

Comparison of Parallelization Frameworks for Shared Memory Multi-Core Architectures

Richard Membarth¹, Frank Hannig¹, Jürgen Teich¹,
Mario Körner², and Wieland Eckert²

¹ Hardware/Software Co-Design, Department of Computer Science,
University of Erlangen-Nuremberg, Germany.

{richard.membarth,hannig,teich}@cs.fau.de

² Siemens Healthcare Sector, H IM AX,
Forchheim, Germany.

{mario.koerner,wieland.eckert}@siemens.com

Abstract. The development of standard multi-core processors changed in the last years moving from bigger, more complex, and faster cores to putting several more simple cores onto one chip. This changed also the way programs are written in order to leverage the processing power of multiple cores of the same processor. In the beginning, programmers had to divide and distribute the work by hand to the available cores and to manage threads in order to use more than one core. In the meantime, several frameworks exist to relieve the programmer from such tasks. In this paper, we present four of these frameworks for parallelization on shared memory multi-core architectures, namely OpenMP, Cilk++, Threading Building Blocks, and RapidMind. We use an example that brings up problems to be considered during parallelization in order to get correct results. Therefore, we use histogram generation which can be implemented straightforward on single-core processors. However, calculated in parallel the non-regular memory access pattern offers potential for race conditions that bias the result. Based on this example, we compare the frameworks and shed light on their distinct features.

1 Introduction

Multi-core processors have become mainstream within the last years. Today, processors with two, four, or even more cores are standard in personal computers and even in portable devices such as notebooks and mobile phones—thus, architectures with multiple cores are ubiquitous in our daily life. Most times, the massive computing power offered by these architectures, is only utilized at a very coarse-grained level of parallelism, namely *task-level parallelism* or *application-level parallelism*. That means, the different processor cores are employed by the operating system, more specific by the scheduler, which assigns applications or tasks to the available cores. In order to accelerate the execution of a large number of running tasks, for instance, in a data base server or in network processing, such a work-load distribution might be ideal.

However, if the performance of a single algorithm shall be increased, more fine-grained parallelism (for example, *data-parallelism* or *instruction-level parallelism*) has to be considered. In this case, the parallelization cannot be left to the operating system, but the software developer has to take care of the parallelization by hand or by using appropriate tools and compilers. Here, techniques like *loop unrolling*, *affine loop transformations*, or *loop tiling* [2, 6, 10] are often the agent of choice in order to parallelize an algorithm onto several cores, to adapt to a cache hierarchy, or to achieve a high resource utilization at instruction-level, with the overall goal to speed up the execution of one algorithm (latency minimization) or to achieve a higher data throughput. Arisen from these needs, a number of *parallelization frameworks*³ have been evolved that assist a developer when writing parallel programs.

In this paper, we present four of these frameworks for parallelization on standard shared memory multi-core architectures, namely OpenMP, Cilk++, Threading Building Blocks, and RapidMind. We use an example that brings up problems to be considered during parallelization in order to get correct results. Therefore, we use histogram generation which can be implemented straightforward on single-core processors, but has non-regular memory access patterns and offers potential for race conditions when done in parallel. Based on this we show the differences between the used frameworks.

The remaining paper is organized as follows: Section 2 introduces the multi-core frameworks used within this paper and the following Section 3 compares the frameworks. Finally, in Section 4 conclusions of this work are drawn.

2 Multi-Core Frameworks

In this section, we describe briefly the properties of the four investigated multi-core frameworks and their parallelization approaches. This gives a better understanding of the targeted platforms and application area as well as of possible deployment in existing environments.

2.1 OpenMP

The Open Multi-Processing (OpenMP) is a standard that defines an application programming interface to specify shared memory parallelism in C, C++, and Fortran programs [8]. The OpenMP specification [7] is implemented by all major compilers like Microsoft's compiler for Visual C++ (MSVC), the GNU Compiler Collection (gcc), or Intel's C++ compiler (icc). OpenMP provides preprocessor directives, so called *pragmas* to express parallelism. These pragmas specify which parts of the code should be executed in parallel and how data should be shared. The basic idea behind OpenMP is a fork-join model, where one master thread

³ The term *parallelization framework* is used within this paper as follows: A parallelization framework is a software solution that abstracts from the underlying threading concepts of the operating system and provides the programmer the possibility to express parallelism without worrying about the implementation details.

executes throughout the whole program and forks off threads to process parts of the program in parallel. OpenMP provides different work-sharing constructs, which allow to express two types of parallelism, namely task parallelism and data parallelism.

