# DNestMap: Mapping Deeply-Nested Loops on Ultra-Low Power CGRAs

Manupa Karunaratne, Cheng Tan, Aditi Kulkarni, Tulika Mitra and Li-Shiuan Peh
{manupa,tancheng,aditi,tulika,peh}@comp.nus.edu.sg
National University of Singapore.

## ABSTRACT

Coarse-Grained Reconfigurable Arrays (CGRAs) provide high performance, energy-efficient execution of the innermost loops of an application. Most real-world applications, however, comprise of deeply-nested loops with complex and often irregular control flow structures that cannot be mapped to CGRAs by existing compilers. This leads to excessive data transfer costs as the execution continuously alternates between the outer loop-nests on the host processor and the innermost loop on the CGRA accelerator. Moreover, ultra-low power CGRAs can only include limited on-chip memory to cache the configuration bitstreams and need frequent swapping of configurations in the presence of multiple innermost loops. We introduce DNestMap, a partitioning and mapping tool for CGRAs, that can judiciously extract the most beneficial code segments of multiple deeply-nested loops and effectively cache them together statically in the configuration memory through spatio-temporal partitioning. DNestMap achieves 1.58X performance improvement compared to dynamic caching of configuration contexts of the innermost loops in the CGRAs with limited on-chip memory.

## 1 INTRODUCTION

CGRAs are gaining traction due to their superior energy efficiency over other forms of computation platforms, making them a good candidate for Internet of Things (IoT) and wearable devices. CGRAs offer a good balance of flexibility and efficiency compared to alternatives such as FPGAs and ASIC accelerators for computationally demanding workloads. A CGRA consists of an array of processing elements (PEs), each including an ALU, a register file, and a config. memory (*CMEM*) as shown in Fig 1. A PE is directly connected to its neighboring PEs. The PEs share a data memory that can be directly accessed by a subset of the PEs. The host processor handles data transfer to/from the data memory in the beginning/end of execution via DMA. CGRAs are ideal for acceleration of loop kernels. The operations (including memory operations) within a loop are scheduled on the PEs, while data flows are routed between dependent operations, all at compile time. This scheduling and routing information is stored in the on-chip *CMEM*.

Conventionally, the frequently executed segments of an application — the innermost loops — have been the target of CGRA acceleration. In recent literature, there have been several works that explore the mapping of loop-nests beyond the innermost level.

*DAC '18, June 24–29, 2018, San Francisco, CA, USA*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196027

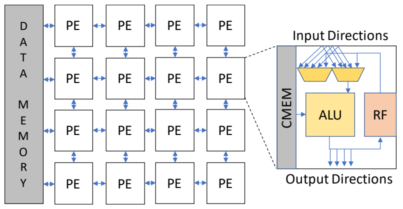

**Figure 1: A 4x4 CGRA Architecture**

For perfectly nested loops, [5] applies polyhedral based mapping techniques. For imperfectly nested loops, [6] proposes to flatten the loop hierarchy at the expense of increased code size for the kernel. [17] proposes a method to map two of the innermost levels of loop-nests using less *CMEM* compared to [6].

In applications with multiple loop-nests, these techniques are often limited by the number of loop-nests that can fit within the small *CMEM*, especially in ultra-low power CGRAs. A variety of low-power CGRAs have been proposed in recent years including HEAL-WEAR[8], Samsung Reconfigurable Processor[2], REMUS_LPP[9] and HyCUBE[12]. Table 1 shows that the on-chip *CMEM* is limited to only a few hundred bytes per PE. This is due to their highly constrained power budget — about 40% of the CGRA power is consumed by the on-chip *CMEM* in [12]. To solve this *CMEM* capacity constraint, recent works [4, 13, 16] propose architectural improvements to use the *CMEM* as a cache that stores the most recently accessed loop-nests at runtime. The dynamic caching leads to performance improvement because more segments of the application can be accelerated and the data transfer between the host and the CGRA is minimized. It is possible to naïvely employ caching even within a loop-nest to expand the mappable loop-nests beyond the innermost loops; but the frequent context switching between outer and inner loops may incur significant overhead.

We propose *DNestMap*, a CGRA mapping framework that selects the beneficial code segments across complex, deeply-nested loops for acceleration (Fig 2) and assigns each segment to a spatio-temporal partition of the aggregated conguration memory of the CGRA, while considering *CMEM* constraints and communication overheads. DNestMap segments the control-data flow graph of the program into mapping units that are either loops or non-loop code segments potentially mappable to the CGRA. It performs static and dynamic analysis of the data and context transfers between mapping units to determine the acceleration potential of each mapping unit, taking into account the communication costs.

