Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 2126–2135

Procedia Computer Science

iWAPT2011
International Conference on Computational Science, ICCS 2011

# Autotuning in an Array Processing Language using High-level Program Transformations

Yusuke Shirota, Jun'ichi Segawa, Masaya Tarui, Tatsunori Kanai

*Corporate Research & Development Center, Toshiba Corporation, 1, Komukai-Toshiba-cho, Saiwai-ku, Kawasaki, 212-8582, JAPAN*

**Abstract**

An autotuning framework based on an algorithm description language dedicated to array processing is introduced. The array processing language allows algorithm developers, may not be equipped with non-trivial knowledge of the increasingly complex architecture of today's processors, to easily perform extensive platform-specific tuning to fully extract performance. A given array processing program is translated into candidate parallel C codes, the best of which can then be selected by empirical evaluation. The high-level abstraction nature of our language allows a unique array processing program to be exposed to wide range of high-level program transformations, thus raising chances of obtaining high performance code. Furthermore, it also enables to extract algorithm-level information which can then be used in heuristic methods for efficient optimization parameter space exploration in the autotuning process. The results of preliminary evaluations show that autotuning using the parameterized C code variants generated with high-level program transformation is useful.

Keywords: Autotuning, program transformation, MATLAB, high productivity, SIMD, parallel programming, high-level programming

## 1. Introduction

Current processors both for high performance and embedded computing are offering more and more parallelism by adding cores and SIMD instructions, and thus not utilizing them means a big loss in performance. Adapting to the memory hierarchy of the target processors is also a key for performance. Therefore, the autotuning concept[1-6] using empirical methods to enable extensive platform-specific tuning to fully extract performance is promising.

Productivity of parallel programs is also increasingly becoming an important issue to cope with the increasing complexity of the underlying architecture. Traditional parallel programming methods with low level languages such as C, C++, and assembly level code compel programmers to be equipped with non-trivial knowledge of both the application’s algorithm and the underlying architecture to write efficient programs. In addition, achieving high performance ends up being highly tuned for a specific architecture and the portability is lost.

To improve software development productivity, new languages, such as X10[7] and XcalableMP[8], that provide abstraction of the underlying architecture are emerging. With the growing complexity and diversity of parallel architecture, it is thus important that algorithm developers, using popular languages which provide high-level of abstraction, such as MATLAB[9] and Octave[10], may not be equipped with non-trivial knowledge of these architectural features, can easily and deterministically access them; it is best if such high-level descriptions are compiled into efficient parallel codes with autotuning. Therefore, autotuning should be performed without any programmer’s knowledge of the underlying parallel architecture.

To address these issues and accelerate parallel program development, we designed an algorithm-level language

1877–0509 © 2011 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.
Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.233

where we further raised the level of description to algorithm-level[11,12]. Our language is a domain-specific language for array processing suitable for data-parallel applications in image processing and signal processing. Based on this language, an autotuning framework is built.

The key advantage of our autotuning approach from the programmer's perspective is that they need not be equipped with non-trivial knowledge of the complex architectural features. In our approach, a given program written with our language is translated to various C code implementations that are multithreaded and vectorized for each architecture by our translator. Generated C code variants have tunable parameters of which are tuned by experiments in the following empirical evaluation phase; the best tuned implementation is then selected. Thus programmers can concentrate on their algorithms.

The two key advantages of our autotuning approach, from efficiency point of view, include:

- ***Deterministic high-level program transformations:*** our language design allows the translator to perform deterministic high-level program transformations of different kinds. Program transformations at the lower level, such as in C, require complex analysis such as data dependency analysis, and are often limited. On the other hand, our language is carefully designed to allow algorithm developers to naturally express their algorithms, while enabling extraction from the parameters of array-oriented operators information, such as how element arrays overlap with one another in nested arrays, useful for program transformations. In this way, the translator can very easily extract and convert algorithm developer's knowledge to optimization information, thus enabling deterministic high-level program transformations. Therefore, a unique array processing program can be exposed to wide range of program transformations, thus expanding the search space and raising chances of obtaining high performance code. We mainly focus on this part in this paper.