2.2 Cilk++

Cilk++ [3] is a commercial version of the Cilk [1] language developed at the MIT for multithreaded parallel programming. Cilk++ was recently acquired by Intel and is since then available from them. It is an extension to the C++ language adding three basic keywords to process loops in parallel, launch new tasks, and synchronize between tasks. These keywords allow to express task as well as data parallelism. In addition, Cilk++ provides *hyperobjects*, constructs that solve data race problems created by parallel access of global variables without locks. For code generation, Cilk++ provides two compilers, one based on MSVC for Windows platforms and one based on gcc for GNU/Linux. Cilk++ provides also tools to detect race conditions and to estimate the achievable speedup and inherent parallelism of *cilkified* programs. Its run-time system implements an efficient work-stealing algorithm that distributes the workload to idle processors.

2.3 Threading Building Blocks

Threading Building Blocks (TBB) [9] is a template library for C++ developed by Intel to parallelize programs and is available as an open source version as well as a commercial version providing further support. Parallel code is encapsulated in special classes and invoked from the program. TBB allows to express task and data parallelism. All major compilers can be used to generate binaries from TBB programs. Instead of encapsulating the parallel code in classes, *lambda functions* can be used to express parallelism in a more compact way. There are, however, only few compilers supporting *lambda functions* of the upcoming *c++0x* standard. TBB provides also concurrent container classes for hash maps, vectors, or queues as well as own mutex and lock implementations. The run-time system of TBB schedules the tasks using a work-stealing algorithm similar as Cilk++.

2.4 RapidMind

RapidMind [5] is a commercial solution that emerged from a high-level programming language for graphics cards, called Sh [4] and was recently acquired by Intel. While Sh was targeting originally only graphics cards, RapidMind takes a data-parallel approach that maps well onto many-core hardware as well as on standard shared memory multi-core architectures and the Cell Broadband Engine. RapidMind programs follow the single program, multiple data (SPMD) paradigm where the same function is applied data parallel to all elements of a large data set. These programs use an own language and data types, are compiled at run-time, and called from standard C/C++ code. Through dynamic

compilation, the code can be optimized for the underlying hardware at run-time and the same code can be executed on different backend like standard multi-core processors and graphics cards. All this functionality is provided by libraries and works with the standard compilers on Windows, GNU/Linux, and Mac OS X.

3 Comparison

In this section, we compare the development using the four frameworks on the basis of a relevant real world example. Most parallelism comes from data parallel processing of large data sets where the function or algorithm can be applied independently to each element of the data set. This kind of algorithms can be mapped using each of the frameworks similarly well. However, there are also a few algorithms where race conditions arise or the elements cannot be processed independently due to data dependencies. We pick one of these examples that show the differences between the frameworks, which is histogram generation. We do not use a task parallel example since RapidMind was not intended to support this type of parallelism.

3.1 Histogram Generation

Histograms are used in many fields, for instance, in image processing to visualize and modify the intensity spectrum of images, but also to compare the similarity of images. Therefore, a histogram represents the frequency distribution of an image. For colored images, histograms can be generated for each color channel, but also for the entire image over all color channels. For simplicity, we consider here only implementations for grayscale images, but the principle applies also to colored images. Figure 1(a) shows a grayscale image and the histogram of that image is shown in Fig. 1(b).

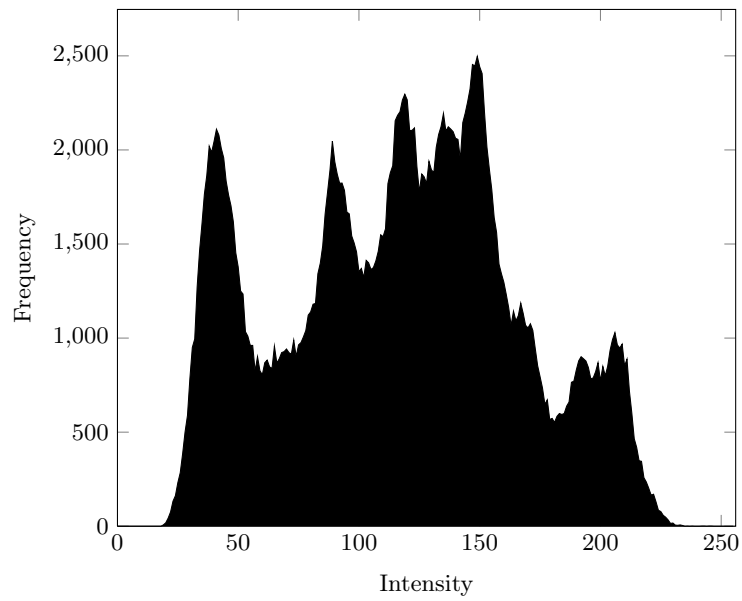
To calculate the histogram of an image, we need a counter for each possible intensity in the image. These *bins* are stored in an array and for each pixel in the image we increment the counter corresponding to the pixels' intensity. That is, for a grayscale image with a resolution of 8-bit, we need $2^8 = 256$ bins. Listing 1 shows an implementation to calculate the histogram of an image sequentially on a standard processor. Two loops are iterating over the width and the height of the image and the bin of the histogram of the current pixels' value is incremented by one.

