AN AUTO-TUNING JIT COMPILER FOR ACCELERATING
MULTIPLE STENCIL COMPUTATIONS

Yan Wang, Brian Vinter
Niels Bohr Institute, University of Copenhagen, Denmark

ABSTRACT:

We present a JIT compiler with auto-tuning capabilities fusing multiple stencil computations. Data arrays for scientific computing of image processing often exceed cache-memory size. To take advantage of spatial and temporal locality, a common method is to partition the images into tiling blocks for multicore architectures. In realistic scenarios, the multiple image algorithms, most of which are stencil computations, should be processed consecutively. We fuse those stencil codes on the tiled edge-aware image blocks for efficient parallel computing. In this work, multiple kernels are concatenated and blocking according to the images’ dimension and the cache memory size of current hardware. To avoid high load communication, a sufficient number of halo layers are added. One auto-tuning strategy can keep an optimal balance between block size and the number of halo layers. Compared to hand-optimized naïve schemes, experiment using our task parallel model achieves at least 50% improvement of system performance on three CPUs workstations.

Keywords: Parallel Computing, Auto-tuning, Stencil computations, Cache Optimization, JIT compiler

[1] INTRODUCTION

Stencil codes are a class of iterative kernels, which update array elements according to some fixed patterns called stencils [1]. Stencil computations perform a sequence of sweeps, which are called timesteps, through given arrays. Generally they are 2-dimensional or 3-D regular grids. Most image-processing algorithms are 2-D stencil computations [2]. If the images are not tiled before being processed, the boundary values of the whole images can be left unchanged in most cases. Unfortunately, the images become extremely large recently in scientific fields. Cache memory capacity is not big enough to load the whole images. And therefore the tiling approach is one of the best ways to speed up the parallel performance on multicore architectures.

The boundaries of tiled image blocks may contain important information that could not be ignored. A common solution is to extend the boundaries with neighbor elements called halo region. One layer of inserting padding is enough for one stencil kernel usually. Even in the cases where the application executes on one image with multiple stencil computations consecutively, one layer of padding also works well if the image is processed as a whole. But it does not work for parallel multiple stencil computations. Since the tasks based on tiled blocks, which will execute in parallel on different CPU cores, are unable to accomplish synchronously, the blocks’ results of the same kernel cannot be written back from cache memory to main memory simultaneously. For the above reasons, the parallel tasks should execute fusion codes of multiple stencil kernels on tilled blocks. The non-stencil codes of image processing can also be fused easily. In this work, a library that includes several common image-processing algorithms of 2-D stencil computations is involved in our JIT compiler, such as Gaussian filter, Laplacian operation and so on.
Auto-tuning technology assumes various optimization strategies for source codes, whose parameters are optimization options. In running time, the current auto-tuning mechanism tries to find the best combination of arguments [3]. Auto-tuning technology is widely used in scientific libraries. When installing this kind of scientific libraries, it often traverses all the optional combinations of data sets to find the best arguments’ combination of optimization parameters, which are recorded for future calls. However, it would cost intolerable long time for the practical application to traverse the whole data sets. Besides, if the application includes multiple functional modules, the search domain of the optimization parameters is much larger than the library functions. Even worse, every time when updating the libraries or transferring them onto a new platform, this traversing is needed to run again. Such overhead is unacceptable. To eliminate the above adverse effects, our auto-tuning mechanism is implemented in a dynamic way – three dynamic modules in the JIT compiler. One auto-tuning strategy can find an optimal balance between the block size and the number of halo layers. Moreover, a dynamic detector can reduce the number of halo layers when some kernels can reuse other kernels’ extended edges.