**Table 1: Low-Power CGRAs**

| CGRA Arch. | Process | Freq.(Mhz) | Power(mW) | $CMEM_{perPE}$ |
|---|---|---|---|---|
| HyCUBE[12] | 40nm | 50 | 14.16 | 256 Bytes |
| HEAL-WEAR[8] | 65nm | 1 | N/A | 384 Bytes |
| ULP-SRP[2] | 40nm | 100 | 22.3 | N/A |
| REMUS_LPP[9] | 65nm | 75 | 23.75 | 560 Bytes |

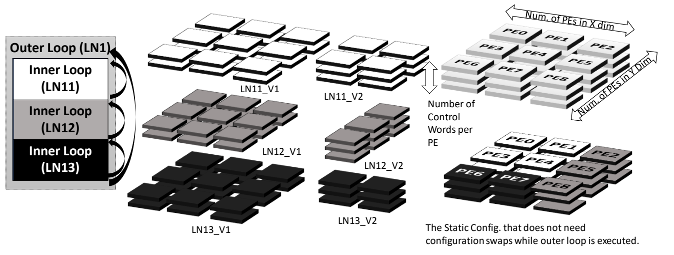

**Figure 2: DNestMap: Packing loop-nests in context memory**

Given limited *CMEM*, DNestMap selects and tightly packs a subset of the mapping units. It does so by formulating the mapping problem as a 3D orthogonal knapsack (3D-OKP) problem. *CMEM* is viewed as a 3D container where height and width are the dimensions of the PE-array (e.g., $4 \times 4$ array) and the depth is the number of configuration words available per PE. Each configuration word encodes the operation mapped to the PE and the routing information in a single cycle. Each mapping unit is treated as a box that needs to be packed inside the *CMEM*. The width and the height of the box indicate the number of PEs used by the mapping unit along X-Y dimensions and the depth represents the number of configuration words needed by the code segment. The goal is to pack a subset of the code segments (boxes) inside the *CMEM* (container) such that the execution time of the entire application is minimized. We propose a dynamic programming approach for the 3D-OKP problem that re-uses prior packing decisions to reach near-optimal mapping solutions.

DNestMap achieves 1.58X improvement in performance on average compared to the state-of-the-art where limited *CMEM* is used as a cache to map the innermost loops.

## 2 MOTIVATION OF DNESTMAP

The CGRA is used to accelerate the innermost loops in an application. The CGRA mapping algorithms [7, 11, 12] schedule the operations inside the loop body on the PEs and route the dataflow between dependent operations. The schedule and route per PE per cycle are encoded as a control word. The schedule length of the loop kernel, which is also the interval between successive loop iterations, is called the Initiation Interval (II). The goal of the mapping algorithms is to minimize the II so as to maximize throughput. Existing mapping approaches utilize all the PEs in the array to minimize the II value. That is, each mapped loop requires $XWidth \times YWidth \times II$ control words, where $XWidth$ and $YWidth$ are the number of PEs in the X-dimension and Y-dimensions, respectively.

Consider the loop-nest in Fig.2 comprising of one outer loop ($LN1$) and three inner loops ($LN11, LN12, LN13$). A conventional CGRA compiler will map each inner loop independently to minimize its individual II value, subject to resource and recurrence constraints. Let us assume that for each inner loop, the minimum possible $II = 2$ in this example utilizing all the PEs. However, the *CMEM* can only hold three control words. Thus all the inner loops cannot be stored together in the *CMEM*. If the *CMEM* is used as a cache, the control words corresponding to each inner loop needs to be swapped in from the off-chip storage before it starts execution. Assuming that the outer loop executes $N$ times, this will result in $3N$ context switches. As each control word is very long for the entire CGRA (for example, 8 x 16 = 128 Bytes for 4x4 HyCUBE CGRA), each context switch incurs significant overhead. We observe that the existing approaches focus on partitioning the *CMEM* among the loop kernels only along the temporal dimension (II dimension). Each loop still uses all the PEs ($LN11\_V1, LB12\_V1, LN13\_V1$). Thus the *CMEM* can accommodate a set of loops $\mathcal{L}$ if and only if $\sum_{l \in \mathcal{L}} II(l) \le T$, where $II(l)$ is the minimum II value for loop $l$ and $T$ is the number of control words available in the *CMEM*.

Clearly, these approaches miss out the opportunity of partitioning along the spatial (XY) dimension, that is, partitioning the PEs among the loop kernels in addition to the temporal dimension. As shown in Figure 2, each inner loop can also be mapped somewhat differently by utilizing only a subset of the PEs ($LN11\_V2$, $LB12\_V2$, $LN13\_V2$). As each loop uses limited PE resources, the II value increases from 2-cycles to 3-cycles. However, now the control words corresponding to all the inner loops can be packed together in the *CMEM* as shown in Figure 2. Thus the execution of the entire loop-nest will not require any context transfers.

The spatio-temporal partitioning of the *CMEM* as opposed to temporal partitioning leads to longer II value for each individual loop kernel. However, if the context transfer and the data transfer overhead of the temporal partitioning solution is higher than the performance gain due to the shorter II value, spatio-temporal partitioning will perform better for the overall application. *DNestMap* exploits this observation to explore the spatio-temporal partitioning of the code segments in *CMEM*. *DNestMap* enumerates the different spatial mapping choices for each loop kernel and their II values. It then intelligently searches through the combination of different choices for the loops and attempts to pack them together in the *CMEM* so as to maximize the application performance. Note that in conventional CGRAs, all the PEs participate in execution in each cycle, but *DNestMap* decouples this association. When a loop executes, only the PEs associated with the loop participates and the rest of the PEs are disabled by the control logic of the CGRA. In the next section, we present the formulation of the *DNestMap* mapping problem as 3D orthogonal packing problem.

