High-Level Synthesis for Nested Loop Kernels with Non-Uniform Dependencies

Akihiro Suda, Hideki Takase, Kazuyoshi Takagi, and Naofumi Takagi
Graduate School of Informatics
Kyoto University
Kyoto, Japan 606-8501
suda.akihiro.82s@st.kyoto-u.ac.jp, {takase,ktakagi,takagi}@i.kyoto-u.ac.jp

Abstract— In high-level synthesis, parallelization for nested loop kernels has been hard due to their complex data dependencies, especially non-uniform dependencies. In this paper, we propose a new method to synthesize a parallelized circuit from such kernels using polyhedral optimization, which has been vigorously studied in the software field. The key point of our contribution is a buffering method for parallel RAM accesses. The experimental result shows that the parallelized circuit with 8 PEs is 5.73 times faster than the sequential one.

I. Introduction

In high-level synthesis for nested loop kernels, the execution cycles of synthesized circuits can be reduced by applying an appropriate parallelization method. However, because of data dependencies, it is generally hard to parallelize such kernels by hand work.

Recently, an automatic nested loop parallelization method called *polyhedral optimization* is coming to practical use in the software field. Polyhedral optimization is a state-of-the-art way to generate legally parallelized kernel by analyzing dependencies using several linear algebra calculations. Although there has been some works for applying polyhedral optimization into high-level synthesis[1, 2], these works do not cover kernels with non-uniform data dependencies (i.e. dependencies that cannot be represented by a constant vector).

In this paper, we propose a new method to apply polyhedral optimization into high-level synthesis for kernels with non-uniform data dependencies. Our method is composed of two parts: *threading* and *buffering*. An input sequential C source code is transformed and annotated as parallel with OpenMP directives (`#pragma omp parallel for`) by PLUTO, that is the de facto standard polyhedral source-to-source compiler in the field of software. Our threading realizes synthesis of parallelized circuit from C source code with OpenMP directives. Buffering accelerates the effect of our threading by reducing collisions of access to single off-chip RAM. Although paralleization can be performed independently from buffering, the execution cycles of parallelized kernel in such a way may be much larger due to waiting time for acquisition of rights to access off-chip RAM. Fig. 1 shows the overview of our method. The global controller manages requests from PEs for access to off-chip RAM, and synchronization of PEs. When the buffering method is enabled, each PE handles their own independent buffer so as to reduce accesses to off-chip RAM.

This paper is organized as the following. In Section II, we present a brief look at polyhedral optimization theory and PLUTO with an example. Our threading method and buffering method are presented in Section III and in SectionIV. We discuss an experimental result of proposed method in Section V. We summarize the paper and discuss future works in Section VI.

II. Polyhedral Optimization

A. Overview and History

Polyhedral optimization is the general term for algorithms that performs optimization such as parallelization and locality improvement on nested loop kernels by applying several linear algebra computations.

Polyhedral optimization has been studied since the early 1990s. Then state-of-the-art progresses such as [3, 4] on automatic code generation with polyhedral models produced possibilities for practical software compilation with polyhedral optimization. In GCC, the polyhedral optimization function is called GRAPHITE[5] and officially supported since 2009. In clang, it is called Polly[6] and also officially supported since 2012.

However, polyhedral optimization for high-level synthesis is still under research[1, 2].

B. PLUTO

PLUTO [7, 8] is the de facto standard polyhedral source-to-source C compilation algorithm and its implementation. We adopt PLUTO for its wide applicable scope including non-uniform dependencies. PLUTO transforms a C source code description of what is called SCoP (Static Control Parts) into parallelized C source code with OpenMP directives. SCoP is a nested loop structure of which all loop boundaries, branch condition, and array indices can be represented by affine expressions

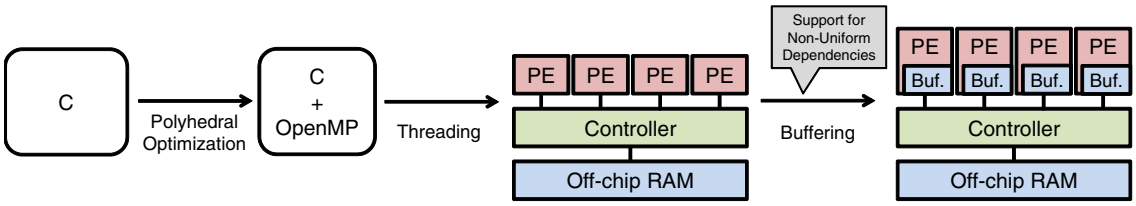

Fig. 1.: Overview of our method

```
for (i = 0; i < N; i++)
  for (j = 1; j < N; j++)
    a[i][j] = a[j][i] + a[i][j-1];
```

Fig. 2.: An example of nested loop kernel with non-uniform dependency

of iteration variables. Therefore, SCoP is also called ACL (Affine Control Loop).