In order to decrease the workload and increase the portability, this JIT compiler includes three modules, which are the module of memory management, the module of compiler optimization approaches and the prediction module of runtime environment. With this JIT compiler, the users can execute multiple stencil computations of image processing in parallel easily. What they need to do is to list the names of image algorithms orderly and indicate the image’s address. The platform’s configurations (cache size and the number of CPU cores), the stencil kernels’ types and the image’s dimensions are the parameters for auto-tuning mechanism. We use the Pthread API for parallelism. Compared to OpenMP, our parallel model is more flexible for implementing the complex thread scheduling. Besides, it will be more portable to across different platforms.

[2] RELATED WORK

There are several auto-tuning frameworks for stencil computations. Kamil et al. present a stencil auto-tuning framework that automatically converts the sequential FORTRAN 95 stencil expression into tuned parallel implementations in Fortran, C, or CUDA, thus allowing performance portability across diverse computer architectures [4]. PATUS [5] is a code generation framework for parallel iterative stencil computations at multi- and many-core processors. As the GPU computation becomes popular, some researches focus on the high performance stencil computations on the GPU architectures [6, 7, 8]. Holewinski et al. present a code generation scheme for stencil computations on GPU accelerator, which optimizes the source codes by trading an increase in the computational workload for a decrease in the required global memory bandwidth [6]. Some scientific computation libraries also can provide auto-tuning kernels [9, 10]. However, those auto-tuning framework do not focus on fusing multiple stencil computations of scientific image processing, especially for dense large-scale images processed in parallel.

It is an expedient method to implement a JIT compiler when people want to present their high-performance optimization or auto-tuning framework as cross-platform to the end
users. Besides, if the optimization options can only be assigned when the software is running, the JIT compiler is a perfect way to realize some optimization methods. Tiwari et al. describe a scalable and general-purpose framework for auto-tuning compiler-generated code [11]. MiSFIT can be used as the central component of a tool set for building a safe extensible system in C++ [12]. MiSFIT transforms C++ source code, compiled by g++, into safe binary code. Datta’s dissertation [3] presents a cache-based multicore platform of auto-tuning stencil codes. However, this work is not easy to reproduce for the persons who have not professional programming skills. In brief, the stencil computation for image processing is still needed to improve on multicore platforms. For the researchers, who haven’t enough professorial programming skills, multiple stencil computation of the image processing, especially on dense large-scale images, a convenient and efficient compiler will be useful.

[3] AUTO-TUNING JIT COMPILER

The architecture of this JIT compiler is illustrated in [Figure-1]. The optimization parameters of auto-tuning mechanism include the number of CPU cores for every node if the platform is cluster architecture, the last-level cache memory size and the image dimensions, i.e. the length and the height of the image. Depending on the above information, the extremely large images can be partitioned into tiled blocks of the suitable size. It can take the advantage of the spatial and temporal locality when the blocks’ size is in an appropriate range. The compiler executes three main duties, i.e. the memory management, the compiler optimization and the runtime environment prediction. There are three modules to realize those duties. When the users put in the list of image algorithms orderly, the module of kernels generator will fuse all image algorithms into one function. It is implemented as a C++ template. The edge-aware stencil codes are marked for extending the halo region in the template. The non-stencil codes are also fused without halo region. The algorithms’ order is identified in a FIFO (First In First Out) structure. The module of the running environment prediction can detect the hardware configurations to archive the size of the last level cache memory and the number of CPU cores. It can also read the image files’ file head to find the dimensions of the images.

If the images are larger than the cache size, this module will partition them into blocks of appropriate size according to the hardware configurations. We try to keep balance between the number of CPU cores and the cache memory size for all the calculating of image...
processing. We can guarantee that the number of CPU cores is at least 2 and the blocks size is larger than the last memory size. This module’s output is a tasks’ list. Each item in this list includes the location information for one tiled block. With this information one block can be processed individually. The module of tasks scheduler allocates the parallel threads for the tasks’ list and keeps the load balance according to the number of CPU cores and the number of finished tasks in this list.

The JIT compiler is implemented by C++ programming language and is compiled by g++ on Linux operating system for CPU architectures.