- ***Efficient optimization parameter space exploration:*** with the extracted information the translator can use heuristic methods to reduce the size of exploration space. With our language, the amount of data accessed while processing a certain size of a result array can be calculated utilizing information regarding adjacency properties of overlapping arrays; this enables blocking-like heuristic method for exploiting cache locality to determine a reasonable size for each block. Our heuristic method uses a conservative estimate, to avoid exceeding the cache size; thus, for example, the exploration can start from the estimated value, and only higher values can be explored until the performance starts to drop, pruning exploration on the lower-value side.

In addition to these advantages, deterministic vectorization realized by our vectorizer is also important in our approach since utilization of parallelism via SIMD instructions is the key to high performance. Our vectorizer emits SIMD codes using C intrinsics provided by C compilers. An alternative way is to use auto-vectorization[13] features of the C compilers[12]. However, performing vectorization directly from the language in SIMD fashion using our vectorizer is less reliant on complex analysis of C compilers and therefore is deterministic[14]. As for autotuning, vectorization decisions – for example, whether to perform AoS to SoA conversion or not, are important autotuning parameters, thus having control over them is important.

Important features added to the array processing language include seamless integration with MATLAB. For MATLAB users, MATLAB description format of our language is provided. This feature realizes programming with the familiar MATLAB language, contributing to mitigating the cost of learning a whole new language. This feature allows MATLAB users to seamlessly use our autotuning framework from MATLAB.

The remainder of this paper is organized as follows. Section 2 provides brief introduction on the array processing language. Section 3 describes our autotuning approach using program transformations. Section 4 gives the preliminary evaluation results on high performance Intel® Core™2 Duo[a] and power efficient Freescale i.MX515 processor. Related work is discussed in Section 5, and finally, conclusion is given in Section 6.

## 2. The Array Processing Language

In this section, we describe some basic elements of our array processing language. Integration with MATLAB is

[a] Intel, Intel Core are trademarks of Intel Corporation in the U.S. and/or other countries.

also briefly introduced. To keep the paper short, only MATLAB description format of our language will be used throughout this paper. Mathematical-looking representations of our language are presented in [11].

## *2.1. The Array Processing Language Basics*

Our array processing language features a rich set of operators and function application methods which act on arrays, contributing to loop-less expressions.

Array operators such as standard arithmetic operators are defined as element-wise operators, and thus return the results of applying an element-wise operation on the corresponding array elements; for example, if a two-dimensional, 2 x 3 array A = [ 0, 1, 2; 3, 4, 5] , B = [6, 7, 8; 0, 9, 10], then A .* B = [0, 7, 16; 0, 36, 50]. Here, ".*" represents element-wise multiplication. If one of the operands is a scalar value, it is promoted so that the scalar value is duplicated in every element of the promoted array.

Aggregation operators calculate a scalar value from an operand array. The array processing language provides, for example, *Sum* (the sum of elements in a given array), *Max* (the maximum value of a given array), *Min* (the minimum value of a given array). Names of functions provided by our language begin with capital letters.

Using these basic array operators, simple array operations can be expressed without using loops. However, they are not powerful enough to express array operations of such are written with deeply nested loops in C. For this reason our array processing language provides two special operators: *Extract* and *MExtract* for extracting sub-arrays, and a *Map* function application method.

An *Extract* operator extracts a single sub-array from a given array as depicted in Fig 1. A *base* parameter represents the initial position of extraction; in Fig 1-(A), the *base* is [1, 2]. A *size* parameter represents the size of the extracted sub-array. If the extracted sub-array is not within a boundary of a given array, values in the elements outside the boundary are treated as "null" as shown in Fig 1-(B). The "null" value simplifies the exceptional handling with pixels on the edges of an image data in image processing applications.

$N$ = ***Extract***$(M, base, size)$

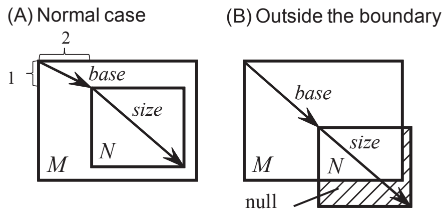

Fig. 1. Single sub-array extraction operator.

$N$ = ***MExtract*** $(M, base, step, size, esize)$

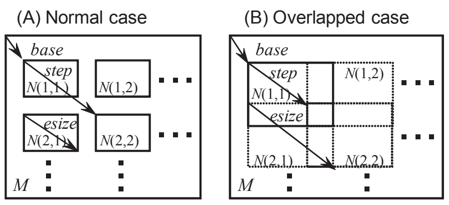

Fig. 2. Multiple sub-arrays extraction operator.