After applying PLUTO, the iteration space of input loop is split into parallelogram-shaped tiles. A tile is an unit for assignment of multiple iteration executions to OpenMP threads, and also for improvement of utilization of CPU caches.

Tiling is done by computation of *space* direction vector and *time* direction vector. The space direction vector is used to assign tiles into threads, and the time direction vector is used to represent execution order of tiles within a thread. For two dimensional nested loop kernel with iteration variable i and j, these vectors are formed as (u_1, w, c_i, c_j). In this vector form, (u_1, w) represents data dependencies between tiles, and (c_i, c_j) represents normal vectors of tile sides. When $u_1 = 0, w = 0$, there is no dependency between tiles. When $u_1 = 0, w > 0$, there is a uniform dependency, i.e., a dependency that can be represented by a constant distance vector of two tiles. When $u_1 > 0, w \geq 0$, there is a non-uniform dependency, which we focus in this paper.

C. Example

Fig. 2 is an example of nested loop kernel with non-uniform dependency, which is focused in several works [8, 9]. Further details of PLUTO and the example kernel are explained in [8].

By applying PLUTO, the tiling vectors of this kernel can be computed as: $(u_1, w, c_i, c_j) = (0, 1, 1, 1)$ for the space direction, $(1, 0, 1, 0)$ for the time direction.

The space direction vector indicates that the tiles are assigned to threads along $(1, 1)$ direction (Fig. 3), and that there is a uniform dependency (Fig. 4) across the space direction. Similarly, the time direction vector indicates that the tiles are executed in $(1, 0)$ order, and that there is a non-uniform dependency across the time direction.

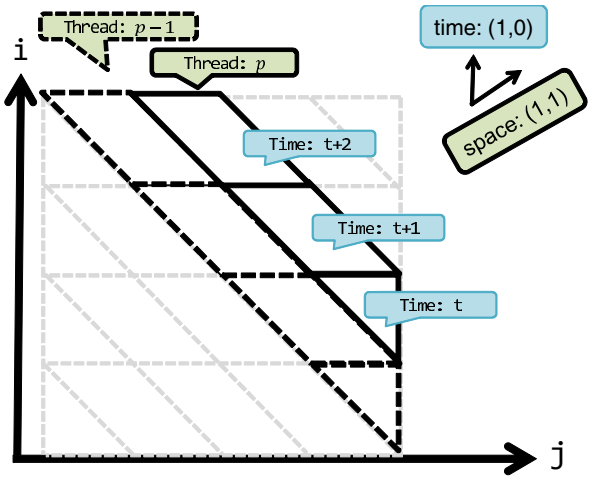

Fig. 3.: Tiling vectors for the example kernel

Using these tiling vectors, PLUTO transforms the original sequential code into the parallelized version. Fig. 5 is the code transformed by PLUTO 0.9.0 with 8×8-sized tiling.

In this transformed code, the loop associated to iteration variable `t2` is executed in parallel. Note that the actual number of threads is usually not `ubp-lbp+1`, but the number is decided according to the environment variable `OMP_NUM_THREADS`.

D. Applying Polyhedral Optimization to High-Level synthesis

Although there is an existing work[10] to transform a C source code with OpenMP directives into a description for high-level synthesis, it cannot be applied to codes generated by PLUTO. This is because the generated codes has non-constant iteration domain. In PLUTO-generated codes, the lower bound of parallelized iteration variable (such as `t2` in Fig. 5) may vary along with an outer iteration variable (such as `t1`). In addition, the number of parallelized iteration times (`ubp-lbp+1`) also may varies. The existing method cannot be applied when the lower bound of parallelized iteration variable is not zero and when the number of iteration times is less than the num-