**[3.1] MEMORY MANAGEMENT**

For stencil computations, most elements need to be updated according to some fixed patterns and the old values of neighbor elements. To keeping the useful information of edge-aware image, the halo region should be extended outside of the original image, shown in [Figure-2].

Cache blocking technology can improve the system performance of stencil computations [13]. However, to update one element, the neighbor elements will be reloaded, if the image’s dimension is larger than cache memory size. It will give rise to cache misses. To improve the spatial and temporal locality, we want to partition the extremely large images into blocks of appropriate size. Because multiple threads running on one processor can read the same image block data from cache memory. An easy strategy is to fuse the stencil kernels that sweep on the same tiled blocks, illustrated in [Figure-3]. In this work, we only consider the stencil computations that sweep in the same direction. If there are stencil computations that sweep in the opposite direction, it is too difficult to fuse those stencil kernels in one function.

Figure: 2. Image extended with halo region

Now we face up to one problem that is how to synchronize the output values after each stencil computation of tiled-block images, if there are multiple stencil computations for one application, especially for edge-aware image algorithms. To solve this problem, we adopt multiple layers of halo regions for multiple stencil computations, illustrated in [Figure-4]. The tasks based on tiled image blocks can be scheduled randomly and don’t need to synchronize the edge elements. The halo layers are not unlimited. One auto-tuning strategy can find an optimal balance between the block size and the number of halo layers.
The way to partition the images will impact the workload of memory access. That is, if the images are partitioned into small tiled blocks, the halo regions will increase and therefore the data transmitting between cache memory and main memory will rise. Besides, the stencil computations on the halo regions will also increase. Considering to the locality, the block dimensions can’t be too large. As we have already discussed before, to reduce the data transmitting of stencil computations, the block dimensions can’t be too small. With the prediction of the running environment, our auto-tuning optimization strategies can keep a balance between workload and locality very well. To demonstrate our auto-tuning optimization strategies, there is a 6×6 image partitioned in two ways, as shown in [Figure-5]. To operate two stencil computations on this image, we insert two layers of halo regions outside of each block. 100 times of stencil computations for the original image are needed if it is not partitioned. By the No.1 partition way, the original image becomes nine 2×2-tiled image blocks. The whole stencil computations are 180 times. By the No. 2 way, only 136 times of stencil computations are needed for four 3×3 tiled image blocks. Suppose that the last level cache memory is large enough for one 3×3 tiled image block. The No. 2 partition method is better than the No. 1 method for the optimization of locality.
[3.3] RUNTIME ENVIRONMENT

Auto-tuning techniques approach the best combination of optimization options. Those techniques provide the best optimization source codes that can achieve the high system performance. However, the traditional auto-tuning method has a disadvantage that the search domain is huge in many scenarios. To decline the searching time consumption, we provide an accurate performance model to reduce the search domain. A combination list of the image dimensions and the hardware configurations (cache memory size and the number of CPU cores) is prepared in a lookup table before the image applications running. The module of the runtime environment prediction in the JIT compiler makes decisions with the help of this lookup table.

A transparent C++ interface is illustrated in [Figure-6]. It should note that the program’s details are not shown here. Each function of kernel_n() realizes one image algorithm, which can be stencil computations or non-stencil computations. When the kernels are called by main function, the corresponding kernels are recorded in the vector of fused kernels. Using auto-tuning strategies, the function of task_creator() partitions the large-scale image into blocks with appropriate size.

```cpp
1. class ImageProcessing {
2. public:
3.   ImageProcessing(int kernelNum);
4.   virtual ~ImageProcessing();
5.   void kernel_0(int orderNum);
6.   void kernel_1(int orderNum);
7.   void kernel_2(int orderNum);
8.   int task_creator(int width, int height, int cacheSize);
9.   int kernel_fusion(void);
10.  void scheduler(void);
11. private:
12.   uint8_t * imageInfo;
13.   uint8_t * taskGroup;
14.   int haloLayer;
15.   int kernelNum;
16.   std::vector<IMAGEKERNEL> scheduleVector;};
```