***Map***(@*f*, ***MExtract***(*M, base, step, size, esize*)) =

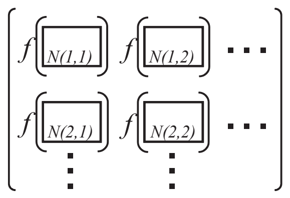

Fig. 3. *Map* function application method.

An *MExtract* operator extracts multiple sub-arrays as shown in Fig 2. The *base* parameter represents the initial position of extraction of the upper-left sub-array, *step* parameter represents the stride of extraction, *size* represents the number of sub-arrays to be extracted, and *esize* represents the size of each sub-array. If *step* is smaller than *esize*, the extracted sub-arrays overlap with neighbor sub-arrays as shown in Fig 2-(B). Here, the upper-left sub-array is indexed N(1,1), following the rule that origins of MATLAB arrays are 1, not 0.

A *Map* function application method applies a function, accepted in the first argument, to each element of the operand arrays in the rest of the arguments and returns packed results in a single array as depicted in Fig 3. Here, @ is a MALTAB operator to create a *function handle*, used in referencing a function, placed before the function name.

If the *Map* function application method is used in combination with the *MExtract* operator, complex array operations of such written with deeply nested loops in C, can be expressed without using loops, thus allowing concise programs.

Besides *Map* function application method, our language supports other function application methods: *Inner* and *Scan*. The details of these methods are explained in Sections 2.3 and 3.2, respectively.

## 2.2. Using the Array Processing Language from MATLAB

MATLAB is a widely used software including high-level MATLAB language. For MATLAB users, we provide ways to integrate our array processing language with MATLAB. It is realized via MEX (short for Matlab EXecutable)[9]. By default, our translator translates a program written with our array processing language in the MATLAB description format into an autotuned C code, but alternative output is a MEX-file, which is a dynamically linked subroutines generated from C code, and in short is an interface to call C routines directly from MATLAB. Thus for example, by translating our Prewitt filter[15] program into a MEX file, speedup of the execution of the interpreted language is realized with an extensively autotuned kernel.

```
img = imread(“testimage.jpg”);    % MATLAB function imread reads in image
result = prewitt(img);            % user-defined Prewitt function written with our array processing language
```

In this way our array processing language complements MATLAB. The details for writing the Prewitt filter program with our language are discussed in the following section.

## 2.3. Array Processing Program Examples

Three programs written with our array processing language in the MATLAB description format are shown.

The first example is the aforementioned Prewitt filter program depicted in Fig 4. Prewitt filter is a well-known neighboring operation in image processing and is used for edge detection. In neighboring operation, the same operation is repeatedly applied across the input image. This data-parallel nature is expressed concisely with the *MExtract* operator and *Map* function application method.

In programs written with our language, a top level function starts with *Type* functions which are used to specify default types of input/output variables. Since by default MATLAB treats all values as double-precision floating point, this contributes to generating optimized C codes, especially with vectorization.

In Prewitt filter, two filters *v* and *h* are applied with the 8-neighbors around each pixel. The program uses an *MExtract* operator to extract sub-arrays of 8-neighbors around each pixel of the input picture *M*, invoking function *f* with *Map* function application method. Function *f* calculates element-wise calculations to each sub-array, and returns a scalar value. Here, *Size(M)* is our built-in utility function which returns the size of a given array *M*, which is 1080x1920, or [1080, 1920] in MATLAB. While it requires quite a few lines of code in C, it is only several lines and without loops with our language.

The second example is a CFAR program depicted in Fig 5. CFAR(Constant False Alarm Rate) is a well-known algorithm in radar systems and is used to detect returning echos from the targets against clutter, or undesired echos.

In CFAR, calculation is applied with the neighboring cells: *n* reference cells on both sides each cell. This program is also expressed concisely with an *MExtract* operator and a *Map* function application method. The program uses an *MExtract* operator to extract sub-arrays of 16-neighbors on both sides of each cell of the input one-dimensional array *M*, with a guarded cell on both sides in between. Then function *f* is invoked with *Map*. Here again, MATLAB function *single* is used to specify the single-precision floating point type to generate efficient C codes.

