Register Blocking: An Analytical Modelling Approach for Affine Loop Kernels

Anthimopulos, Theologos; Keramidas, Georgios; Kelefouras, Vasilios; Stamoulis, Iakovos

DOI:
10.1145/3649153.3649194

Publication date:
2024

Document version:
Publisher's PDF, also known as Version of record

Register Blocking: An Analytical Modelling Approach for Affine Loop Kernels

Theologos Anthimopoulos
School of Informatics, Aristotle University of Thessaloniki
Thessaloniki, Greece
tanthimop@csd.auth.gr

Georgios Keramidas
School of Informatics, Aristotle University of Thessaloniki
Thessaloniki, Greece
gkeramidas@csd.auth.gr

Vasilios Kelefouras
School of Engineering, Computing and Mathematics,
University of Plymouth
Plymouth, United Kingdom
vasilios.kelefouras@plymouth.ac.uk

Iakovos Stamoulis
Think Silicon, S.A. An Applied Materials Company
Patras, Greece
i.stamoulis@think-silicon.com

ABSTRACT

For the past several decades, optimizing compilers have been a primary area of focus in both industry and academia. This continued research interest is a testament to the complexity of this task, primarily stemming from the vast number of parameters that must be explored to attain near-optimal results. One of the key compiler optimizations is "Register Blocking (RB)" also known as "Register-level Tiling" or "unroll-and-jam". RB can strongly reduce the number of executed Load/Store (L/S) instructions, and as a consequence the number of data accesses in memory hierarchy, but due to its inherent complexities, fine-tuning is essential for its effective implementation. To address this problem, in this work a new methodology is proposed for RB. The RB factors, the loops to apply RB, the number of allocated variables/registers per array reference, and the loops' ordering are generated by an analytical model, leveraging the target hardware (HW) architecture details and loop kernel characteristics. The proposed methodology has been evaluated on both embedded and general-purpose CPUs across seven well-known loop kernels, achieving high speedups and L/S instruction gains over GCC compiler, handwritten optimized codes, and the popular Pluto tool.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; • **Computing methodologies** → **Parallel algorithms**.

KEYWORDS

Compiler Optimizations, Register Blocking, Register Tiling, Unroll-and-Jam, High Performance Computing, Data Reuse, CPUs

ACM Reference Format:
Theologos Anthimopoulos, Georgios Keramidas, Vasilios Kelefouras, and Iakovos Stamoulis. 2024. Register Blocking: An Analytical Modelling Approach for Affine Loop Kernels. In *21st ACM International Conference on Computing Frontiers (CF '24), May 7–9, 2024, Ischia, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3649153.3649194

CF '24, May 7–9, 2024, Ischia, Italy

ACM ISBN 979-8-4007-0597-7/24/05.
https://doi.org/10.1145/3649153.3649194

1 INTRODUCTION

Register Blocking (RB) is a crucial compiler optimization, particularly for memory-bound loop kernels [8]. It enhances the Arithmetic Intensity (ARI) of the program, improving its position in the roofline model. This increase in ARI is achieved by reducing the number of executed L/S instructions. RB is also essential in crafting micro-kernels (ukernels) for optimized libraries like Intel oneDNN [6]. Generally, RB combines two compiler optimizations: loop unroll and scalar replacement. Loop unrolling creates common array references in the loop body, allowing them to be replaced by variables. This reduces the number of L/S instructions. While modern compilers support RB, manual application by experienced programmers can lead to notable performance enhancements. This is due to the complexity involved in overcoming challenges that compilers may face, particularly in handling kernels with a high number of loops, like convolution kernels [25].

The **RB challenges** addressed in this paper **within a unified framework include**:**(i)** identifying loops suitable for RB, **(ii)** determining the RB factor for each loop, **(iii)** allocating the appropriate number of registers for each array reference, **(iv)** coordinating RB with other code optimizations, such as loop permutation, and **(v)** addressing scenarios where RB factors do not perfectly divide the loops' upper bounds. These challenges intensify the computational complexity involved in optimizing kernels to determine the optimal RB parameters, especially tailored for a specific processor. The difficulty is particularly pronounced when dealing with loop kernels featuring a substantial number of loops.

```
for (b=0;b<B;b++) //batch
  for (m=0;b<M;m++) //feature maps (channels)
    for(y=0;y<Y;y++) //output height
      for(x=0;x<X;x++) //output width
        for(ky=0;ky<KY;ky++) //kernel's height
            for(kx=0;kx<KX;kx++) //kernel's width
                for(d=0;d<D;d++) //input Depth
                out[b][y][x][m]+=
in[b][y*str.y+ky][x*str.x+kx][d]*filter[m][ky][kx][d];
```

Figure 1: 2D forward convolution kernel

More specifically, in this paper, an analytical model for RB is proposed addressing the above five challenges. Since the target of RB is to reduce the number of executed L/S instructions, we use the number of L/S instructions as the objective function in our methodology. It is crucial to emphasize that the minimum number of L/S instructions does not always align with execution time, as other factors, such as cache misses, also affect execution time. Consequently, additional optimizations, such as loop tiling, are often applied after RB to further enhance overall performance. However, in this work, we focus on RB and loop permutation only.

The proposed methodology drastically prunes the exploration space, first, by using mathematical equations to constraint the number of RB factors and variables/registers used based on the number of available hardware registers and algorithm characteristics and, second, by using equations to approximate the number of executed L/S instructions based on the RB factors and input parameters of the target loop kernels. Last, our methodology is realized in the form of a standalone tool (based on Pluto) targeting to streamline the time-consuming development process of highly optimized ukernels.

The contributions of this paper are: i) an analytical model for efficient RB, ii) a standalone source-to-source transformation tool that can facilitate the development process of generating ukernels, and iii) an experimental evaluation that demonstrates the efficacy of the proposed method (an average speedup of 3.7x over GCC on scalar and vectorized kernels is achieved). Additionally, the proposed method outperforms Pluto, a tool widely used in industry, by 1.85x on average in scalar kernels.

Structure of the paper: Section 2 outlines the related work. Section 3 presents our approach. Section 4 describes our evaluation framework. The validation strategy of our tool is presented in Section 5. Section 6 provides our experimental results using seven well-known loop kernels. Section 7 concludes this work.