Listing 1: Sequential code for histogram generation.

```
1 for (int y=0; y<height; y++) {
2     for (int x=0; x<width; x++) {
3         int img_value = img[x + y*width];
4         hist[img_value] += 1;
5     }
6 }
```



(a) Grayscale image.



(b) Histogram of image (a).

Fig. 1: Histogram of a gray-scale image: (a) shows the gray-scale image and (b) shows the histogram of the image. The histogram shows the frequency distribution of the intensity in the image.

There is no problem calculating the histogram sequentially, however, when this is done in parallel race condition may occur. When two or more threads try to increment the same bin simultaneously, it is not clear if the bin is incremented by one or two after both threads updated the bin: Thread t_0 reads the counter value from the bin increments the value by one. If thread t_1 reads the counter value from the bin before thread t_0 writes its result back to memory, thread t_1 increments the same value as thread t_0 does. Eventually the counter is only incremented by one instead of two. At such code segments, the programmer has to ensure either that the counters are incremented sequentially or find a different solution that has no race conditions.

3.2 Histogram Generation Using OpenMP

Using OpenMP, the sequential code can be extended using compiler directives as seen in Listing 2. Firstly, the outer for loop is parallelized using the *omp parallel for* directive. This directive has several clauses to specify how the data is shared between the threads, for example data can be *shared* between threads or *private* to a thread. The loop iteration variables x and y are private. The same applies to the *width* and *height* variables, however, these have to be initialized in addition before the loop is entered. This is expressed by the *firstprivate* clause. Secondly, the access to the bins of the histogram has to be synchronized. This can be accomplished by the *omp critical* directive. All instruction within the specified section are executed only by one thread at a time and resolves the race condition described previously.

Listing 2: Histogram generation using OpenMP. See Appendix A.1 for a more efficient implementation.

```
1 #pragma omp parallel for default(none) shared(img, hist) private
   (x, y) firstprivate(height, width)
2 for (int y=0; y<height; y++) {
3     for (int x=0; x<width; x++) {
4         int img_value = img[x + y*width];
5         #pragma omp critical
6         {
7             hist[img_value] += 1;
8         }
9     }
10 }
```

Using compiler directives to parallelize the code, OpenMP provides an easy way to use the cores found in today's commodity hardware and to solve problems encountered during parallelization. It integrates easily into existing workflows as long as the compiler supports OpenMP. However, there is no support by OpenMP to detect race conditions, which makes development cumbersome since race conditions have a nondeterministic behavior.

3.3 Histogram Generation Using Cilk++

Cilk++ introduces several keywords like *cilk_for* to process loops in parallel, *cilk_spawn* to launch parallel tasks, and *cilk_sync* to synchronize between these tasks. For the histogram example, we replace the outer *for* keyword by *cilk_for* in order to parallelize the histogram generation. This is all to parallelize the loop. To synchronize the access to the histogram bins, we use a *mutex* provided by Cilk++, which ensures that only one thread enters the section protected by the mutex at a time.