The third example is a matrix multiplication program depicted in Fig 6. This example uses *inner* function

```
function img = prewitt(M)
   Type({[1080, 1920], 'uint8'}, M);                                   % input image type
   Type({[1080, 1920], 'uint8'});                                      % output image type
   base = [-1, -1];  step = [1, 1];  size = Size(M);  esize = [3, 3];

   function ret = f(m)
      v = [ 1,   1,   1;
            0,   0,   0;
           -1,  -1,  -1];
      h =  [1,   0,  -1;
            1,   0,  -1;
            1,   0,  -1];
     p = abs(Sum(h .* m)) / 2 + abs(Sum(v .* m)) / 2;
     if p > 255                                                        % saturated at largest possible value
       ret = 255;
     else
       ret = p;
     end
   end

   img = Map(@f, MExtract(M, base, step, size, esize));
end
```

Fig. 4. Prewitt filter program.

```
function ret = cfar(M)
   Type({[2048], 'single'}, M);
   Type({[2048], 'single'});
   n = 16;

   function r = f(p, near, far)
      d = Sum(near) + Sum(far);
      r = single(10.0) * log10(single(32.0) * p / d);
   end

   ret = Map(@f, M,
      MExtract(M, [-(n + 1)], [1], Size(M), [n]),
      MExtract(M, [2], [1], Size(M), [n]));
end
```

Fig. 5. CFAR program.

```
function ret = mm(M, N)
   Type({[1024, 1024], 'double'}, M);
   Type({[1024, 1024], 'double'}, N);
   Type({[1024, 1024], 'double'});
   ret = Inner(@Plus, @Multi, M, N);
end
```

Fig. 6. Matrix multiplication program.

application method. *Inner*(*@f1*, *@f2*, *M, N*), where the size of *M* = [m, k] and size of *N* = [k, n], returns an array of size [m, n] whose element at row i and column j is calculated in two steps; first, function *f2* is applied to i-th row of *M* and j-th column of *N*, and function *f1* is applied to the resulting vector. Here, the *Plus* and *Multi* are built-in functions for addition and multiplication respectively, and thus matrix multiplication is simply expressed.

## 3. Autotuning using High-level Program Transformations

This section contains an overview of autotuning using program transformations in the array processing language.

### *3.1. Using Extracted Parameters for Deterministic Program Transformations and Parameter Space Exploration*

When the translator generates C codes from array processing programs, the translator must decide which set of program transformations to apply. Instead of dealing with complex tradeoffs, the translator generates code variants with different set of transformations applied. The generated C codes are parameterized; it is autotuned by experiments using naive search exploring the whole parameter space.

However, algorithm-level information specified in the parameters of array operators, can be used to reduce the size of parameter space. With our *MExtract* operator, as is illustrated in Fig 2, it is trivial how the extracted sub-arrays overlap with one another. These adjacency properties of overlapping arrays can be used for deciding block sizes in blocking-like optimization. Algorithm-level information also includes dependency information derived from the *Map* function application method. The *Map* guarantees that there are no data dependencies between function calls since our language is a functional language, and that function calls can be executed in parallel. The information realizes deterministic program transformations, deterministic vectorization, multithreading, and efficient parameter space exploration in the autotuning process. Further details are shown using concrete examples in Section 3.2.

Program transformation is performed by the translator using idiom recognition. Therefore in our framework, autotuning can be enhanced by adding new transformation rules. Further information on idiom recognition is provided in [12].

### *3.2. High-level Program Transformation Examples*

Four examples of high-level program transformations are introduced in this section. Using these program transformations, a unique array processing program can be exposed to wide range of optimization, thus raising chances of obtaining high performance code.

The first transformation, an *MExtract2Extract*[12], maps an algorithm using intuitive *MExtract* to an algorithm using multiple SIMD-friendly *Extract* operations. When it comes to vectorizing the neighboring operation in Fig 4, without this program transformation the vectorizer fails to vectorize each function *f*, lucking parallelism for vectorization. Detecting the combination of an *MExtract* and a *Map* function application method, the translator transforms an assignment to *p* in a program in Fig 4 into a program depicted in Fig 7, using twelve *Extract*s. Each *Extract* corresponds to each of the two filter's non-zero coefficients. *MExtract2Extract* enables to exploit parallelism amongst neighboring operations, generating loops with high trip counts and consecutive memory accesses that are amendable to the vectorizer. The transformation is deterministic for two reasons: 1) the translator can easily guarantee this transformation since the *Map* guarantees that there are no dependencies between the function calls, 2) the translator can easily analyze that pixels that correspond to each coefficient consist an array, since the stride between the adjacent sub-arrays are explicitly specified as [1, 1] in the *step* parameter of the *MExtract* operator, thus possible to express with an *Extract*. Exceptional handlings with pixels on the edges of an image data are also easily taken care of by the transformation.

