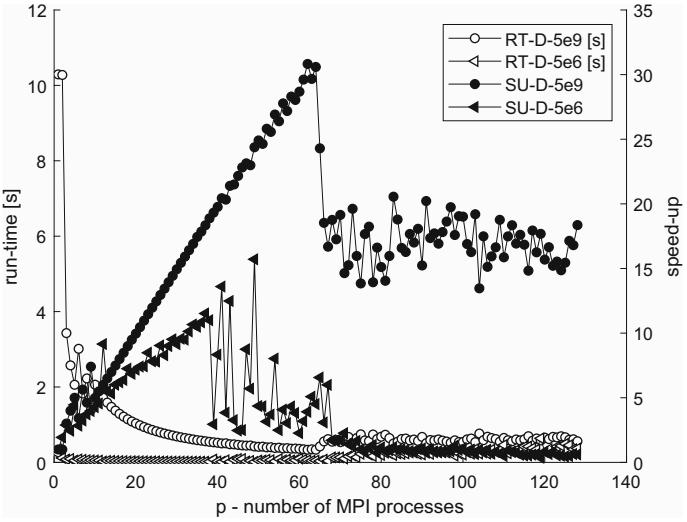

Fig. 6.6 Parallel run-time (RT-D) and speedup (SU-D) in computations of π on a cluster of 8 interconnected computers with total 32 cores and $p = [1, \ldots, 128]$ MPI processes with default parameters of `mpirun.mpich`

```
>mpirun.mpich --hostfile myhosts.mpich.txt -np $np ./MPIPi $N
```

Parameters `N` and `np` are provided from `bash` file as $N = [5e9, 5e8, 5e7, 5e6]$ and $\texttt{np} = [1, \ldots, 8]$. The maximum number of MPI processes, i.e., 128, was determined from the command line when running the bash file:

```
>./run.sh 128 > data.txt
```

where `data.txt` is an output file for results. The obtained results for parallel run-times, in seconds, with default parameters of `mpirun` (RT-D) and corresponding speedups (SU-D), are shown in Fig. 6.6. For better visibility only two pairs of graphs are shown, for largest and smallest N.

Let us look first the speedup for $N = 5e9$ intervals. We see that the speedup increases up to 64 processes, where reaches its maximal value 32. For more processes, it drops and deviates around 17. The situation is similar with thousand times smaller number of intervals $N = 5e6$, however, the maximal speedup is only 5 and for more than 64 processes there is no speedup. We expected this, because calculation load decreases with smaller number of sub-intervals and the impact of communication and operating system overheads prevail.

We further see that the speedup scales but not ideal. 64 MPI processes are needed to execute the program 32 times faster as a single computer. The reason could be in the allocation of processes along the physical and logical cores. Therefore, we repeat experiments with `mpirun` parameter `-bind-to core:1`, which forces to run just a single process on each core and possibly prevents operating system to move

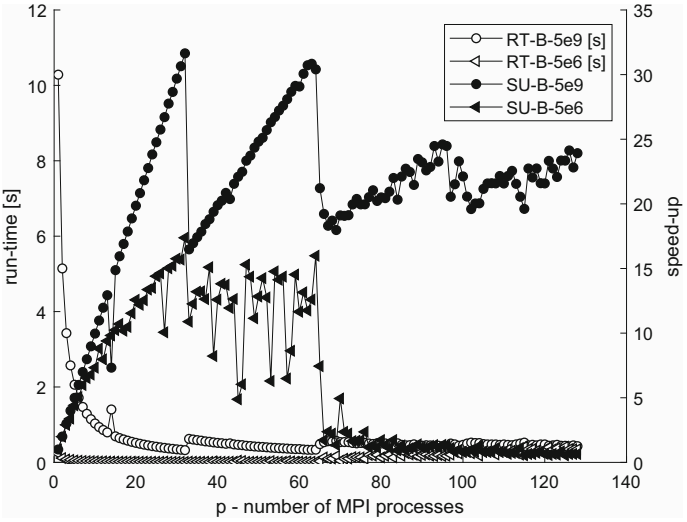

Fig. 6.7 Parallel run-time (RT-B) and speedup (SU-B) in computations of π on a cluster of 8 interconnected computers for $p = [1, \ldots, 128]$ MPI processes, bound to cores

processes around cores. The obtained results for parallel run-times in seconds (RT-B) with processes bound to cores and corresponding speedups (SU-B), are shown in Fig. 6.7. The remaining execution parameters are the same as in previous experiment.