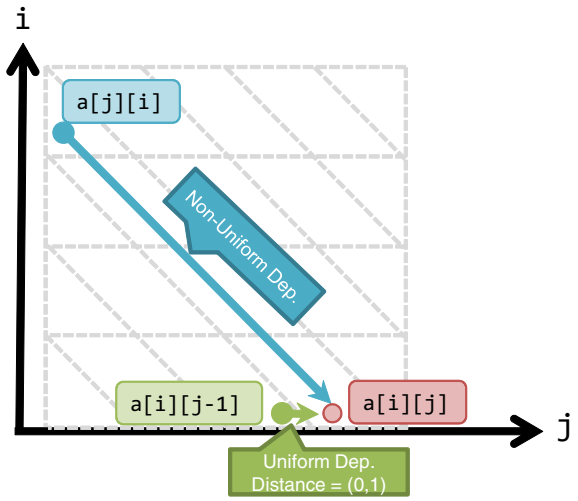

Fig. 4.: Dependencies of the example kernel

```
#define ceild(n,d)  ceil(((double)(n))/((double)(d)))
#define floord(n,d) floor(((double)(n))/((double)(d)))
for (t1 = 0; t1 <= floord(3*N-3, 8); t1++){
  lbp = max(max(0,ceild(4*t1-N+1,4)),ceild(8*t1-N-6,16));
  ubp = min(floord(t1,2),floord(N-1,8));
#pragma omp parallel for \
  private(t2, t3, t4, lbt3, ubt3, lbt4, ubt4) \
  shared(a, t1, lbp, ubp)
  for (t2 = lbp; t2 <= ubp; t2++){
    lbt3 = max(8*t2,8*t1-8*t2-N+1);
    ubt3 = min(min(N-1,8*t2+7),8*t1-8*t2+6);
    for (t3 = lbt3; t3 <= ubt3; t3++){
      lbt4 = max(8*t1-8*t2,t3+1);
      ubt4 = min(8*t1-8*t2+7,t3+N-1);
      for (t4 = lbt4; t4 <= ubt4; t4++){
        a[t3][-t3+t4] = a[-t3+t4][t3] + a[t3][-t3+t4-1];
      }}}
```

Fig. 5.: Parallelized version of the example kernel (variable names are modified due to limitation of the paper space)

ber of threads.

In the next section, we propose an alternative method to handle these problems.

III. Threading Method for OpenMP Directives

A. Threading

In this section, we propose a method to transform a C source code with OpenMP directive `#pragma omp parallel for` into a description for high-level synthesis.

Our method is applicable to general parallelized loop with non-constant iteration domain ($B_L \leq i \leq E_L$) that can be expressed as in Fig. 6. This means that applicable scope of the method is not limited to codes generated by PLUTO.

We call an iteration of parallelized loop as "logical thread", and a processing element of synthesized circuit

```
#pragma omp parallel for private(i) shared(B_L, E_L)
/* NOTE: maybe (B_L > E_L). */
for (i = B_L; i <= E_L; i++) S(i);
```

Fig. 6.: General parallelized loop with an OpenMP directive

as "PE". We consider how to bind logical threads as "chunks", and how to assign chunks to a limited number of PEs. In the following, N_L defined in eq. (1) denotes the number of logical threads and PEs, and N_P denotes the number of PEs.

$$N_L = \begin{cases} E_L - B_L + 1 & (B_L \leq E_L) \\ 0 & (B_L > E_L) \end{cases} \quad (1)$$

The chunk size (i.e. the number of local threads bound per a PE) C and the actual number of utilized PE J can be computed as eq. (2). This indicates that only N_L PEs are actually utilized when N_L is smaller than N_P.

$$\begin{aligned} C &= \lceil N_L/N_P \rceil \\ J &= \lceil N_L/C \rceil \end{aligned} \quad (2)$$

t-th PE ($0 \leq t < N_P$) is assigned to logical threads with the statements $S(i)$ in range $i \in [b_t, e_t]$ if $t < J$. Here the bounds of physical-thread-local iteration variable b_t and e_t can be computed as eq. (3).

$$\begin{aligned} b_t &= B_L + C \times t \\ e_t &= \begin{cases} E_L & (t = J-1) \\ b_t + C - 1 & (t < J-1) \end{cases} \end{aligned} \quad (3)$$

If $t \geq J$, the thread just waits for completion of the other threads.

IV. Buffering Method for Off-Chip RAM Access

A. Advantage of Buffering

In this section, we propose a method to optimize accesses to off-chip RAM by constructing buffers using on-chip RAM blocks or registers. The buffers are independently constructed on each PE and can be accessed in less cycles than off-chip RAM. Our buffering method contributes to the reduction of the execution cycles of the kernel.