The second transformation, an *MExtract2Scan*, transforms an *MExtract* operation to a *Scan* operation. It is performed in order to reduce the calculation cost by reusing calculation results. In the CFAR program depicted in Fig 5 for example, *Sum* is applied to each sub-arrays of 16-neighbors extracted with *MExtract*.

However, amongst the summations, there are re-computations. To reuse the result of previous summation, the combination of a *Map* and summation is transformed to a program shown in Fig 8 using *Scan* function application method. Let $M = [M_1, M_2, ..]$ and $N=[N_1, N_2, ..]$ be one-dimensional array of the same size, *Scan*(*@f*, init, *M*, *N*) returns

```
P =  (abs(Extract(M, [-1, -1], Size(M)) + Extract(M, [-1, 0], Size(M)) + Extract(M, [-1, 1], Size(M)) -
      Extract(M, [1, -1], Size(M)) - Extract(M, [1, 0], Size(M)) - Extract(M, [1, 1], Size(M))) ./ 2)  +
     (abs(Extract(M, [-1, -1], Size(M)) + Extract(M, [0, -1], Size(M)) + Extract(M, [1, -1], Size(M)) -
      Extract(M, [-1, 1], Size(M)) - Extract(M, [0, 1], Size(M)) - Extract(M, [1, 1], Size(M))) ./ 2);
```

Fig. 7. Prewitt program transformed with *MExtract2Extract*.

an array of size equivalent to that of *M* and *N*, whose first element is *f*(init, $M_1, N_1$), and the second element is *f*(*f*(init, $M_1, N_1$), $M_2, N_2$), and so on. Thus the program depicted in Fig 8, first calculates value *init*, and uses it as the initial value of *Scan*. Function *g* repeatedly adds value of a new cell and subtracts a value of an old cell. Here, the *Overlay* function specifies that the value in the second argument "0" substitutes for "null" induced by the *Extract* operator. *Overlay* function adds flexibility to program transformation.

The CFAR program, however, is a neighboring operation and the translator has an option to perform *MExtract2Extract* as well. The transformed program introduces an intermediate array *D* with *Scan*, adding tradeoffs between the two transformations.

```
function r = f(p, d)
   r = single(10.0) * log10(single(32.0) * p / d);
end

function r = g(init, near_old, near_new, far_old, far_new)
   r = init - near_old + near_new - far_old + far_new;
end

init =  Sum(Overlay(Extract(M, [-(n + 2)],[n]),0)) + Sum(Extract(M, [1], [n]));
D = Scan(@g, init,
           Overlay(Extract(M, [-(n + 2)], Size(M)),0),  Overlay(Extract(M, [-2], Size(M)),0),
           Extract(M, [1], Size(M)),  Extract(M, [1+n], Size(M)));

ret = Map(@f, M, D);
```

Fig. 8. CFAR program transformed with *MExtract2Scan*.

The third transformation, range scheduling[11], is performed to adapt to the memory hierarchy of the target processor. In range scheduling, the top level range is divided recursively into smaller ranges, and the execution order between them is determined. Top level range is defined as the result array of the top level expression in the array processing program. With a multicore processor with private L1 cache, the simplest example of range scheduling is performed in two steps: first, the range is divided horizontally between each core, then the divided range is further divided vertically by the width of which the cache reuse is enhanced. The width can be left as a parameter in the C code, and can be tuned by empirical evaluation.

The adequate width can be calculated thus eliminating some parameter explorations. The near-optimal width is calculated with blocking-like heuristic method in two steps. First, the *working set* of the range is calculated. The working set is the set of sub-arrays of input and output variables accessed during the execution of the range. With a conservative assumption that each sub-array is assigned to distinct ways of the L1 cache, the width of the range is maximized to the extent of that the size of its working set do not exceed the single way size.