2 BACKGROUND AND RELATED WORK

Background: An example of RB is illustrated in Fig. 1 through a 2D forward (***forw.***) convolution kernel. Fig. 2a depicts the kernel when loop unrolling is applied, while Fig. 2b shows the case when both scalar replacement and loop unrolling are employed at the same time (for convenience we assume: *B=1, KY=1, KX=1 and Stride=1*). As it observed, by unrolling the m and x loops, common array references occur in the innermost loop body (Fig. 2a that are consequently replaced by variables (Fig. 2b). The arrays are now accessed fewer times from memory e.g., the *filter[]* array elements are loaded and re-used three times. Note that although it is always safe to unroll one loop, it is not always safe to unroll more than one loops; for the latter a data dependence analysis is required to ensure the correct program functionality.

This paper tackles RB in the context of a set of perfectly nested loops characterized by array subscripts that are Single-Input Variables (SIV) and separable [4]. Although addressing this category of loop kernels, seems somehow restrictive, RB is normally not efficient in cases where these conditions do not hold [4]. Addressing more general cases is left for future work.

Therefore, given a number of perfectly nested loops, we develop a methodology addressing the following problems: **i)** find the best RB factor for each loop, **ii)** find the best order of the loops (loop permutation), **iii)** find the number of variables/registers used for each array reference, and **iv)** find the RB factor values for the padding code (multiple ukernels might be generated in this case).

```
1  for (m=0;b<M;m+=2) //feature maps
2   for(y=0;y<Y;y++) //output height
3    for(x=0;x<X;x+=3) //output width
4      for(d=0;d<D;d++){ //input Depth
5 //m=0
6 out[y][x][m]+=in[y][x][d]*filter[m][d];
7 out[y][x+1][m]+=in[y][(x+1)][d]*filter[m][d];
8 out[y][x+2][m]+=in[y][(x+2)][d]*filter[m][d];
9 //m=1
10 out[y][x][m+1]+=in[y][x][d]*filter[m+1][d];
11 out[y][x+1][m+1]+=in[y][(x+1)][d]*filter[m+1][d];
12 out[y][x+2][m+1]+=in[y][(x+2)][d]*filter[m+1][d];
13    }
```

((a))

```
1   for (m=0;b<M;m+=2) //feature maps (channels)
2    for(y=0;y<Y;y++) //output height
3     for(x=0;x<X;x+=3){ //output width
4 out_reg00 = out[y][x][m]; .. out_reg12=out[y][x+2][m+1];
5       for(d=0;d<D;d++){ //input Depth
6         filter0=filter[m][d];
7         filter1=filter[m+1][d];
8         x_reg0 = in[y][x][d];
9         x_reg1 = in[y][(x+1)][d];
10        x_reg2 = in[y][(x+2)][d];
11        //x=0
12        out_reg00 += x_reg0 * filter0;
13        out_reg10 += x_reg0 * filter1;
14        //x=1
15        out_reg01 += x_reg1 * filter0;
16        out_reg11 += x_reg1 * filter1;
17        //x=2
18        out_reg02 += x_reg2 * filter0;
19        out_reg12 += x_reg2 * filter1;
20       }
21 out[y][x][m]=out00; ... out[y][x+2][m+1]=out12;
22    }
```

((b))

Figure 2: Optimised *forw.* convolution using a) loop unrolling and b) scalar replacement after loop unrolling

Fig. 3 sheds light on questions 1 and 3 illustrating the impact of RB application for various Unroll Factors (UFs). As we observe, there is a strong correlation between the number of L/S instructions and RB factors wrt. the utilization of hardware registers. In the context of this example, RB is applied only to the outermost loop. As it is evidenced by Fig.3a, the larger the RB factor (and therefore the more the HW registers used), the lower the number of L/S instructions; however, when we exceed the number of available hardware (HW) registers in the innermost loop body, then the number of L/S instructions is highly increased (when the inflection point equals to the number of the hardware registers, which is 16).

Although finding the best RB factor of a single loop (as in Fig.3a) is straightforward, this is not the case when multiple loops must be considered. Fig. 3b shows the effect of RB on one and two loops in MMM kernel; as shown, selecting the suitable loops for RB is also important. Another important consideration is the strong correlation between RB and loop permutation. Altering the loop order of a kernel allows for the movement of various array references to the outer loop(s), underscoring the interplay between RB and loop permutation. For example, in Fig. 1, the *out*[] array can be moved outside of (ky,kx,d) loops; however, if we put the *m* loop as the innermost loop, then *in*[] array can be moved outside of the

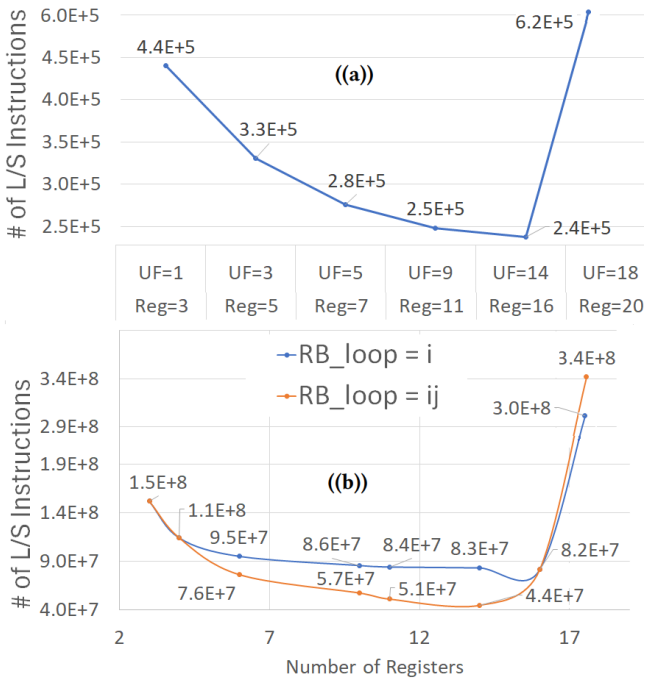

Figure 3: Evaluation of RB for a) Matrix-Vector Multiplication (MVM) and b) Matrix-Matrix Multiplication (MMM) kernels

innermost (m) loop. This affects the way RB is applied as well as the number of L/S instructions.

Another challenge in RB is padding code (epilogue code), required when the RB factor does not perfectly divide the loop's upper bound. Related works address this issue either by using RB factors that perfectly divide the loop or by using a padding ukernel in the end without RB [19]. As we prove in this work, the aforementioned approaches are not the most efficient and alternative RB factors must be examined for the main loop and also for the padding ukernel.