Suppose that the bus width of off-chip RAM is 64-bit and the size of an element of array `a[N][N]` is 8-bit. Under this assumption, the consecutive 8 elements of the array should be processed at once so as to exploit the off-chip RAM bandwidth. At a glance, it is obvious that a consecutive access such as `a[i][j-1]` of the example kernel can be easily done. On the other hand, how to access `a[j][i]` consecutively is nontrivial. Our proposed method can also access `a[j][i]` using tiling information generated by PLUTO.

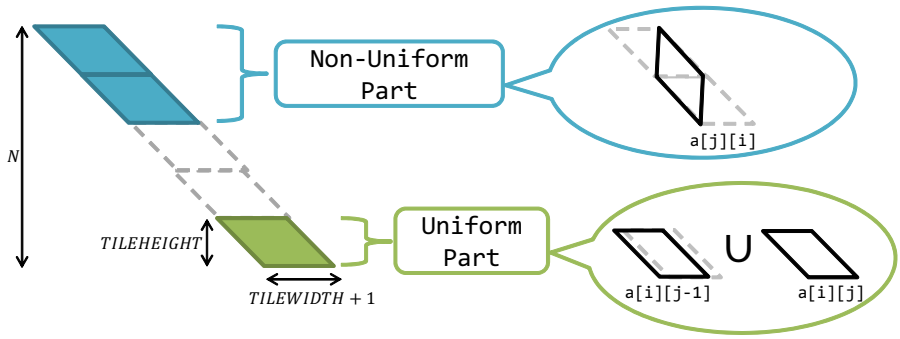

Fig. 7.: Partitioned iteration space of the example kernel

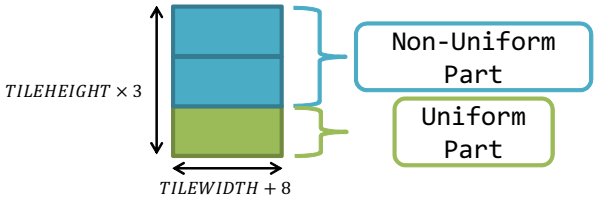

Fig. 8.: Partitioned buffer space of the example kernel

B. *Space Partition*

In our proposed method, the iteration space of logical thread is attributed as uniform part and non-uniform parts as in Fig. 7, then distorted to the rectangular buffer space as in Fig. 8.

The method can be applied to a SCoP kernel that satisfies the following conditions:

1. The kernel has two-dimensional iteration space.
2. The kernel has no conditional expression other than e_2 of `for(`e_1`;` e_2`;` e_3`)` statements, i.e., the kernel has no irregular access (also known as data-dependent access).
3. The kernel is a perfectly nested loop, i.e., there is no statement between `for` statements.
4. The kernel has only one array write access of which access vector corresponds to the iteration vector.

B.1. Uniform Part

The uniform part of the iteration space is an union of the domain of write access and domains of its uniform dependent read accesses when the iteration vector moves within one tile shape. In the example kernel, the uniform part is an union of the domain of (i, j) and the domain of $(i, j-1)$.

The size of the uniform part of the iteration space is larger than the original tile shape. Suppose that the size of original tile shape is (t_h, t_w) and the set of the distance vectors of uniform dependencies is D. Then the size of the uniform part of the iteration space (t'_h, t'_w) can be computed as eq. (4).

$$(t'_h, t'_w) = (t_h, t_w) + \sum_{d \in D} d \tag{4}$$

In the example kernel, $(t'_h, t'_w) = (8, 8) + (0, 1) = (8, 9)$ when the original tile size is 8×8. Ref. [2] describes components of d is 1 in most cases. Therefore we consider that (t'_h, t'_w) is not so much larger than (t_h, t_w).

For the buffer space, the size of the uniform part (t''_h, w''_h) is aligned to the off-chip RAM bus width along to horizontal direction because the off-chip RAM must be accessed for 8 elements at once. In the example kernel, $(t''_h, t''_w) = (8, 16)$. By aligning the size, no additional off-chip RAM read access and write back to unaligned element are needed.

B.2. Non-Uniform Parts

A non-uniform part of the iteration space encloses the domain of a non-uniform dependent read accesses when the iteration vector moves within one tile shape. Although there is invariably the single uniform part, non-uniform parts exist as many as non-uniform dependencies. Also note that although the uniform part is both readable and writable due to the limitation 4, non-uniform parts are not writable.