The working set of the range is easily calculated by the translator using algorithm-level information. Consider Prewitt filter program in Fig 4. Let us calculate the working set of the range of which the size is [1, c]. The working set of the input variable *M* can be calculated as *esize* + *step* .* [1-1, c-1], which is equivalent to [3, 3 + (c – 1)]. The working set of the output variable is [1, c]. Since the amount of memory required to contain the working set of size [r, c] can be calculated as r .* c .* sizeof(uint8), the memory required for input and output variables are calculated as 3c + 6 and c, respectively. Therefore, the near-optimal range width is determined by the input array, which is the largest of all. Assuming that a single way of the L1 data cache is 4KB, the near-optimal range width is the largest c which satisfies 3c + 6 <= 4KB (such that c <= 1920), and is normalized to 960, half the width of the input picture.

In the empirical evaluation phase, the values higher than the value determined by the heuristic method can be experimented to mitigate parameter space exploration.

Furthermore, with range pool scheme[11], ranges are executed dynamically from a pool of ranges to address load imbalance, in such cases where the computation load of each range differs from the others.

The fourth transformation is the hierarchical execution for memory access optimization in the *Inner* function application method. Matrix multiplication program depicted in Fig 6 can be transformed into a program depicted in Fig 9. This transformation is equivalent to the common blocking. Input arrays *M* and *N* are divided into sub-arrays of size [a, b] and [b, a], respectively. Then *Inner* function application method is applied with functions *Plus* and *submm*. The *submm* function multiplies two corresponding sub-arrays and the *Plus* adds them up. *Inner* function returns an array of sub-arrays and thus the *Join* function is used to flatten it. Each element of the resulting array of the top-level *Inner* is executed in parallel.

Although by no means are we claiming that the performance of this particular example is comparable to the extensively tuned libraries, this is a good example showing that function application methods offered by our language contribute to deterministic program transformations and creating tunable parameters that can be tuned with autotuning. Even blocking and multithreading of matrix multiplication can be realized within a general framework.

```
a = 16; b=32;
function r = submm(m, n)
   r = Inner(@Plus, @Multi, m, n);
end
ret = Join(Inner(@Plus, @submm,
                    MExtract(M, [0, 0], [a, b], Size(M)./[a, b], [a, b]),
                    MExtract(N, [0, 0], [b, a], Size(N)./[b, a], [b, a])));
```

Fig. 9. Matrix multiplication program with blocking.

## 4. Evaluation

This section shows how the three programs can be tuned with our autotuning approach.
First, Prewitt filter program was evaluated on a 3.00 GHz Intel Core 2 Duo processor with 32KB, 8-way set associative L1 Data Cache and 6MB shared L2 Cache, using GCC ver. 4.4.3. After *MExtract2Extract* is performed, range scheduling is applied. Finally, the computation of each range is vectorized in 8-way 16 bit modes. While 4-way 32 bit mode is the naive vectorization for integers assuming integral promotion of the C-language, our vectorizer performs bit-width inference allowing the computation in narrow bit-width to enhance parallelism.

As a result, parameterized Pthreaded C code variants were generated. Compared with the single-threaded *MExtract2Extract* version which was not vectorized, generated only for comparison purpose, two-threaded

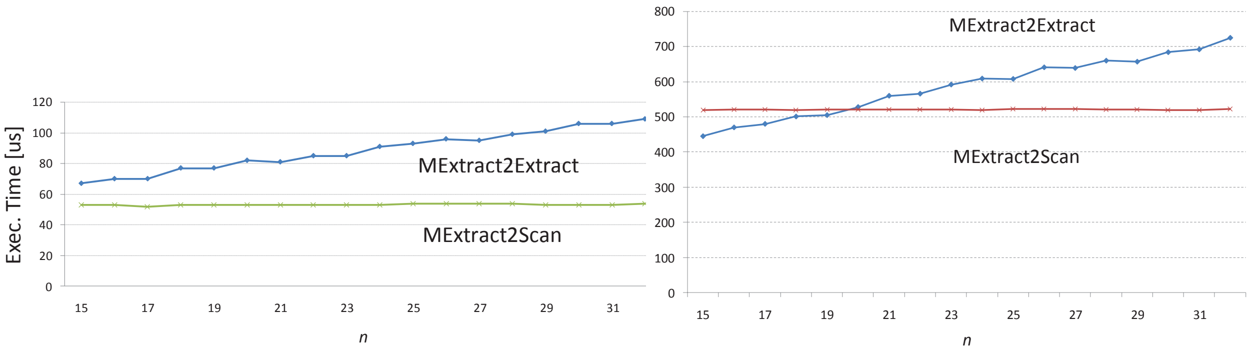

Fig. 10. CFAR program (N=2048) execution result on (a) Intel® Core™2 Duo using SSE; (b) i.MX515 using NEON.