Figure: 6. Image Processing C++ Class

[4] EXPERIMENTS AND RESULTS

The image-processing algorithm of edge detection is demonstrated in this work. To explain our auto-tuning schedule, it is better to choose multiple stencil computations as demos. LOG (Laplacian of Gaussian) is a common image algorithm to detect the images’ edges.
Laplacian filter is a derivative filter used to find the areas of rapid change (edges). Because the derivative filters are sensitive to noise, it is common practice to smooth the images (e.g., using a Gaussian filter) before applying Laplacian operations. This two-step process is referred as Laplacian of Gaussian (LoG) [14, 15]. Finally a zero-crossing edge detector is used to generate edges. In the experiments, there are three stencil kernels, i.e. Gaussian Filter, Laplacian filter and zero-crossing operations.

The experiments are measured on three different multicore platforms. The resource constraining of those machines are illustrated in Table 1. The first machine called platform A is laptop, Macbook Pro with Intel Core 2 Duo P8600. The second machine (platform B) is Server with Intel XEON® processors E5640. The last machine (platform C) is an AMD OPTERONTM processor 6272. The input RGB image is 104×104 pixels. The stencil patterns’ precision is float.

<table>
<thead>
<tr>
<th>Platform</th>
<th>CPU Info</th>
<th>L1 Cache Size</th>
<th>L2 Cache Size</th>
<th>L3 Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2.4, 2</td>
<td>2x64KB</td>
<td>3MB</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2.6, 8</td>
<td>4x64KB</td>
<td>4x256KB</td>
<td>12MB</td>
</tr>
<tr>
<td>C</td>
<td>2.1, 32</td>
<td>16x48KB</td>
<td>16x1MB</td>
<td>16MB/Socket</td>
</tr>
</tbody>
</table>

* — the CPU information include the CPU frequency (GHz) and the CPU cores.

Table: 1. Resource Constraints of Platform A/B/C

For the application of edge detection, we test a same image on three platforms of different CPU cores’ number. It is called strong scaling, which is a common method for practical scientific applications. The parallel execution by our JIT compiler shows markedly speedup ratios, compared to hand-optimized naïve schemes, experiment using our task parallel model achieves at least 50% improvement of system performance on three CPUs workstations. This JIT compiler for large-scale dense image processing is more effective than the naïve computations accelerated by OpenMP. To verify this compiler, we run it on one scientific image, a microscope picture of cells as shown in [Figure-7](a)[16]. This protein image becomes as [Figure-7](b), after being blurred by Gaussian filter. The edges of protein are created as shown in [Figure-7](c) after the operations of Laplacian filter and zero-crossing operations.

![Figure 7. All the Steps of Edge Detection on the Picture of Microscope Cells](image-url)
[5] CONCLUSIONS

This auto-tuning JIT compiler can accelerate most images processing of 2D iterative stencil computations, especially for the dense large-scale images. There is no limitation for the image size. The throughout depends on the computers configurations. We believe that this method can also be deployed for the stencil computations in other scientific fields. The parallel executions show markedly speedup ratios on three different platforms, compared to naïve executions. Furthermore, it is more effective than the program accelerated by OpenMP. The fusing method for multiple stencil computations speedups system performance successfully. With the prediction of running environment, our auto-tuning optimization strategies can keep a balance between workload and locality efficiently. Our auto-tuning JIT compiler handles all the optimization work for the users who can ignore most programming details. It should be noted that the method could only achieve high performance for the stencil computations on dense 2D/3D data. For the sparse grid data, recursively traversing on a quad-tree data structures and pruning the blank sub-tree are more effective for the large-scale blocks [17].

ACKNOWLEDGMENTS
This work is funded by grant # 09-067060 from the Danish Council for Strategic Research.
REFERENCES