Related work: RB can be applied manually or automatically through compilers or optimization tools. While manual application is time-consuming and error-prone, it often results in superior performance. On the other hand, automatic application is more convenient, but offers poor results [24]. Given the vast exploration space and the current limitations of compilers, semi-automatic optimization approaches have emerged. Examples are Pluto [2, 17], PoCC [18, 20], R-stream [15, 23], PPCG [26], and Orio [16], where the optimization process is managed jointly by the programmer and the compiler. In these approaches, the programmer sets crucial parameters, such as the RB factor, while the execution of the RB task is automated. Despite being successful in their own scope, tools like Pluto pose notable limitations, prompting performance engineers to resort in manually applying RB. In particular, Pluto i) does not provide support for vectorized input code, e.g., with AVX/NEON intrinsics, and ii) it does not offer to the programmer the flexibility to selectively specify certain loops for RB, as Pluto uniformly applies the same RB factor to all loops except to the innermost.

PoCC++ is an enhanced version of Pluto+ featuring an improved RB approach [1]. PPCG suffers from for "Register Pressure (register spills)" effect [26]. On the other hand, R-stream is mostly dedicated on modern GPUs. The mentioned tools, while useful, come with their own set of limitations. They often necessitate heuristic methods for determining optimal UFs for implementing RB or rely on fixed methodologies. A notable limitation is their lack of efficiency, as they are not inherently aware of the underlying HW architecture, the constant parameters of the input kernel, and rely on compilers, such as GCC or Clang, for efficient register utilization. Also most of these tools can not unroll vectorized kernels.

Table 1: Other papers related to this work

Related Works	Combine RB with other opt.	source-to -source	Use of ukernels	Consider all UFs
[9],[10]	✓	✓	✓	✗
[21]	✗	✗	✗	✓
[3]	✗	✓	✗	✓
[4]	✗	✗	✗	✓
[13]	✗	✗	✗	✗
This work	✓	✓	✓	✓

RB is also tackled by various research works such as [3, 4, 9, 10, 13, 21]. Table 2 highlights the major limitations of related approaches. PoCC++ [9, 10] employs a different objective function and also uses a limited number of UF. In [21], the authors propose an objective function that does not take into account the loops' upper bounds and it assigns an unnecessary higher number of registers. Also, loop permutation is not addressed by [3, 4, 21]. Finally, [13] addresses the register pressure problem in loop unrolling on vectorized codes; [13] forces the UFs to be equal to the power of two and relies on a heuristic approach defined by the user (i.e., user provides max UF).

Compared to related works, the proposed method i) achieves lower number of L/S instructions by applying register allocation more efficiently, ii) effectively handles cases where the RB factors cannot perfectly divide the loops, and iii) the method addresses RB and loop permutation as a unified problem, recognizing their interdependence, leading to a further exploration space reduction.

Finally, there are manually optimized libraries for specific algorithms such as OneDNN [12] and CMSIS-NN [11]. The RB parameters and the loops to apply RB are found by using exploration and fine-tuning. Last, there are algorithm-specific methodologies that apply RB for popular kernels, such as MMM, Jacobi, and convolution, that try to find suitable RB factors either by using empirical search or analytical models [5, 7, 24].

3 PROPOSED METHODOLOGY

The proposed methodology [20] is depicted in Fig. 4. Since the RB and loop permutation optimizations depend on each other, RB should be applied for all different loop permutations (so as not to exclude any efficient solutions). However, we show that RB is affected only by a limited number of loop permutations, thus the exploration space is reduced by applying RB for a limited number of permutations only. Then, for each of the remaining loop permutations, the RB factors and the number of required variables/registers for the array references are constrained by mathematical equations, based on the number of HW registers and algorithmic characteristics; all the solutions that do not satisfy these equations are discarded as inefficient. Last, we select the solution that minimizes the number of L/S instructions by using an analytical model.

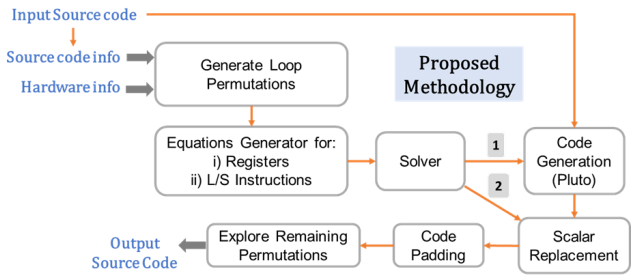

Figure 4: Proposed methodology

Once the RB parameters are determined, we employ the Pluto tool to carry out loop unrolling (Pluto does not support RB, but loop unroll only, with just one factor for all the loops). To this end, we have modified Pluto's source code to implement loop unrolling with varying factors for different loops. The subsequent phase involves the manual application of scalar replacement (the automation of this step is left for future work). The final steps include adding the padding ukernels and perform an exploration step to extract the solution with the most efficient loop permutation (among the ones minimizing our target objective function). The proposed methodology acts as a source-to-source tool, aiding in the development of microkernels in optimized libraries for diverse applications.

Input: The first input to our tool is the annotated source code. The target loop kernels are annotated by using Pluto's annotations, so as Pluto can carry out the loop unroll transformation. As shown in Fig. 4, the second input to our tool (source code info) is a text file containing vital information about the input kernels, such as the number of arrays and their names, the number of loops and their respective names, the array subscripts, and the loops where vectorization was applied (optional feature). This information can be easily extracted from the source code using LLVM or Pluto, but in this work was performed manually. Finally, the length of the vectorization engine and the number of HW registers are also provided as additional inputs (hardware info).

Generate Loop Permutations & Explore Remaining Permutations: Loop permutation affects RB only when it allows for an array reference to be relocated to an outer loop(s). For example, in Fig. 1, *out* array reference can be moved out of the three innermost loops, affecting the way RB is applied, thus reducing the L/S instructions. However, if the loops are in the following order (b,d,y,x,ky,kx,m), then the *in* and not *out* reference will be moved outside of the innermost loop affecting the way RB is applied. Similarly, the following two permutations (b,d,y,x,ky,kx,m), (b,d,x,y,ky,kx,m) do not affect RB; the way RB is applied is the same and the number of L/S instructions remains the same. A detailed analysis of the latter is provided in Section 3.1. Although these two permutations do not affect the number of L/S instructions, they do affect execution time and this is why an exploration step is applied in the last box of Fig. 4; in this step, all the remaining loop permutations are run and the fastest one is selected.