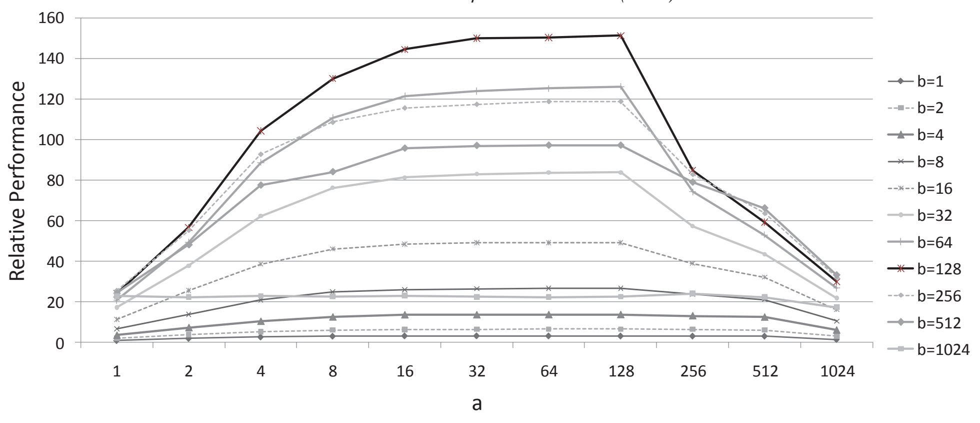

Fig. 11. Matrix multiplication program execution result on Intel® Core™2 Duo.

*MExtract2Extract* version with the near-optimal range size of 960, successfully vectorized using Intel SSE intrinsics showed 9.8x speedup: multithreading and vectorization contributing 1.9x and 4.9x speedups, respectively. Higher values can be experimented in the empirical phase, obtaining 11.9x speedup at the range size of 1080: multithreading and vectorization contributing 1.9x and 6.0x, respectively.

Secondly, the vectorized C codes of CFAR using *MExtract2Extract* and *MExtract2Scan* were evaluated on the same Intel processor. Fig 10-(a) depicts the execution result. The size of variable *n* is varied from 15 to 32. In all cases *MExtract2Scan* versions outperform the *MExtract2Extract* versions, reducing the calculation cost.

Next, we evaluated C code variants on 800MHz i.MX515 processor (ARM Cortex-A8) with 32KB, 4-way L1 Data Cache and 192KB L2 Cache, using GCC 4.3.3. C codes are vectorized using NEON instructions, SIMD instructions for the Cortex-A processors. Fig 10-(b) depicts the result. The size of variable *n* is again varied from 15 to 32. This time the *MExtract2Extract* versions outperform the *MExtract2Scan* versions in some cases. Although calculation cost is reduced in the *MExtract2Scan* version, the effect is small where the size of *n* is small.

Lastly, parameterized and multithreaded C code for matrix multiplication with the hierarchical execution in the *Inner* function application method was evaluated on the Intel processor. The values of both a and b are varied from 1 to 1024. The speedup over a=1 and b = 1 case is shown in Fig 11. With autotuning, the optimal results can be obtained with size a = 128 and b = 128 as depicted in the figure. It is possible to combine simple heuristic methods. With the *Inner* function, non-unit stride access to the second argument array affects performance. Thus, for example, with conservative assumption that square sub-block is assigned to a single way of the L1 cache, the size of it is maximized to the extent that do not exceed a single way size. The estimated block size is calculated as 16.

## 5. Related Work

To improve development productivity, new multicore languages such as X10[7], Sequoia[16], and XcalableMP[8] are emerging. These languages provide abstraction of the underlying architecture, requiring the programmers to divide given problems into sub-problems. By raising the level of description to algorithm-level, our language completely conceals architecture information, and the proper mapping of the problem to the target architecture is performed automatically by the translator.

Intel Array Building Blocks(Intel ArBB)[17] is a well-designed C++ library for data-parallel applications. Although there is still some gap between Intel ArBB descriptions and algorithm descriptions, since Intel ArBB virtual machine can emit codes for the latest Intel CPUs as well as for future Intel architectures, it is a promising candidate for us to use; the translator can target it to support Intel processors and accelerators[18].