## 3 PROBLEM FORMULATION

### 3.1 Mapping Units

We start with the segmentation of the application program into *Mapping Units (MUs)*, the granularity of mapping in our framework.

$Let\ \mathcal{L} = set\ of\ all\ loops$

$Def, child\ loop\ = immediate\ inner\ loop$

$Def, \psi : l_i \rightarrow \{l_{i1}, \ldots, l_{ip}\} \mid s.t. \{l_{i1}, \ldots, l_{ip}\} are\ child\ loops\ of\ l_i.$

$Def, \psi^{-1} : l_j \rightarrow l_i \mid s.t.\ l_j \in \psi(l_i),\ l_i\ is\ the\ parent\ of\ l_j.$

$Def, P_i = segments\ of\ l_i \in \mathcal{L}\ for\ \psi(l_i) \neq \emptyset \mid def, P : l_i \longrightarrow P_i$

$Def, k_i = kernel\ of\ l_i \in \mathcal{L}\ for\ \psi(l_i) = \emptyset \mid def, K : l_i \longrightarrow k_i$

$Def : MU\ can\ be\ any\ P(l_i)\ or\ K(l_i) \mid l_i \in \mathcal{L}$

The control data flow graph (CDFG) is partitioned into multiple MUs. A loop $l_i$ is an innermost loop if it does not have any child loop ($\psi(l_i) = \emptyset$). Similar, a loop $l_i$ is an outermost loop if it does not have a parent ($\psi^{-1}(l_i) \neq \emptyset$). In Fig 3, $l_2$ and $l_4$ are innermost loops, while $l_1$ is an outermost loop. $l_2$ and $l_3$ are the children of $l_1$.

An innermost loop $l_i$ comprises only of the kernel $k_i$. Any outer loop $l_i$ comprises of one or more segments appearing before (prologue) and/or after (epilogue) of each child loop. We define these set of segments for loop $l_i$ as $P_i$. The segments $P_i$ can be viewed as loops with single iterations. For example, loop $l_1$ has three such segments : $MU_1$,$MU_2$ and $MU_3$. For any outer loop $l_i$, a kernel $k_i$

does not exist and instead $\psi(l_i)$ represents the inner loops of the current loop $l_i$, that could be further expanded into segments and inner loops/kernels. We denote each of the segments and kernels as a MU. A MU is owned by the loop for which the MU is either the kernel or a segment. Based on the final decision of DNestMap, a subset of the loops will be mapped to the CGRA in terms of the MUs they own.

*3.1.1* ***Inter Mapping Unit Data Transfers :*** Each MU may produce data that needs to be accessed in other MUs. When mapping to the CGRA, it is important to take into account the possible data communication costs in choosing the optimal subset of MUs for the CGRA. All the possible data flow edges between MUs can be inferred through static analysis. We introduce edges connecting an MU that modifies a variable to all the other MUs that accesses the variable. Once all the edges have been added, we assign a weight to each edge based on the amount of data transfer along that edge. The amount of data transfer depends on the number of variables involved and the number of times the control flow transitions between the MUs. The latter is obtained through profiling.

From the profiling information, the number of times each loop is invoked can be calculated as $Inv(l) \mid l \in \mathcal{L}$ where $\mathcal{L}$ is the set of all loops. There are two variants of inter-MU data flow dependencies.

1) *Dataflow dependency along ancestry line:* Any data flow from an MU in an outer loop $l_a$ to an MU in any of its descendant loops $l_d$ is defined as dataflow dependency along ancestry line. For example in Fig.3, the dependency through the variable $var1$ from $MU_2$ to $MU_7$ is such dependency as $l_1$ is an ancestor of $l_4$ ($l_4 \in \psi(l_3) \wedge l_3 \in \psi(l_1)$). Such dependencies will be weighted (denoted by $w$ in Fig.3) according to the number of invocations of $\psi^{-1}(l_d)$ as the data will not be modified after $\psi^{-1}(l_d)$ (the loop $l_3$ in our example).