Listing 3: Histogram generation using Cilk++. See Appendix A.2 for a more efficient implementation.

```
1  cilk::mutex c_mutex;
2
3  cilk_for (int y=0; y<height; y++) {
4      for (int x=0; x<width; x++) {
5          int img_value = img[x + y*width];
6          c_mutex.lock();
7          hist[img_value] += 1;
8          c_mutex.unlock();
9      }
10 }
```

Cilk++ provides an easy way to parallelize existing code with little changes to the original source code. However, the compiler provided by Cilk++ has to be integrated into the existing workflow. To detect race conditions, Cilk++ provides the *Cilkscreen Race Detector*. This tool runs the parallel application on one core and reports any location in the program that may result in a race condition, that is writing to the same memory location. To estimate the achievable speedup of an application, the *Cilkscreen Parallel Performance Analyzer* counts the instructions of the executed application and gives information of the potential inherent parallelism of the currently executed and *cilkified* application. This gives an upper bound of the maximum concurrency in the current implementation and helps to assess the implementation. It helps also to see if all cores of the current hardware can be fully utilized by the implementation.

3.4 Histogram Generation Using Threading Building Blocks

TBB provides template functions for parallelization like the *parallel_for* template function to execute loops in parallel. The *parallel_for* template functions requires iteration space information and a function object as parameter. The function object is normally declared within a separate class for each function. This incurs a lot of overhead in terms of additional lines of code for parallelization. Therefore, *parallel_for* supports also *lambda functions* as argument, which allows to use the function directly as parameter. This is used here to reduce the overhead. The first three parameters specify the lower bound, the upper bound, and the increment of the iteration space. To synchronize the bin access, we use again a mutex provided by TBB.

Listing 4: Histogram generation using TBB. See Appendix A.3 for a more efficient implementation.

```
1 spin_mutex hist_mutex;
2
3 parallel_for(int(0), int(height), int(1), [&] (int y) {
4     for (int x=0; x<width; x++) {
5         int img_value = img[x + y*width];
6         {
7             spin_mutex::scoped_lock lock(hist_mutex);
8             hist[img_value] += 1;
9         }
10    }
11 });
```

Using TBB allows to parallelize in a way that integrates seamlessly into existing object-oriented C++ projects. It provides libraries for parallelization and requires no changes to the workflow as long as the compiler supports ISO C++. Unfortunately, *lambda functions* are currently only supported by some compilers and also only some template functions of TBB support *lambda functions*. Hence, in most cases a lot of additional source code has to be written for parallelization. TBB provides like OpenMP no support to detect race conditions. This has to be done by the programmer or by other tools.

3.5 Histogram Generation Using RapidMind

The previously described frameworks require only little code modification or annotations for parallelization, whereas the approach taken by RapidMind is different. RapidMind uses an own language and data types for data parallel processing of large data sets. The data types used are, for instance, *Value1i* for a single integer and *Value1f* for a single float. RapidMind programs start with the *BEGIN* keyword and end with the *END* keyword. The parameters for the programs are declared as input variables using the *In* keyword and variables returned use the *Out* keyword. Also for loops and if statements have own keywords. When a RapidMind program is executed, an array is passed as parameter to the program and the result is again stored to an array. The program is executed for each element in the array and the element gets assigned to the input variable. Accordingly, the output variable is stored to the corresponding location of the output array. Therefore, the size of the input and output arrays have to match. That is, the approach taken previously to calculate the histogram cannot be used here since the size of the input image and of the output histogram do not match. The benefit of the approach taken by RapidMind is that it matches very well for many problems and that no race conditions can occur by design.

The approach we use here to generate the histogram is as follows: We calculate many subhistograms in parallel and reduce these subhistograms in a second step to the final histogram. One subhistogram is calculated per image line and stored to an array containing all subhistograms. Here, the input to the program

is a grid, where each element corresponds to one line in the image and one output element is a complete subhistogram. Using this approach, one thread can be used per image line resulting in a lower degree of parallelism compared to the previous approach. In a second step these subhistograms are reduced to one histogram. One thread per bin is used to add the corresponding counters of the subhistograms to the final counter value.

Listing 5: Histogram generation using RapidMind. See Appendix A.4 for a more efficient implementation.

```

1  /* temporary array, storing one histogram per line */
2  Array<1, Value1i> hist_tmp = (MAX_INTENSITY*height);
3  Array<1, ArrayAccessor<1, Value1i> > hist_tmp_accessor = dice(
   hist_tmp, MAX_INTENSITY);
4
5  /* calculate one histogram per line */
6  Program rm_hist_diced = BEGIN {
7      In<Value1i> line;
8      Out<Array <1, Value1i> > res(MAX_INTENSITY);
9
10     FOR (Value1i i=0, i<MAX_INTENSITY, i++) {
11         res[i] = 0;
12     } ENDFOR;
13     FOR (Value1i i=0, i<width, i++) {
14         Value1i img_val = img[Value2ui(i, line)];
15         res[img_val] = res[img_val] + 1;
16     } ENDFOR;
17 } END;
18
19 /* reduce histograms into final histogram */
20 Program rm_hist_reduce = BEGIN {
21     In<Value1i> pos;
22     Out<Value1f> res;
23
24     res = 0;
25     FOR (Value1i i=0, i<height, i++) {
26         res += hist_tmp[pos + i*MAX_INTENSITY];
27     } ENDFOR;
28 } END;
29
30 hist_tmp_accessor = rm_hist_diced(grid(height));
31 hist = rm_hist_reduce(grid(MAX_INTENSITY));