A non-uniform part is aligned to the size of uniform tiling shape, i.e., (t'_h, t'_w) for the iteration space and (t''_h, t''_w) for the buffer space. This limitation is due to reduction of the execution cycles and number of flip-flops of the synthesized buffer controller circuit. In the example kernel, the area of non-uniform part is as large as two uniform tiles. Generally, this area estimation can be done by analysis on the bounds of the non-uniform dependent access vector.

```
for assigned logical threads to the PE do
  Load the contents of off-chip RAM into the uniform part.
  Load the contents of off-chip RAM into the non-uniform parts.
  for iteration vector in a original tile do
    for array read access do
      if the accessed element exists in both uniform (RW) part and non-uniform (RO) part then
        Fetch the element in the uniform (RW) part into a register.
      else
        Fetch the element into a register.
      end if
    end for
    Execute the processing statement of kernel and store the result in the uniform (RW) part.
  end for
  Write back the contents of the uniform part to the off-chip RAM.
  {The PEs are synchronized here}
end for
```

Fig. 9.: Flow of a PE with buffer management function

C. *Buffer Size*

The buffer size per PE b can be computed as sum of the areas of uniform part b_u and non-uniform parts b_n of the buffer space. In the example kernel, this can be computed as eq. (5). The unit of b is the size of an array element.

$$b = b_u + b_n = ((t_w+8)\times t_h)+((t_w+8)\times 2t_h) = (t_w+8)\times 3t_h \quad (5)$$

The total buffer size B for N_P PEs can be computed as $B = b \times N_P$. If the original tile size is 8×8 and the synthesized circuit has $N_P = 8$ PEs, the total buffer size B is $384 \times 8 = 3072$ elements.

D. *Buffer Management Flow*

Fig. 9 shows the buffer management flow of each PE. An access to off-chip RAM must be done exclusively so as to single PE can acquire access rights.

V. Evaluation

We applied our proposed method to the example kernel in Fig. 2 and evaluated its effectiveness for threading-only version and threading+buffering version. We used Mentor Graphics Handel-C 5.1 as the high-level synthesis tool. We evaluated the execution cycles and number of flip-flops of the synthesized circuit using the Handel-C simulator.

We assumed an access for off-chip RAM needs 8 cycles and on the other hand on-chip buffers can be accessed in only one cycle. We also assume the bus width of off-chip RAM is 64-bit (i.e., 8 array elements) as in Section IV. We verified correctness of the synthesized circuit by calculating a checksum of the array after all write accesses to off-chip RAM finished.

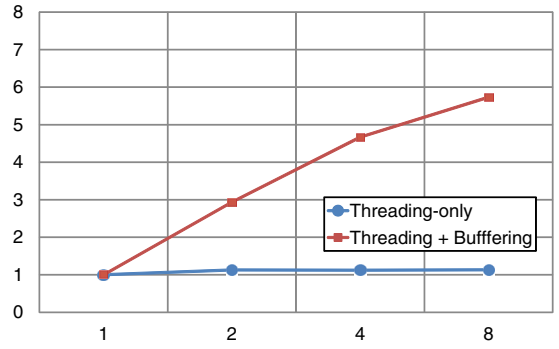

Fig. 10.: Speed-up of execution cycles for $N_P = 2, 4, 8$ (the tile size is 48×48)

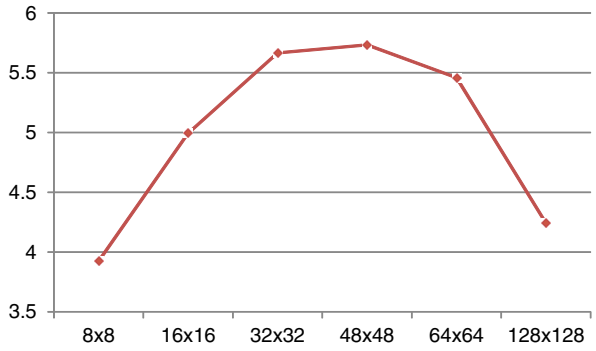

Fig. 11.: Speed-up of execution cycles for several tiling sizes (Threading+Buffering, $N_P = 8$)

As shown in Fig. 10, the speed-up of execution cycles to the original sequential kernel for $N_P = 2, 4, 8$ can be estimated. In this case, the array size of `a[N][N]` is set to 1024×1024 and the tile size is set to 48×48. The sequential kernel as the basing-point actually takes $77,522,948$ cycles for the execution. Our buffering method can accelerate the kernel to 5.73 times when $N_P = 8$. The experimental result also shows that the performance stays low without buffering.