2) *Dataflow dependency outside ancestry line:* Any dataflow from an MU in loop $l_i$ to another MU in loop $l_j$ where there is no ancestor-descendant relationship between the two loops is outside the ancestry line. In this case, we need to find the closest common ancestor loop $l_p$ such that $l_p$ is an ancestor of both $l_i$ and $l_j$ and there does not exist any other common ancestor in the ancestry line $l_p \rightsquigarrow l_i$ and $l_p \rightsquigarrow l_j$. Let $l_{i'}$ and $l_{j'}$ be the children of $l_p$ along the ancestry lines $l_p \rightsquigarrow l_i$ and $l_p \rightsquigarrow l_j$, respectively. Then, the dependency between the MUs is weighted by the minimum of the number of invocations of $l_{i'}$ and $l_{j'}$, that is, $min(Inv(l_{i'}), Inv(l_{j'}))$. This is because data transfer is exercised by the minimum of the times the data is read or modified. Moreover, after the divergence, the variable is not modified. For example in Fig.3, the dependency through variable $var2$ from of $MU_7$, to $MU_4$ is such dependency as $l_3$ ($l_4 \in \psi(l_3)$) and $l_2$ are siblings. Thus, the edge would be weighted according to the minimum number of invocations of the two siblings where the line of ancestry diverges. That would be $l_3$ and $l_2$ and $min(12, 16) = 12$.

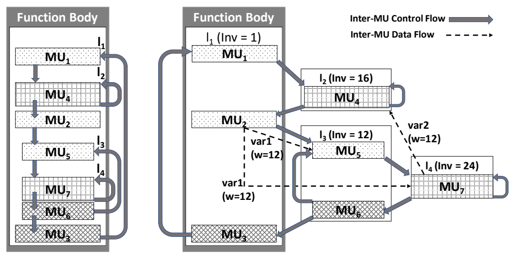

**Figure 3: MU partitioning and inter-MU data transfers.**

## 3.2 Mapping Problem

DNestMap chooses the best possible loop candidates in the deeply-nested code structure given limited *CMEM*. Each individual loop owns several mapping units as described in Section 3.1. Each MU can be mapped to a sub-CGRA with dimensions ranging from 1×1 to $XWdith \times YWidth$, where $XWidth$ and $YWidth$ are the number of PEs along X and Y dimension, respectively. Thus we can enumerate all possible mappings for an MU with different II values (different number of control words) depending on the sub-CGRA dimensions. We call each of these mapping unit schedule (MUS). Let us define $MUS_{i,x,y}$ as the mapping of the $i^{th}$ MU ($MU_i$) to a rectangular sub-CGRA with $x \times y$ dimension. $MUS_{i,x,y}$ can be visualized as a rectangular cuboid in the 3D space, created by time-extending the 2D sub-CGRA to the required number of control words. Similarly, the total configuration space (*CMEM*) could be visualized as a 3D container, created by time-extending the 2D CGRA to the available number of control words. For maximum acceleration, we need to fit the optimal subset of rectangular cuboids from the MUs (at most one per MU) inside the container.

Prior to finding the optimal subset of MUS, we need to solve the decision problem of whether a given set of MUS fit inside *CMEM*. The decision problem of whether such $MUS_{i,x,y}$ from different MU can fit in the *CMEM* becomes analogous to finding whether a set of boxes (with dimensions $(x_i, y_i, t_i)$) will fit in a container (with dimensions $x_C, y_C, t_C$). The naive approach would be to search for combinations of all possible placements of each box in the 3D space of the container, until a legal placement is found. This problem had been studied in the operations research community and the exact solution is discussed in [15]. An abstraction called **Packing Class**[14] is introduced where the search space is reduced to constructing three graphs with number of nodes equal to the cardinality of the set of boxes.

*3.2.1 A Packing Class.* Given a feasible packing, we can construct a set of graphs ($G_i = (V, E_i)$), one corresponding to each dimension (in our case $i = 0, 1, 2$). Each graph represents the projections of the packed boxes to the corresponding dimension. The nodes of the graphs $V$ represent the boxes ($v_n$). If the boxes overlap once projected to the $i^{th}$ dimension, we add an edge: $(v_{n1}, v_{n2}) \in E_i$. For example Fig 4. shows a packing of 2D boxes in a 2D container. When the boxes are projected towards the right (Y-axis), $G_2 = (V, E_2)$, shows the edges between boxes that are overlapping: $v_3, v_2, v_1$. Similarly, $G_1$ shows the overlap when projected to the X-axis. A set of graphs $G_i = (V, E_i)$ form a packing class, if the following properties are satisfied [14]

- P1 : $\overline{E_i}$ does not contain an odd 2-chordless cycle.[1]
- P2 : $E_i$ does not contain chordless cycle of length=4

[1]A chord is an edge joining two nonadjacent nodes in a cycle. A 2-chord is a chord that have the two nonadjacent nodes only separated by a single node in the cycle.

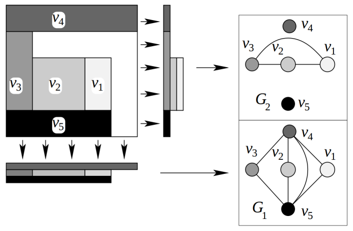

**Figure 4: An example packing class for a 2D container [14]**

- P3 : $\overline{E_i}$ does not contain a clique $S$, such that the summation of lengths of $v \in S$ in the $i^{th}$ dimension is less than the length the container in the $i$ dimension.
- P4 : $\cap_i E_i = \emptyset$

As a packing class abstracts many possible ways of packing, the search space is reduced to just finding the $G_i$s that satisfy the above properties. [15] proposes to use a branch and bound solution with strong pruning conditions, aimed to construct $G_i$s starting with edge-less graphs and then, iteratively add edges until the above properties are satisfied. This is called the decision problem of Orthogonal Packing Problem of d-dimensions (OPP-d). This problem is known to be NP-Complete in a strict sense. However, due to the packing class abstraction, the complexity is now in terms of the edges in the projected graphs as opposed to all possible placements in the 3D space. We develop a modified version of constructing the packing class for the CGRA mapping problem.