Equation Generator for Registers: The next step is to constraint the RB factors and the number of variables/registers used for each array reference. To this end, Eq. 1 is generated. All the solutions that do not satisfy Eq. 1 are discarded.

$$Arr.refs + other \leq Arrays.Regs + other \leq Num.Regs \quad (1)$$

where $Arr.refs$ is the number of array references in the loop body, $Arrays.Regs$ is the number of variables/registers allocated for all the array references (given by Algorithm 1), $other$ is number of any other variables included in the loop body and $Num.Regs$ is the number of hardware registers (either floating point or integer registers depending on the arrays' data). Note that $Arrays.Regs$ depends on the RB factor values. The upper bound of Eq. 1 derives from the fact that if more registers than the available are used, the number of L/S instructions will be increased due to register spilling. On the other hand, the lower bound derived from the fact that at least one register is needed for each array reference.

Algorithm 1 calculates $Arrays.Regs$ and the number of registers need for each array reference (reg_per_ref[]) for fixed RB factors. Initially, an iteration for each array reference is performed (line 4) that leads to three *IF* cases. *The first IF statement (line 6)* is followed for array references that can be relocated to outer loop(s). The number of registers used for an array reference is given by $Array_i Regs = \prod_{j=1}^{j=n} UF_j$, where n is the number of iterators consist in the subscript of the ith array reference, while UF_j is the unroll factor of loop j. For example, in Fig. 2a, the array *out[b,y,x,m]* does not depend on *(ky,kx,d)* subscripts, thus it moved to lines 5-7 of Fig. 2a. Also, $Array_{out}.Regs = UF_m \times UF_x$ since $UF_b = UF_y = 1$. *The second IF statement (line 12)* is followed when an array is reused in the innermost-loop body (e.g., the *filter* and *out* array references in Fig. 2a). *The third IF statement (line 19)* is activated when an array reference is not reused. In this case only one register is allocated to save the precious registers for the references achieving data reuse.

A crucial part of Algorithm 1 relates to the max function used in line 15, which allows for more efficient register allocation compared to the existing methods. For clarity reasons, this is further discussed in Sec. 3.1.

Equation Generator for L/S Instructions: For each solution that satisfies Eq. 1 (this includes the RB factors and the number of registers per array reference), we approximate the number of L/S instructions.

Algorithm 2 presents the pseudocode of our approach when applying fixed RB factors. Initially, the number of L/S instructions of each array reference takes the value of the product of all the loops' upper bounds (lines 4-6). If an array reference is relocated to an outer loop(s) (if-statement in line 8), then its number of executed L/S instructions is lower, thus the number of L/S instructions is updated/reduced (lines 9-13). On the other hand, if an array reference is reused in the innermost-loop body (line 15), its number of L/S instructions needs to be decreased. In this case, we need to divide by the RB factor values (line 18). Finally, for vectorized codes, our algorithm will go through lines 19-23; in this case the number of L/S instructions need to be further reduced as a single vector L/S instruction includes multiple scalar L/S instructions. A working example of Algorithm 2 is provided in Section 3.1.

Equations Solver: In this step, the solution that minimizes the objective function is found. This is realized by applying Algorithm 2 (calculating the number of L/S instructions) for each solution satisfying Eq. 4. In particular, for each loop permutation, the solver

Algorithm 1 Calculate registers when RB applied to specific loops

Input: array_references[], rb_loops[], rb_factors[]
Output: Registers_used, reg_per_ref[]
$Arrays.Regs = 0$, $max\_reg = 0$ //init.
for it in number_of(array_references) **do**
  $tmp\_reg = 1$
  if array_references[it] can moved in outer loop(s) **then**
    for i in $number\_of(rb\_loops)$ **do**
      if $rb\_loops[i] \in array\_references[it]$ **then**
        $tmp\_reg *= rb\_factors[i]$
  else
    if array_references[it] is reused in innermost-loop body **then**
      for i in $number\_of(rb\_loops)$ **do**
        if $rb\_loops[i] \in array\_references[it]$ **then**
          $tmp\_reg *= rb\_factors[i]$
      $max\_reg = \max(tmp\_reg, max\_reg)$
      if $max\_reg == tmp\_reg$ **then**
        $max\_it = it$
    else
      $tmp\_reg = 1$
  $Arrays.Regs += tmp\_reg$
  $reg\_per\_ref[it] = tmp\_reg$
$Arrays.Regs -= max\_reg + 1$
$reg\_per\_ref[max\_it] = 1$

outputs the following: **i)** the UF for each loop (arrow *1* in Fig. 4) and **ii)** the number of registers needed per array reference (arrow *2)*.

Code Generation based on Pluto tool: Having the RB parameters extracted, the next step is to generate the output source code. As noted, RB consists of two main steps: loop unroll and scalar replacement. Loop unroll is carried out by enhancing/modifying Pluto tool so as to support multiple unroll factors per loop.

Scalar Replacement: In this step, scalar replacement transformation is carried out. The number of variables/registers to use for each array reference (*reg_per_ref[]*) is calculated in Algorithm 1. The number of variables/registers allocated in the innermost loop body does not exceed the number of the available HW registers. It is important to note that different scalar replacement options exist even for fixed RB factors. As mentioned in Algorithm 1, we use a max function to better utilize the limited and precious hardware registers. Due to this reason, this step cannot be automated by the compiler and this is why it is carried out manually.

Code Padding: In this step we apply padding using more than one ukernels. This step is crucial when the RB factors' values are comparable to the loops' upper bounds, e.g., when $X = 5$ in Fig. 2.

When the RB UF does not perfectly divide the loop's upper bound (N), we configure the padding ukernel. If ($N \mod UF == 1$) then the last ukernel uses a RB factor of $UF + 1$ e.g., if $N = 13$ and $UF = 6$, then there will be two ukernels one with $UF = 6$ and another with $UF = 7$. As it is further discussed in Section 4, using a slightly larger/smaller UF does not significantly impact performance. On the contrary, if the factor is significantly larger/smaller then performance is degraded. If ($N \mod UF == N - 1$), then the last ukernel will have an RB factor of $UF - 1$. In the remaining cases, the last two ukernels will be assigned UF_1/UF_2 values, so as they both are as close to the RB factor as possible.