The speed-up to the original sequential kernel for $N_P = 8$ when the tile size varies can be estimated as Fig. 11. The best speed-up can be achieved when the tile size is 48×48. The result indicates that too small tile size leads to poor performance because of small buffers. In contrast, too large tile size also leads to poor performance. This seems to be caused by the waiting time of each PEs to acquire rights to access off-chip RAM.

The buffer size b calcuated as eq. (5) can be displayed as Fig. 12. When the tile size is set to 48×48 so as to achive the best performance, b becomes $8,064$ array

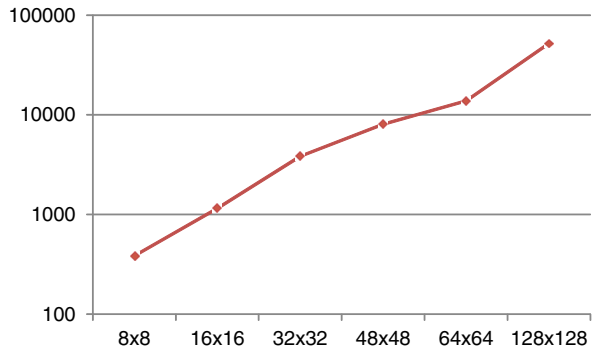

Fig. 12.: Buffer size b (in number of 8-bit array elements) for several tiling sizes

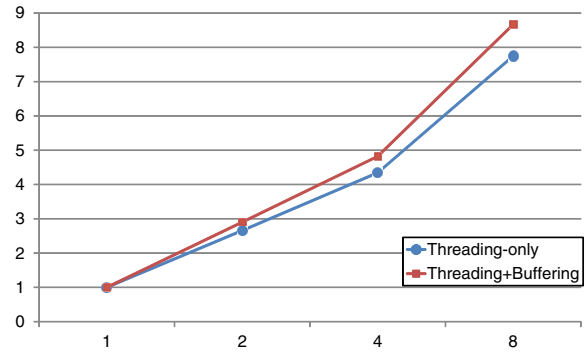

Fig. 13.: Increase of the number of flip-flops

elements.

The increase of the number of flip-flops (including circuits other than the kernel) can be estimated as Fig. 13. The actual number of the basing point is $1,582$ flip-flops. It should be noted that the number of flip-flops is constant to the variation of tile size. The increase is almost linear to N_P and there is only small difference between threading-only version and threading+buffering version. This indicates that the area of flip-flops for the buffer controller circuit is not large.

VI. Conclusions

We proposed a new method to handle nested loop kernel with non-uniform dependencies efficiently in high-level synthesis. Our proposed method achieved 5.73 times speed-up for 8 PEs when the tile size is 48×48. To the best of our knowledge, this is the first work to handle non-uniform dependencies in high-level synthesis.

We are planning to apply our proposed method to other kernels and evaluate its effectiveness.

Acknowledgements

This work is supported by VLSI Design and Education Center(VDEC), The University of Tokyo with the collaboration with Mentor Graphics Corporation.

References

[1] Guiming Wu, Yong Dou, and Miao Wang. Automatic synthesis of processor arrays with local memories on fpgas. In *FPT '10*, pp. 249–252. IEEE, 2010.

[2] Louis-Noël Pouchet, Peng Zhang, P Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proc. of the Int'l Symposium on FPGAs*, pp. 29–38. ACM, 2013.

[3] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *Int'l Journal of Parallel Programming*, Vol. 28, No. 5, pp. 469–498, 2000.

[4] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. of the 13 th Int'l Conf. on PACT*, pp. 7–16. IEEE Computer Society, 2004.

[5] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In *Proc. of the 2006 GCC Developers Summit*, 2006.

[6] Tobias Grosser, Hongbin Zheng, Raghesh A, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly - polyhedral optimization in llvm. In *Proc. of the First Int'l Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2011.

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08*, 2008.

[8] Uday Kumar Reddy Bondhugula. *Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model*. PhD thesis, Ohio State University, 2008.

[9] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *Int'l Journal of Parallel Programming*, Vol. 25, No. 6, pp. 447–496, 1997.

[10] Y.Y. Leow, C.Y. Ng, and W.F. Wong. Generating hardware from openmp programs. In *FPT '06*, pp. 73–80. IEEE, 2006.