**Packing Class for Mapping Unit Schedule :** When solving the Orthogonal Packing Problem of 3-Dimension[14] (OPP-3D) for the case of Mapping Unit Schedules (MUS), there is a slight difference that the packing class is allowed to have chord-less cycle of length 4 in the time dimension. For example Fig.5 illustrates a valid packing that has a chord-less cycle of length 4 in the projected graph along the time dimension. This is because only one MU executes in the CGRA fabric at any point in time and its schedule is repeated (modulo) after II cycles, allowing it to initiate from a larger address and wrap back to the beginning of the *CMEM* ($v_4$ in Fig.5).

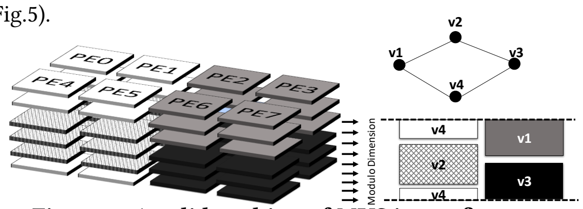

**Figure 5: A valid packing of MUS in config. memory**

## 3.3 3D Orthogonal Knapsack Problem with Synergistic Selection (3D-OKP-SS)

The CGRA mapping problem aims to select a subset of the MU $S_{MU}$ and an MUS version $MUS_{(i,x,y)}$ for each MU $m_i \in S_{MU}$ such that the selected combination of MUS fit within the *CMEM*. However, we need to take into account the data communication cost when the control flows from an MU chosen to be implemented in the CGRA to an MU implemented in the host and vice-versa. The bytes to be transferred between the CGRA and the host processor is calculated, by performing summation of the weights on the edges that cross the boundary between the CGRA and the host. As discussed in Section 3.1.1, the edges are weighted by the variables that need to be transferred and the invocation frequency of the transfer. In order to avoid double counting, they are enumerated using a tuple that consists of the source mapping unit where the data is created, the loop in the boundary and the variable identifier. The cost is calculated via weighted (invocation weight) summation of the sizes of the variables.

Now the problem is to find the optimal set $S_{MU}$ that maximizes the Savings (Eq.1), while $\forall m \in S_{MU}$ can form a valid packing class (by choosing appropriate MUS version for each MU), as described in Section 3.2.1. This problem is very similar to the 3D-Orthogonal Knapsack Problem (3D-OKP), except that the selected MUs may contribute to synergistic gain or loss depending on the communication cost. Thus, we define the problem as 3D-Orthogonal Knapsack Problem with Synergistic Selection (3D-OKP-SS) that is also NP-Complete.

$$Savings(S_{MU}) = HostCycles(S_{MU}) - CGRACycles(S_{MU}) - CommCost(S_{MU}) \quad (1)$$

**Algorithm 1:** 3D-OKP-SS - DNestMap Algorithm

```
P(l) = {child loops of l}
Function PopulatePotLp($L_+$, $L_-$)
    foreach $l \in L$ do
        if $P(l) \cap L_+ == P(l)$ and $l \cap L_+ == \emptyset$ and $l \cap L_- == \emptyset$ then $PotL := l$ ;
    end
    return PotL;
si = init;   SI.push(si);
while $SI \neq \emptyset$ do
    Select $si \in SI$; $SI = SI \setminus si$;
    (si.OPP3DInfo,result) = CheckPacking (si.MU+);
    if $result == sucess$ then UpdateSavings(si.MU+,si.OPP3DInfo) ;
    else continue ;
    si.PotL = PopulatePotLp($si.L_+$, $si.L_-$);
    if $si.PotL == \emptyset$ then continue;;
    Select $nL \in si.PotL$;   $si.PotL = si.PotL \setminus nL$;
    MUS_Combos = AllMUSCombos(nL);
    lets denote $< MUS >_p > \in MU\_Combos \; \forall p$;
    $< MUS >_a = argmax_p \; CalcGainDensity(< MUS >_p)$;
    $< MUS >_b = argmin_p \; CalcCGRACycles(< MUS >_p)$;
    $siN_1 = si$;      $siN_2 = si$;      $siN_3 = si$;
    $siN_1.L_+ := nL$;   $siN_2.L_+ := nL$;   $siN_3.L_- := nL$;
    $siN_1.MU+ := < MUS >_a$;   $siN_2.MU+ := < MUS >_b$;
    $SI.push(siN_1)$; $SI.push(siN_2)$; $SI.push(siN_3)$;
end
```