Algorithm 2 Cal. L/S with RB applied to specific loops (rb_loops[])

Input: array_references[], vector_length, Loops[], rb_loops[], RB_factor[], vector_length
Output: l_s
$l\_s = 0$, $vec\_unroll = \frac{vector\_length}{\text{sizeof}(array.\text{type})}$, $Init\_ls\_inst = 1$
for each loop in Loops[] **do**
  $Init\_ls\_inst *= loop$.upper_bound
for each array_reference in array_references[] **do**
  $array\_ls\_inst = Init\_ls\_inst$
  if array_reference can moved in outer loop(s) **then**
    for each loop in Loops[] with innermost order **do**
      if loop ∉ array_reference.subscripts **then**
        $array\_ls\_inst \mathrel{/}= loop$.upper_bound
      else
        break
  else
    if array_reference is reused in the innermost-loop body **then**
      for i in number_of(rb_loops) **do**
        if $rb\_loops[i] \notin array\_reference$ **then**
          $array\_ls\_inst \mathrel{/}= RB\_factor[i]$
  if array_reference is stored **then**
    if vectorized_loop ∈ array_reference **then**
      $l_s += \frac{array\_ls}{vec\_unroll}$
    else
      $l_s += array\_ls$
  if array_reference is loaded **then**
    $l_s += \frac{array\_ls}{vec\_unroll}$

3.1 Working Example

In this section, we will exemplify its application (in a step-by-step basis) using the example in Fig. 1.

Generate Loop Permutations. The loop permutations that affect RB are: **i)** P1={*(b,y,x,m) (k.y,k.x,d)*}: *out[]* is moved out of (k.y,k.x,d) loops, **ii)** P2={*(b,y,x,k.y,k.x,d) (m)*}: *in[]* is moved out of (m) loop, and **iii)** P3={*(m,k.y,k.x,d) (b,y,x)*}: *filter[]* is moved out of (b,y,x) loops.

In the above permutations, any order of the loops inside a parenthesis does not affect the application of RB and therefore the number of L/S instructions is identical, e.g., $\{(b, y, x, m), (k.y, k.x, d)\}$ and $\{(b, x, y, m), (d, k.y, k.x)\}$ do not affect RB. Loop permutation affects RB only when the innermost loop (or loops) are the ones that are not part of the array subscript. Thus, in this case RB is applied only for these three loop permutations. Although the three loop permutation categories above contain multiple loop permutations each that do not affect RB, they do affect the kernel's execution time; this why a small exploration step is applied in the last step of Fig. 4, e.g., for P1, all different loop permutations inside the following parenthesis of (b,y,x,m) and (k.y,k.x,d) are considered.

Equations Generation. The register equation is the following :

$$Regs \geq PUF + UFy \cdot UFx + UFm - \max(UFy \cdot UFx, UFm) + 1 \quad (2)$$

```
  for (m=0;b<M;m+=2) //feature maps (channels)
   for(y=0;y<Y;y++) //output height
    for(x=0;x<X;x+=4){ //output width
out00 = out[y][x][m]; .. out13=out[y][x+3][m+1];
      for(d=0;d<D;d++){ //input Depth
        filter0=filter[m][d];filter1=filter[m+1][d];
        //x=0
        x_reg=in[y*str.y][x*str.x][d];
        out00 += x_reg * filter0;
        out10 += x_reg * filter1;
        //x=1
        x_reg = in[y*str.y][(x+1)*str.x][d];
        out01 += x_reg * filter0;
        out11 += x_reg * filter1;
        //x=2
        x_reg = in[y*str.y][(x+2)*str.x][d];
        out02 += x_reg * filter0;
        out_reg12 += x_reg * filter1;
        //x=3
        x_reg = in[y*str.y][(x+3)*str.x][d];
        out03 += x_reg * filter0;
        out13 += x_reg * filter1;
       }
out[y][x][m]=out00; .. out[y][x+3][m+2]=out13;
    }
```

Figure 5: Proposed register allocation

where: i) PUF is the number of registers for *out[]* and $PUF = UFx \times UFy \times UFm$, ii) $UFy \times UFx$ is the number of registers used for *in[]*, and iii) UFm for *filt[]*. The L/S equation is the following:

$$L/S.instrs = 2(M \cdot X \cdot Y) + \frac{M \cdot X \cdot Y \cdot D}{UF_m} + \frac{M \cdot X \cdot Y \cdot D}{UF_y \cdot UF_x} \quad (3)$$

where: i) *(M*X*Y)* are the L/S instructions for *out[]*, ii) *(M*X*Y*D)/UFm* are the load instructions for *in[]*, and iii) *(M*X*Y*D)/(UFy*UFx)*, the loads for *filt[]*. In addition, since i) our target is to minimize the L/S instructions and ii) the parameter *UFd* is not part of eq. 3, a *UFd* equal to one is presumed.

Solver. Assuming $Num.Regs = 11$ (11 hardware registers) this step gives $UF_m = 2$, $UF_y = 1$, and $UF_x = 4$. Also, there are $(8, 2, 1)$ allocated registers for $(out, filter, in)$, arrays respectively.

Code Generation & Scalar Replacement.

The output source code is shown in step 5. In Section 3, we mentioned that a crucial step of Algorithm 1 is the max function used to improve register allocation. If we apply Algorithm 1 without using this technique, then the output code is the one shown in Fig.2b. In the figure, there are 2/3 variables for $filter/in$ arrays, respectively, and in overall 11 registers are needed. However, instead of loading all the $filter/in$ values prior computation, we can use just one register for the *in* array and overwrite it three times (Fig.5); this way, not all the arrays' elements are loaded prior computation. By doing this trick, we can reduce the number of registers required further increasing the RB factor. This will give an even lower number of L/S instructions. For example, in Fig. 5 we use a higher RB factor for x loop by using the same number of registers as in Fig. 2b.

4 EXPERIMENTAL FRAMEWORK

To validate our methodology, we use seven well-known loop kernels with single floating point precision (type float): Matrix-Matrix Multiplication (***MMM***), 2D forward Convolution (***Conv_forw***), Matrix-Vector Multiplication (***MVM***), backward Convolution (***Back_prop***), Gradient Decent (***Grad_des***), a kernel with mixed vector multiplication and matrix addition (***Gemver***; first loop kernel only) and a multi-resolution analysis kernel (***Doitgen***). Valgrind [22] is used to extract L/S instructions and memory/cache accesses of the kernels. The initial, scalar (un-optimized version of the kernels' source code is provided in the Appendix. In all cases, the correctness of each experiment was validated by cross-referencing it with the output of scalar code. For the ARM ISA, the vectorization code follows the same logic as in the x64 ISA vectorized code (we transform the NEON intrinsics using 128-bit vector length).