The `bind` parameter improves the execution performance with $N = 5e9$ intervals in the sense that the speedup of 32 is achieved already with 32 processes, which is ideal. But then the speedup falls abruptly by a factor of 2, possibly because of the fact, that with more than 32 MPI processes, some processing cores must manage two processes, which slows down the whole program execution.

We further see that with more than 64 processes speedups fall significantly in all tests, which is a possible consequence of inability to use the advantage of hyper-threading. With larger number of processes, larger than the number of cores, on several cores run more than two processes, which slows down the whole program by a factor of 2. Consequently, the slope of speedup scaling, with more than 32 processes, is also reduced by 2. With this reduced slope, the speedup reaches the second peak by 64 processes. Then the speedup falls again to an approximate value of 22.

The speedup with $N = 5e6$ intervals remains similar as in previous experiment because of lower computation load. It is a matter of even more detailed analysis, why the speedup behaves quite unstable for some cases. The reasons could be in cache memory faults, MPI overheads, collective communication delays, interaction with operating system, etc.

6.3 OpenCL

If we look at Algorithm 1, we can see that it is very similar to dot product calculation covered in Sect. 5.3.4. We use the same principle: we will use a buffer in local memory named `LocalPiValues` to store each work-item's running sum of the π value. This buffer will store `szLocalWorkSize` partial π values so each work-item in the work-group will have a place to store its temporary result. Then, we use the principle of reduction to sum up all π values in the work-group. We now store the final value of each work-group to an array in global memory. Because this array is relatively small, we return control to the host and let the CPU finish the final step of the addition. Listing 6.1 shows the kernel code for computing π.

```
__kernel void CalculatePiShared(
                            __global float* c,
                            ulong iNumIntervals)
{
    __local float LocalPiValues[256]; // work-group size = 256

    // work-item global index
    int iGID = get_global_id(0);
    // work-item local index
    int iLID = get_local_id(0);
    // work-group index
    int iWGID = get_group_id(0);
    // how many work-items are in WG?
    int iWGS = get_local_size(0);

    float x = 0.0;
    float y = 0.0;
    float pi = 0.0;

    while (iGID < iNumIntervals) {
     x = (float)(1.0f/(float)iNumIntervals)*((float)iGID-0.5f);
     y = (float)sqrt(1.0f - x*x);
     pi += 4.0f * (float)(y/(float)iNumIntervals);
     iGID += get_global_size(0);
    }

    //store the product
    LocalPiValues[iLID] = pi;
    // wait for all threads in WG:
    barrier(CLK_LOCAL_MEM_FENCE);

    // Summation reduction:
    int i = iWGS/2;
    while(i!=0){
        if (iLID < i) {
            LocalPiValues[iLID] += LocalPiValues[iLID+i];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        i=i/2;
    }

    // store partial dot product into global memory:
    if (iLID == 0) {
        c[iWGID] = LocalPiValues[0];
    }
}
```

Listing 6.1 The compute π kernel.

Table 6.1 Experimental results for OpenCL π computation

No. of intervals	CPU time [s]	GPU time [s]	Speedup
10^6	0.01	0.0013	7.69
33×10^6	0.31	0.035	8.86
10^9	9.83	1.07	9.18

To analyze the performance of the OpenCL program for computing π, the sequential version has been run on a quad-core processor Intel Core i7 6700HQ running at 2,2 GHz, while the parallel version has been run on an Intel Iris Pro 5200 GPU running at 1,1 GHz. This is a small GPU integrated on the same chip as the CPU and has only 40 processing elements. The results are presented in Table 6.1. We run the kernel in NDrange of size:

```
szLocalWorkSize = 256;       // # of work-items in work-group
szGlobalWorkSize = 256*128; // total # of work-work-items
```

As can be seen from the measured execution times, noticeable acceleration is achieved, although we do not achieve the ideal speedup. The main reason for that lies in reduction summation that cannot be fully parallelized. The second reason is the use of complex arithmetic operations (square root). The execution units usually do not have their own unit for such a complex operation, but several execution units share one special function unit that performs complex operations such as square root, sine, etc.