```

RapidMind introduces an own syntax to parallelize data parallel algorithms. Existing programs have to be mapped to this syntax, which involves a complete rewrite of the function to be executed in parallel. However, programs written in RapidMind can be executed on different platforms like standard multi-cores and graphics cards with almost no change to the source code. The syntax matches very well data parallel problems and provides built-in solutions for a lot of common tasks like border handling. Mapping of non-regular algorithms is sometimes

tricky, though. Race conditions are eliminated by design. The functionality of RapidMind is provided by libraries and should work by all standard compilers on Windows, GNU/Linux, and Mac OS X.

4 Conclusions

In this paper, we presented four parallelization frameworks for standard shared memory multi-core architectures, namely OpenMP, Cilk++, Threading Building Blocks, and RapidMind. We showed how to generate histograms using each of the frameworks and to resolve problems like race conditions that arise when the histogram is calculated in parallel. On the one hand, OpenMP, Cilk++, and Threading Building Blocks are similar and allow parallelization with only minor source annotations or extensions. However, OpenMP and TBB do not provide any support detecting and resolving race conditions. In contrast, Cilk++ provides tools reporting race conditions and the estimated speedup of the program. On the other hand, RapidMind comes along with an own syntax to write parallel code, which is free of deadlocks and race conditions by design. RapidMind allows to compile the code at run-time to several backends with no code modifications like standard multi-core architectures or graphics cards. We showed that depending on the existing workflow and the application domain one of these frameworks can support parallelization of applications sufficiently.

References

1. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices* 30(8), 207–216 (1995)
2. Kejariwal, A., Nicolau, A., Banerjee, U., Veidenbaum, A., Polychronopoulos, C.: Cache-Aware Iteration Space Partitioning. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. pp. 269–270. Salt Lake City, UT, USA (2008)
3. Leiserson, C.: The Cilk++ concurrency platform. In: *Proceedings of the 46th Annual Design Automation Conference*. pp. 522–527. ACM (2009)
4. McCool, M., Du Toit, S.: *Metaprogramming GPUs with Sh*. AK Peters, Ltd. (2004)
5. Monteyne, M., Inc, R.: *RapidMind Multi-Core Development Platform*. RapidMind, Tech. Rep (2007)
6. Muchnick, S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann (1997)
7. OpenMP Architecture Review Board: *OpenMP Application Program Interface*. OpenMP Architecture Review Board (May 2008)
8. OpenMP Architecture Review Board: *Open Multi-Processing*. <http://openmp.org/> (Oct 2009), visited 23/10/2009
9. Reinders, J.: *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc. (2007)
10. Wolfe, M.: *High Performance Compilers for Parallel Computing*. Addison-Wesley (1996)

A Efficient Histogram Implementations

An efficient implementations of histogram generation is shown here. Compared to the implementations in Sec. 3 where locking mechanisms are used intensively, critical sections are avoided in the implementations belows as far as possible. This allows to parallel histogram generations while achieving decent speedups.

A.1 OpenMP

```
1 int hist_tmp[MAX_INTENSITY];
2 memset(hist_tmp, 0x0, MAX_INTENSITY*sizeof(int));
3
4 #pragma omp parallel default(none) shared(img, hist) private(x,
5     y) firstprivate(height, width, hist_tmp)
6 {
7     /* calculate one histogram per thread */
8     #pragma omp for
9     for (int y=0; y<height; y++) {
10         for (int x=0; x<width; x++) {
11             int img_value = img[x + y*width];
12             hist_tmp[img_value] += 1;
13         }
14     }
15     /* merge the subhistograms */
16     #pragma omp critical
17     for (int i=0; i<MAX_INTENSITY; i++) {
18         hist[i] += hist_tmp[i];
19     }
20 }
```