We use the standard -O2 and -O3 GGC optimization level for the scalar and vectorized kernels, respectively. We use -O2 in the scalar case to prevent GCC to apply auto-vectorization (disable -ftree-vectorize). The kernel sizes are selected in order to cover a wide range of cases. We configured the conv. kernels with the following sizes (*(b,y,x,m,kx,ky,d)*): i) size1: 20,30,30,32,1,1,32, ii) size2: 20,30,30,256,1,1,128, and iii) size3: 20,60,60,256,1,1,128. In the remaining kernels, we use square arrays and the selected sizes are depicted in the graphs. Our methodology is assessed in terms of speedups and FLOPS conducting experiments on i) Nvidia Carmel ARM V8.2, 64-bit, 1.9GHz CPU with a 4MB L2-cache and 8MB L3-cache with a peak performance of 30.4 GFLOPs/core and ii) Intel i9-10850K, x64, 64-bit, 5.2GHz CPU with a 256MB L2-cache and 20MB L3-cache with a peak performance of 166.4 GFLOPs/core. To obtain precise timing measurements, each kernel is executed multiple times for at least one second and the smallest time is reported.

5 VALIDATION USING VALGRIND

As noted, to validate the accuracy of our model, we used Valgrind [22], a widely used profiling tool capable of estimating memory accesses (L/S instructions and memory/cache accesses). We relied on seven well-known kernels considering both scalar and vectorized kernels. Table 2 presents the absolute error of reads/writes between our methodology and Valgrind for various RB factors. Notably, in the scalar case, the error is negligible, approaching zero. In vectorized scenarios, the error remains close to zero, except for ***Conv_back_prop*** and ***Conv_grad_des***, where a 7-8% error is observed. Our analysis reveals that this discrepancy is attributed to aggressive compiler optimizations affecting the memory accesses of the kernels, as indicated in the following LLVM issue [14]. In summary, our analytical model shows high accuracy with near-zero error rates, instilling confidence in the experimental results.

6 EXPERIMENTAL RESULTS

This section is divided into four main parts. The first part outlines the speedups and FLOPs increases achieved by our approach over the GCC compiler on the ARM platform for both scalar and vectorized codes. The second part shows the corresponding statistics for the x64 platform. In third part compares our methodology with the Pluto tool on both platforms. Finally, the last part showcases the additional benefits of padding when combined with RB.

Evaluation over GCC on ARM: Fig. 6 depicts the speedups offered by our approach for scalar (blue bars) and vectorized (orange bars) kernels over GCC on the ARM platform. On average, our methodology achieves 3.1x lower execution times for scalar codes

Table 2: Error (%) of L/S instructions extracted by the proposed analytical model compared to Valgrind

kernel	code case	RB	errors	kernel	code case	RB	errors	kernel	code case	RB	errors
Conv_forward	AVX	m=2 m=8 m,x=2,5	1% 1% 0%	Gemver, MVM	AVX	i=4 i=8 j=12	0% 0% 0%	Grad_des	AVX	x=4 d,x=2,4 d,x=3,4	2% 2% 8%
	Scalar	m=2 m,y=2,2 m=8	0% 0% 0%		Scalar	i=2 i=4 i=8	0% 0% 0%		Scalar	d=2 d,x=2,2 d=8	0% 1% 1%
MMM	AVX	j=4 i,j=2,2 i,j=4,3	0% 0% 0%	diogen	AVX	p=2 p=4 q,p=3,3	0% 0% 0%	Back_prop	AVX	d=4 m,d=2,4 d,x=3,4	2% 1% 7%
	Scalar	j=2 i=4 i,j=4,2	0% 0% 0%		Scalar	p=2 p=4 q,p=4,2	0% 0% 0%		Scalar	m=2 d,x=2,2 m=8	1% 0% 1%

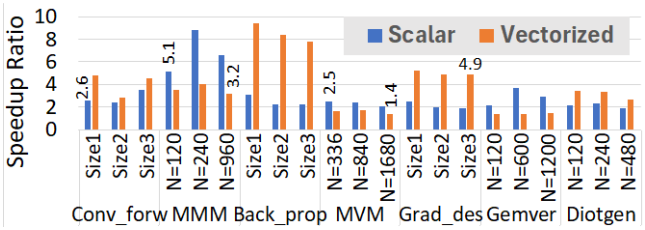

Figure 6: Evaluation over GCC (ARM)

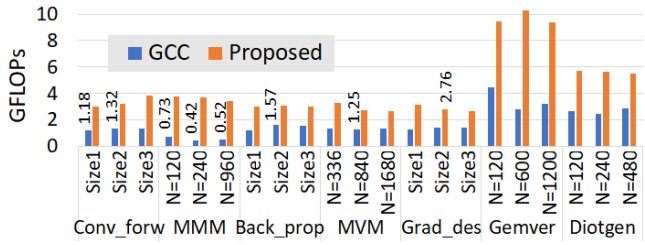

Figure 7: Evaluation over GCC for scalar kernels (ARM)

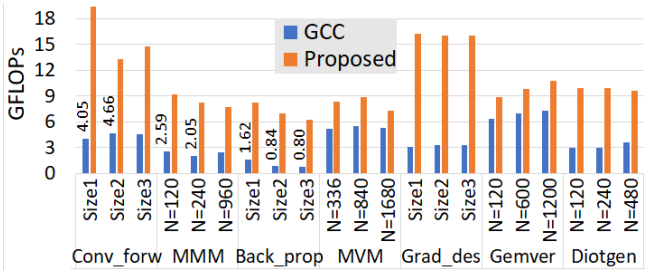

Figure 8: Evaluation over GCC for vectorized kernels (ARM)

and 3.9x for vectorized ones. Notably, significant speedups of almost 7x and 8.5x are observed in the scalar MMM and vectorized convolution back-propagation kernels, respectively. In MVM and Gemver kernels, lower speedups are observed because these kernels are less memory-bound, making the impact of RB less pronounced.