## 4 DNESTMAP ALGORITHM

We now present the DNestMap algorithm (Alg.1) to find the optimal subset of MUs ($S_{MU}$) using a branch-and-bound (B&B) search with efficient pruning. The original OPP-3D algorithm [15] is adapted for our purposes. The main change is that the success criterion is changed to find the packing class for MUS (Section 3.2.1). At a high-level, DNestMap algorithm is a 2-level B&B algorithm. The top level B&B is for the selection of the set of Mapping Units Schedules ($MUS = M_{i,x,y}$) to be included in the CGRA. At each search node in the top-level B&B tree, a decision on which MUS to be included is already taken. Based on that decision, the low-level B&B is invoked to test whether the given set of MUS can fit in the container. The low-level B&B is a building block for the top-level B&B algorithm.

**Low-Level B&B :** Given a set of MUS, finding whether they will fit inside *CMEM* starts with constructing the projected graphs for each dimensions : $G = \{G_x, G_y, G_t\}$ that represent overlaps in the corresponding dimension, similar to the exact solution provided for OPP-d [15]. Initially, all pairs of MUS s.t. their sum of widths of $x, y, t$ dimensions exceed the width of the container (*CMEM*) are included to the projected graph. Then, the low-level B&B will try to add edges in projected graphs until the properties for packing class for MUS is satisfied (Section 3.2.1) or prune the branch when

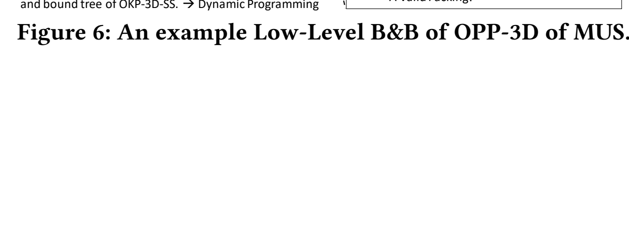

**Figure 6: An example Low-Level B&B of OPP-3D of MUS.**

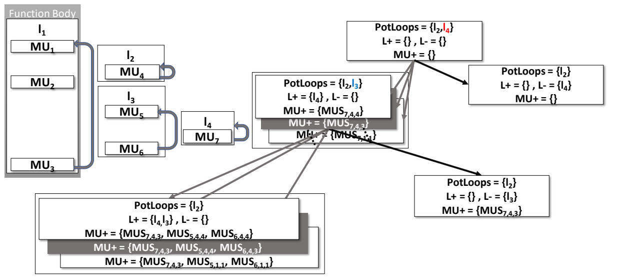

**Figure 7: An example Top-Level B&B tree of 3D-OKP-SS.**

the existence of such packing class becomes impossible based on the considered edges [15].

**Top-Level B&B :** This is the main B&B algorithm that uses the low-level B&B as a building block. Initially, the MUSs belonging to the innermost loops are selected. Based on the outcome of the low-level B&B (i.e., whether the packing is successful), the current branch will either proceed or be terminated. If the current packing is successful, the search will move forward to include any possible parent loop : $l_p$ to the list potential loops ($PotL$) that could mapped in the future (using $PopulatePotLp$ - line 3-10). A loop $l_i$ is selected out of the potential loops ($PotL$) and explored based on the following two characteristics:

$$GainDensity = \frac{Savings(MU_1, \ldots, MU_n)}{\sum_{MU_{i,x,y}} Volume(MU_{i,x,y})}$$

$$CGRACycles = \sum_{MU_{i,x,y}} CyclesCGRA[MU_{i,x,y}]$$

We will include the set of MUS that provide the maximum *GainDensity* and the set of MUS that provide the minimum CGRA cycles. Based on our experiments, the maximum *GainDensity* strategy works well for application with larger number of loops and the minimum *CGRACyles* works well for applications with lesser number of loops. In order to avoid re-explorations of the low-level B&B search in the child nodes of the current search node of the top-level B&B, the already explored projected graphs are forwarded to the child nodes, exploiting memoization. An example is shown in Fig.7. At every successful packing, maximum savings of cycles and the set of MUS are stored, for it to be a possible end solution.

## 5 EVALUATION

In this section, we evaluate the DNestMap framework for any generic ultra-low power CGRAs with limited *CMEM*.

Fig.8 illustrates the overview of the framework. The target source code is first partitioned into MUs using the $MappingUnitExtraction$. We implement this functionality based on LLVM 3.9[3] and generate two outputs: (1) LLVM-IR segments for each of the MU that can be passed on to the CGRA compiler, and (2) Instrumented Binary to capture the execution details in the granularity of MU and basic blocks in the host processor. The CGRA compiler (specific to the target CGRA) generates all possible Mapping Unit Schedules (MUS with dimensions: 1x2,...,4x4 etc. depending on the PE array size) for each MU. Note that in reality, the CGRA compiler is invoked on a need to know basis as the mapping algorithm explores the different MUs and it is not necessary to generate all possible MUS a-priori. The execution of the instrumented binary provides the number of host cycles to execute an MU. Using these inputs, DNestMap Alg.1 finds the optimal subset of MUS that provides the maximum performance benefit. The actual performance benefit is calculated by executing the partitioned application with the chosen MUS on the CGRA and the remaining MUs on the host processor.