A.2 Cilk++

```
1 cilk::mutex c_mutex;
2
3 int num_workers = cilk::current_worker_count();
4 int nheight = (int) ceil((float)(height)/(float)num_workers);
5
6 /* use one subhistogram per thread */
7 #pragma cilk_grainsize = 1
8 cilk_for (int k=0; k<num_workers; k++) {
9     int hist_tmp[MAX_INTENSITY];
10    memset(hist_tmp, 0x0, MAX_INTENSITY*sizeof(int));
11
12    /* calculate one histogram per thread */
13    for (int iy=0; iy<nheight; iy++) {
14        int y = iy + k*nheight;
```

```

15     if (y >= height) break;
16     for (int x=0; x<width; x++) {
17         int img_value = img[x + y*width];
18         hist_tmp[img_value] += 1;
19     }
20 }
21
22 /* merge the subhistograms */
23 c_mutex.lock();
24 for (int i=0; i<MAX_INTENSITY; i++) {
25     hist[i] += hist_tmp[i];
26 }
27 c_mutex.unlock();
28 }

```

A.3 Threading Building Blocks

```

1 class CalcHist {
2     int *my_img;
3     int my_width, my_height;
4 public:
5     int *my_hist;
6
7     /* calculate one histogram per thread */
8     void operator()(const blocked_range<int>& r) {
9         int width = my_width;
10        int height = my_height;
11        int *img = my_img;
12
13        for (int y=r.begin(); y!=r.end(); ++y) {
14            for (int x=0; x<width; x++) {
15                int img_value = img[x + y*width];
16                my_hist[img_value] += 1;
17            }
18        }
19    }
20
21    /* merge the subhistograms */
22    void join(const CalcHist& y) {
23        for (int i=0; i<MAX_INTENSITY; i++) {
24            my_hist[i] += y.my_hist[i];
25        }
26        scalable_free(y.my_hist);
27    }
28
29    CalcHist(CalcHist& x, split):
30        my_img(x.my_img),
31        my_width(x.my_width),

```

```

32     my_height(x.my_height)
33     {
34         my_hist = (int *)scalable_malloc(sizeof(int)*
35             MAX_INTENSITY);
36         memset(my_hist, 0x0, sizeof(int)*MAX_INTENSITY);
37     }
38
39     CalcHist(int *img, int width, int height):
40         my_img(img),
41         my_width(width),
42         my_height(height)
43     {
44         my_hist = (int *)malloc(sizeof(int)*MAX_INTENSITY);
45         memset(my_hist, 0x0, sizeof(int)*MAX_INTENSITY);
46     }
47 };
48
49 CalcHist ch(img, width, height);
50 parallel_reduce(blocked_range<int>(0, height),ch,
51     auto_partitioner());
51 hist = ch.my_hist;

```

A.4 RapidMind

Note: This is the same implementation as in Sec. 2.4.

```

1  /* temporary array, storing one histogram per line */
2  Array<1, Value1i> hist_tmp = (MAX_INTENSITY*height);
3  Array<1, ArrayAccessor<1, Value1i> > hist_tmp_accessor = dice(
4      hist_tmp, MAX_INTENSITY);
5  Array<1, Value1i> hist(MAX_INTENSITY);
6  Array<2, Value1i> rm_img(width, height);
7
8  /* calculate one histogram per line */
9  Program rm_hist_diced = BEGIN {
10     In<Value1i> line;
11     Out<Array <1, Value1i> > res(MAX_INTENSITY);
12
13     FOR (Value1i i=0, i<MAX_INTENSITY, i++) {
14         res[i] = 0;
15     } ENDFOR;
16     FOR (Value1i i=0, i<width, i++) {
17         Value1i img_val = img[Value2ui(i, line)];
18         res[img_val] = res[img_val] + 1;
19     } ENDFOR;
20 } END;
21
22 /* reduce histograms into final histogram */

```

```
22 Program rm_hist_reduce = BEGIN {  
23     In<Value1i> pos;  
24     Out<Value1f> res;  
25  
26     res = 0;  
27     FOR (Value1i i=0, i<height, i++) {  
28         res += hist_tmp[pos + i*MAX_INTENSITY];  
29     } ENDFOR;  
30 } END;  
31  
32 hist_tmp_accessor = rm_hist_diced(grid(height));  
33 hist = rm_hist_reduce(grid(MAX_INTENSITY));
```