MATLAB is a widely accepted software including high-level MATLAB language. Our language is designed to be as succinctly expressive as MATLAB language and other array-oriented languages like APL[19], while enabling powerful optimization to run default on multicores. As it has been mentioned above, MATLAB users can write programs in our language via MATLAB description format, allowing seamless usage of our language from MATLAB environment further enhancing software development in MATLAB.

Real-Time Workshop Embedded Coder[20] is a tool to generate C codes with high readability from MATLAB codes. It is intended primarily for generating C code that runs on any microprocessor and it is not aimed for target specific tuning. Our array processing language complements these tools in terms of extensive platform tuning.

## 6. Conclusion

This paper proposed an autotuning approach using the deterministic high-level program transformations in the array processing language designed for high productivity. Algorithm developers simply write *what they want to do* in our language, and candidate implementations by applying different sets of program transformations are generated; the best implementation is then selected by empirical evaluation. Preliminary evaluation results showed that appropriate program transformation sequence depends on the architectural parameters and application parameters, and that our approach is feasible. Also with our framework, autotuning can be enhanced by adding new transformation rules. However the search is limited to combination of naive search and heuristic methods. This is future work.

The accessibility of autotuning from algorithm-level programs can bridge the gap between algorithm developers such as MATLAB and Octave users and complex parallel processors, which is the most important issue with today's processor architectures.

## References

1. J. Bilmes, K. Asanovic, C. W. Chin, J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. Proceedings of International Conference on Supercomputing, 1997, pp. 340-347.
2. R. C. Whaley, A. Petitet, J. J. Dongarra, Automated empirical optimization of software and the ATLAS project, Parallel Computing, Vol. 27, No.1-2, 2001, pp. 3-35.
3. M. Frigo, A fast fourier transform compiler, Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation, 1999, pp. 169-180.
4. M. Puschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, et al, SPIRAL: Code generation for DSP transforms, Proceedings of the IEEE, Special issue on "Program Generation, Optimization, and Adaptation", Vol. 93, No. 2, 2005, pp. 232-275.
5. T. Katagiri, K. Kise, H. Honda, T.Yuba, ABCLibScript: A directive to support specification of an auto-tuning facility for numerical software, Parallel Computing, Vol. 32, No. 1, 2006, pp. 92-112.
6. K. Naono, K. Teranishi, J. Cavazos, R. Suda. Software Automatic Tuning, Springer, 2010.
7. K. Ebcioglu, V. Saraswat, V. Sarkar, X10: Programming for hierarchical parallelism and non-uniform data access, International Workshop on Language Runtimes, 2004.
8. XcalableMP. Available from: <http://www.xcalablemp.org/>.
9. MATLAB. Available from: <http://www.mathworks.com/products/matlab/>.
10. Octave. Available from: <http://www.gnu.org/software/octave/>.
11. J. Segawa, T. Kanai, The array processing language and the parallel execution method for multicore platforms, The First International Symposium on Information and Computer Elements, 2007, pp. 98-103.
12. Y. Shirota, J. Segawa, Y. Matsui, T. Kanai, Autovectorization-friendly program transformation in the array processing language, Parallel and Distributed Computing and Networks, 2009, pp. 157-162.
13. A. J. C. Bik, The software vectorization handbook, Intel Press, 2004.
14. F. Franchetti, Y. Voronenko, M. Puschel, A rewriting system for the vectorization of signal transforms, High Performance Computing for Computational Science (VECPAR), Vol. 4395, 2006, pp. 363-377.
15. R. C. Gonzalez, R. E. Woods, Digital image processing, Prentice-Hall, 2002.
16. K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, Sequoia: programming the memory hierarchy, Proceedings of the 2006 ACM/IEEE conference on Supercomputing, 2006.
17. Intel Array Building Blocks (Intel ArBB). Available from: <http://software.intel.com/en-us/articles/intel-array-building-blocks/>.
18. M. D. McCool, Data-parallel programming on the Cell BE and the GPU using the RapidMind Development Platform, GSPx Multicore Applications Conference, 2006.
19. K. E. Iverson, A Programming Language, Proceedings of Spring Joint Computer Conference, 1962.
20. Real-Time Workshop Embedded Coder. Available from: <http://www.mathworks.com/products/rtwembedded/>.