In order to obtain concrete performance results, we use ARM-Cortex M3 (common in wearable platforms, running at 48 MHz) as the host processor and our previously designed $4 \times 4$ HyCUBE[12] as the CGRA. The communication interface between the host and the CGRA is modelled as a standard AMBA-AHB[1] connection that could transfer 32bits per cycle for transferring array variables and 16bits per cycle for transferring scalar variables (address and data alternate every cycle). HyCUBE can achieve 4X lower II compared to the regular CGRAs for the same kernel due to the single-cycle, multi-hop communication between distant PEs and hence impose less demand on the *CMEM* due to the lower II. Thus, if DNestMap framework shows performance improvement with HyCUBE, the improvement will be more pronounced with regular CGRAs that require more *CMEM* per kernel. We modified the architecture by extending the data memory connectivity to all the PEs in a given row (with negligible area overhead), such that any subset of the PEs that gets activated can use the memory as if it was a sub-CGRA. As only one MU is activated at a given point of time, there is no conflict for data memory access among the MUs.

We use typical wearable applications that demand an energy-efficient execution platform for our evaluation: (1) FFT : is essential to all sensory data computations and is typically used in applications like gesture recognition. (2) AES: Encryption is a critical function that guarantees security of personal data collected. (3) DCT: is used for general compression of natural data collected. (4) SVD: is the ideal algorithm for characterising features of wearable sensory data.

We compare DNestMap with existing approaches: (1) **CacheInner1:** Most CGRA compilers[7, 11, 12] target the innermost loops. More recent works [4, 13, 16] have used the *CMEM* as a cache to store the configuration context of recent innermost loops. This baseline evaluates the performance when the innermost loops are dynamically cached in the *CMEM* using perfect LRU policy. (2) **CacheInner2:** Some recent works[6, 17] consider one level beyond the innermost loops to achieve better performance. To evaluate potential benefits, we combine this with caching to mitigate the *CMEM* constraints, similar to the previous baseline. (3) **StaticTemporal:** This is a static caching approach that partitions the *CMEM* along the temporal axis by placing the most executed MUs mapped to full CGRA dimensions of 4x4. This approach is used to understand the tradeoff between dynamic caching *CacheInner1, CacheInner1* versus static caching and to evaluate the impact of spatio-temporal partitioning in the DNestMap approach compared to a pure temporal partitioning technique.

We consider four different *CMEM* sizes. Table 2 shows the *CMEM* power compared to the total power of the CGRA for HyCUBE running at 50MHz. The *CMEM* power is estimated using CACTI 6.5 [10] while the rest of the chip power is derived from RTL synthesis and layout (onto 40nm, using Synopsys DC for synthesis and Cadence Encounter for P&R) results. Clearly, *CMEM* is a significant contributor to the CGRA power. Fig.9 shows the accelerated execution

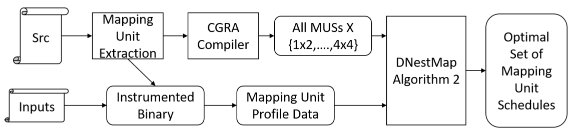
Src
Mapping Unit Extraction
CGRA Compiler
All MUSs X {1x2,....,4x4}
DNestMap Algorithm 2
Optimal Set of Mapping Unit Schedules
Inputs
Instrumented Binary
Mapping Unit Profile Data

**Figure 8: The DNestMap Framework.**

time breakdown (in terms of cycles taken for segments executed on host, CGRA, data and context communications) as a fraction of the execution time completely on the host processor. Table.2 shows the performance improvements (averaged across all kernels) when executed on CGRAs with different *CMEM* size. On average, DNestMap is 1.58X faster compared to *CacheInner1* and 1.7X faster compared to *CacheInner2*. With limited *CMEM*, dynamically caching 2-levels of innermost loops can result in overall performance degradation due to frequent cache replacements compared to *CacheInner1*, as majority of their execution time is spent on context swaps as shown in Fig.9. Moreover, *StaticTemporal* performs better than dynamic caching with smaller *CMEM*, exhibiting the cost of configuration replacements. However, *DNestMap* delivers 30% better performance than *StaticTemporal* due to the superior spatio-temporal packing. For *FFT*, DNestMap performs worse than *CacheInner2* and *CacheInner1* with 2KB and 1KB *CMEM*. *FFT* contains two nested loops and their execution is one after the other. Thus the cache is replaced only once when the control flows from the first nested loop to the next. This dual phase behaviour enables the mapping of the loop kernels using all the PEs for each nested loop and therefore performs slightly better than *DNestMap* where it attempts to pack-in more loops statically. The *SVD* kernel consists of 58 MUs and some MUs are considerably larger than the rest of the kernels. Therefore the two dynamic caching approaches suffer significantly (even slowing down the execution) when they are mapped indiscriminately. *DNestMap* is able to select the most beneficial MUs and gain an average performance improvement of 50% for *SVD*.