To further highlight the effectiveness of our methodology, Fig. 7 and Fig. 8 show the number of FLOPs (as absolute values) achieved by our tool (orange bars) and GCC (blue bars) for scalar and vectorized versions of the kernels. In general, Fig. 7 and Fig. 8 follow the same trends as in Fig. 6. ied kernels are compiled with the GCC

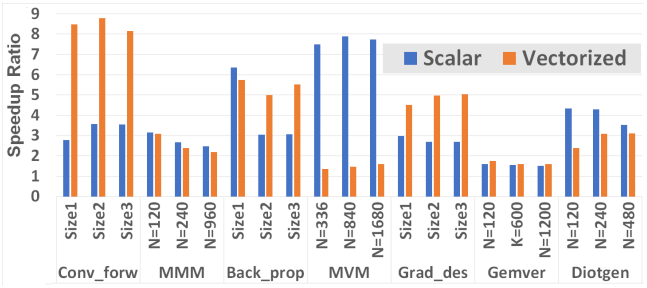

Figure 9: Evaluation over GCC (x64)

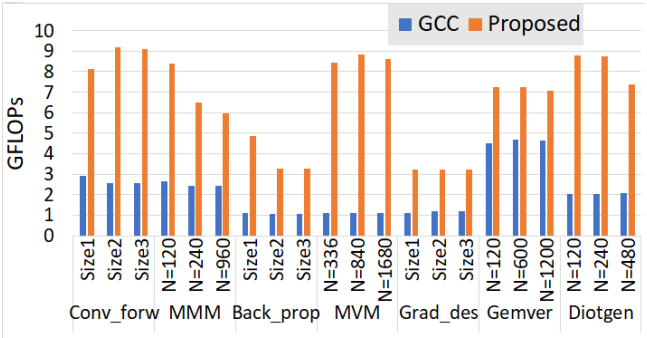

Figure 10: Evaluation over GCC for scalar kernels (x64)

compiler are 1.7 and 3.8 GFLOPs. Our approach attains average FLOP values of 4.5 and 10 for scalar and vectorized kernels, respectively, while the corresponding values with the GCC compiler are 1.7 and 3.8 GFLOPs for the seven studied kernels.

Evaluation over GCC on x64: In this part, we present our experimental results for the x64 platform. Fig. 9 depicts the speedup ratios achieved by our tool normalized to GCC. As Fig. 9 indicates, similar speedups are provided by our approach also in the x64 platform proving the strength of our methodology also in high-performance platforms. In particular, average speedups of 3.6x and 3.7x are reported for scalar and vectorized kernels. However, in this case, the higher speedups appear in the scalar MVM (7.7x) and vectorized version of ***forw.*** convolution kernel (8.5x), respectively. The latter behavior is mainly due to GCC limitation in applying RB to arrays that their upper bounds are not constant values.

The FLOP numbers for the x64 platform are shown in Fig. 10 (scalar versions) and Fig. 11 (vectorized versions). On scalar kernels,

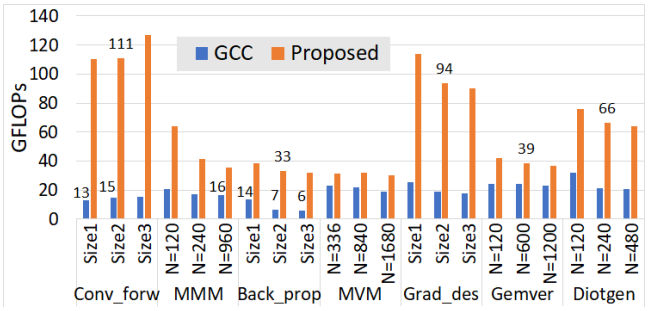

Figure 11: Evaluation over GCC for vectorized kernels (x64)

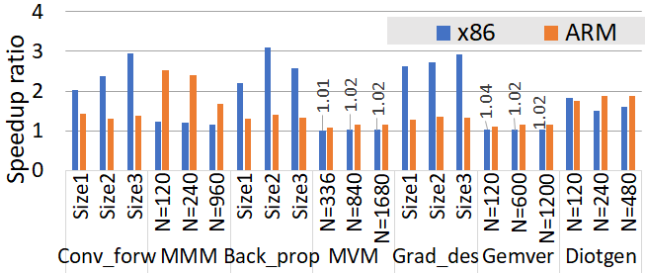

Figure 12: Evaluation over Pluto for scalar kernels

GCC achieves an average of 2.1 GFLOPs, while 6.8 GFLOPs are reported by our methodology. In the vectorized versions, the corresponding numbers are 18.8 and 64 GFLOPs, respectively. Obviously, x64 platform can achieve higher FLOPs compared to ARM, due to higher core frequencies and faster and larger memory subsystem.

Evaluation over Pluto tool: This part compares our methodology against Pluto tool. Note that Pluto automatically unrolls all loops (except the innermost) and the UF is specified by the user. We evaluate Pluto using the default unrolling factor (8) and for a fair comparison we disable the optimizations that are not supported by our tool (i.e., thread parallelism, loop tilling etc.). Pluto relies on GCC to apply scalar replacement to the unrolled code. Since Pluto cannot accept vectorized code, we only evaluate the scalar kernels.

Fig. 12 shows the speedups offered by our tool over Pluto on the x64 (blue bars) and ARM (orange bars) platforms. For kernels such as MVM and Gemver (in both platforms), our tool exhibits execution times similar to Pluto (up to 1.04x speedups). This stems from the fact that the static policy followed by Pluto happens to coincide with the RB configuration extracted by our tool. In the remaining cases, our methodology is superior compared to Pluto. Pluto's suboptimal performance stems from its static policy, which overlooks specific unroll combinations beneficial for certain kernels.

Evaluation over different padding ukernels: In this part, our focus is to reveal the importance of padding ukernels (Section 3). For illustrative reasons, we use the convolution kernel as a working example. Specifically, the unroll factors that the proposed methodology gives to m-loop (output channels) and x-loop (output width) are two and six. However, as the output width is not always evenly divisible by six, padding is required.

Fig. 13 shows the measured FLOPS of the studied kernel for various padding schemes and seven different input sizes i.e., output

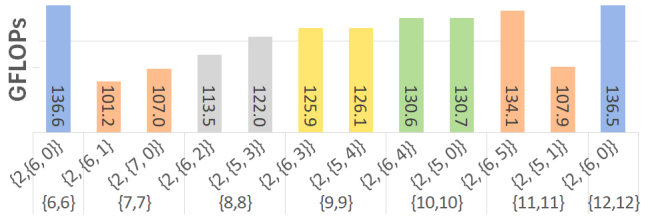

Figure 13: Evaluation for different padding ukernels for Conv_Forw (x64)

```
//Gradient Descent
for (b=0; b<Batch;b++)
  for ( m=0; m<ft_Map;m++)
    for ( y=0; y<out_he;y++)
      for ( x=0; x<out_wi;x++)
        for ( d=0; d<in_depth;d++)
din[b][y*stride.y][x*stride.x][d] +=
dout[b][y][x][m] * filter[m][d];

//Gemver
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
out[i][j] += u1[i] * v1[j] +
u2[i] * v2[j];

//Diogen
for (r = 0; r < N; r++)
  for (q = 0; q < N; q++)
    for (s = 0; s < N; s++)
      for (p = 0; p < N; p++)
out[r][q][p]+= A[r][q][s]*C[s][p];

//MMM
for (i=O;i!=N;i++)
  for (j=O;j!=N;j++)
    for (k=O;k!=N;k++)
C[i][j]+=A[i][k]*B[k][j];

//Back Propagation
for (b=0; b<Batch;b++)
  for ( m=0; m<ft_Map;m++)
    for ( y=0; y<out_he;y++)
      for ( x=0; x<out_wi;x++)
        for ( d=0; d<in_depth;d++)
dfilter[m][d] += dout[b][y][x][m]
*in[b] [y*stride.y][x*stride.x][d];

//MVM
for (i=O;i!=N;i++)
  for (j=O;j!=N;j++)
C[i]+=A[i][j]*B[j];
```

Figure 14: The initial source codes of the studied kernels

height (y) × width (x); shown at the bottom of x-axis. The triplets juxtaposed the x-axis are defined as {Rm, {Rx, Rp}}, where Rm and Rx are the RB factors of m and x loops, and Rp is the factor for the padding code. As Fig. 13 shows, exploring different padding schemes (as suggested in this work) can offer additional speedups e.g., more than 7% in the {7, 7}} case. This is because the {2, {7, 0}} ukernel results to a lower number of L/S instructions (in total), even if a few register spills occur in the main kernel.

7 CONCLUSIONS

This paper presents an analytical model for RB optimization and introduces a semi-automatic tool to streamline ukernel design in optimized libraries. The approach employs the number of L/S instructions as an objective function to generate RB factors and loop permutations that minimize this function. The methodology is open-sourced for integration into the Pluto tool. Experimental results demonstrate significant improvements, achieving an average 3x speedup over both GCC and Pluto across two platforms.

ACKNOWLEDGMENTS

This research has been supported by a sponsored research agreement between Applied Materials, Inc. and Aristotle University of Thessaloniki, Greece (Grant Agreement 72714).

APPENDIX

Fig. 14 shows the un-optimized source code of the studied kernels (the data types are in 32-bit floating-point format).

REFERENCES

[1] A. Acharya, U. Bondhugula, and A. Cohen. Effective Loop Fusion in Polyhedral Compilation Using Fusion Conflict Graphs. Transactions on Architecture and Code Optimization, 2020

[2] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. International Conference on Programming Language Design and Implementation, 2008

[3] S. Carr and K. Kennedy. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. Transactions on Programming Languages and Systems, 1994

[4] S. Carr and Y. Guan. Unroll-and-Jam Using Uniformly Generated Sets. International Symposium on Microarchitecture, 1997

[5] C. Chen, J. Chame, and M. Hall. Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy. International Symposium on Code Generation and Optimization, 2005

[6] E. Herruzo, G. Bandera, E.L. Zapata, and O. Plata. Reducing Cache Misses by Loop Reordering. International Conference on Parallel Computing, 2006

[7] V. Kelefouras and K. Djemame. A Methodology Correlating Code Optimizations with Data Memory Accesses, Execution Time, and Energy Consumption. Journal of Supercomputing, 2019

[8] V. Kelefouras and G. Keramidas. Design and Implementation of Deep Learning 2D Convolutions on Modern CPUs. Transactions on Parallel and Distributed Systems, 2023

[9] M. Kong and L.N. Pouchet. Model-Driven Transformations for Multi- and Many-Core CPUs. International Conference on Programming Language Design and Implementation, 2019

[10] M. Kong and L.N. Pouchet. A Performance Vocabulary for Affine Loop Transformations. arXiv preprint arXiv:1811.06043, 2018

[11] L. Lai, N. Suda, and V. Chandra. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. arXiv preprint arXiv:1801.06601, 2018

[12] J. Li, Z. Qin, Y. Mei, J. Cui, Y. Song, C. Chen, Y. Zhang, L. Du, X. Cheng, B. Jin, J. Ye, E. Lin, and D. Lavery. oneDNN Graph Compiler: A Hybrid Approach for High-Performance Deep Learning Compilation. arXiv preprint arXiv:2301.01333, 2023

[13] X. Liu, L. Ding, Y. Li, G. Chen, and J. Du. Research of Register Pressure Aware Loop Unrolling Optimizations for Compiler. MATEC Web of Conferences, 2018

[14] LLVM Compiler: https://github.com/LLVM/LLVM-project/issues/38004

[15] B. Meister, N. Vasilache, D. Wohlford, M. Baskaran, A. Leung, and R. Lethin. R-stream Compiler. In Encyclopedia of Parallel Computing, 2011

[16] Orio Tool: https://github.com/brnorris03/Orio

[17] Pluto Tool: https://Pluto-compiler.sourceforge.net/

[18] L.N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop Transformations: Convexity, Pruning and Optimization. International Symposium on Principles of Programming Languages, 2011

[19] Polly Tool: https://polly.LLVM.org/docs/UsingPollyWithClang.html

[20] Github url, Register Blocking Source-to-Source:https://github.com/Theoo1997/RB_s2s

[21] V. Sarkar. Optimized Unrolling of Nested Loops. Journal of Parallel Programming, 2001

[22] Valgrind Tool: https://valgrind.org/

[23] N. Vasilache, B. Meister, M. Baskaran, and R. Lethin. Joint Scheduling and Layout Optimization to Enable Multi-Level Vectorization. International Workshop on Polyhedral Compilation Techniques, 2012

[24] L. Wilkinson, K. Cheshmi, and M.M. Dehnavi. Register Tiling for Unstructured Sparsity in Neural Network Inference. International Conference on Programming Languages, 2023

[25] N. Tollenaere, G. Iooss, S. Pouget, H. Brunie, C. Guillon, A. Cohen, P. Sadayappan, F. Rastello. Autotuning Convolutions is Easier than you Think. ACM Transactions on Architecture and Code Optimization, 2023

[26] O. Zinenko, S. Verdoolaege, C. Reddy, J. Shirako, T. Grosser, V. Sarkar, and A. Cohen. Modeling the Conflicting Demands of Parallelism and Temporal/Spatial Locality in Affine Scheduling. International Conference on Compiler Construction, 2018