The three baselines use all the PEs of CGRA for each of the MUs. Moreover, power gating of the PEs that are not used seems to be impractical on a cycle-by-cycle basis (due to delays involved in changing the states of SRAM : deep-sleep to active). Most of the PEs at least have routing configuration in the control word if not for an operation. However, DNestMap schedules the MUs on a subset of PEs and gurantees that the rest of the PEs could be power gated

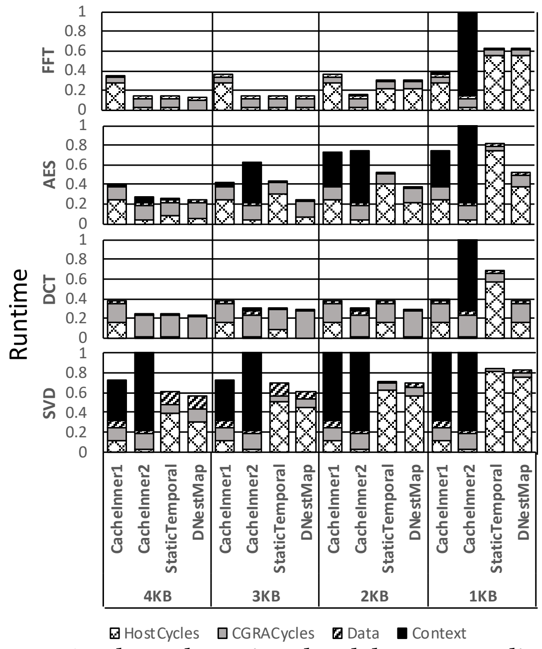

**Figure 9: Accelerated runtime breakdown normalized to host-only runtime**

**Table 2: Power and performance characteristics**

| CMEM Size | HyCUBE Power | | Speedup w.r.t host-only execution | | | |
|---|---|---|---|---|---|---|
| | CMEM | Total | Cache Inner1 | Cache Inner2 | Static Temporal | DNest Map |
| 4KB | 4.7 mW | 11.76 mW | 2.3X | 3.99X | 3.57X | 4.5X |
| 3KB | 3.6 mW | 10.66 mW | 2.24X | 3.05X | 2.87X | 4.03X |
| 2KB | 2.5 mW | 9.56 mW | 1.86X | 2.78X | 2.11X | 2.68X |
| 1KB | 1.4 mW | 8.46 mW | 1.60X | 0.54X | 1.34X | 1.84X |

during the execution of the MUS. Therefore, DNestMap can achieve 38.54% reduction in total energy consumption, on an average across the benchmarks (-31.80% vs StaticTemporal, -55.42% vs CacheInner1 and -28.39% vs CacheInner2) for the same *CMEM* size. Moreover, on an average DNestMap achieves 4.11X performance-per-watt compared to host-only execution for *CMEM* size of 4KB.

## 6 ACKNOWLEDGMENTS

This work was partially funded by the Singapore Ministry of Education Academic Research Fund Tier 2 MOE2014-T2-2-129 and by the Singapore National Research Foundation : NRF-RSS2016-005.

## 7 CONCLUSION

CGRAs provide high performance, energy-efficient execution of the innermost loops of an application, making them good candidates for IoT and wearable devices. However, most application kernels comprise of deeply-nested loop structures with inter-loop data communications. We proposed DNestMap, a partitioning and mapping tool for ultra-low power CGRAs with limited on-chip configuration memory, that judiciously extracts the most beneficial code segments from a deeply-nested loop structure. DNestMap achieves 1.58X performance improvement compared to dynamic caching of configuration contexts of the innermost loops on CGRAs with limited on-chip memory.

## REFERENCES

[1] 2017. ARM-Cortex M3 Technical Reference. https://goo.gl/eyKE1u. (2017).
[2] C. Kim et al. '12. ULP-SRP: Ultra low power Samsung Reconfigurable Processor for biomedical applications. In *FPT'12*.
[3] C. Lattner et al. '04. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO'04*.
[4] C. Yang et al. '16. Data cache prefetching via context directed pattern matching for coarse-grained reconfigurable arrays. In *DAC'16*.
[5] D. Liu et al. '13. Polyhedral model based mapping optimization of loop nests for CGRAs. In *DAC'13*.
[6] J. Lee et al. '14. Flattening-based mapping of imperfect loop nests for cgras?. In *CODES+ISSS'14*.
[7] L. Chen et al. '14. Graph minor approach for application mapping on cgras. In *TRETS'14*.
[8] L. Duch et al. '17. HEAL-WEAR: An Ultra-Low Power Heterogeneous System for Bio-Signal Analysis. *IEEE Transactions on Circuits and Systems I: RP'17*.
[9] L. Liu et al. '15. An energy-efficient coarse-grained reconfigurable processing unit for multiple-standard video decoding. *IEEE Transactions on Multimedia'15*.
[10] Muralimanohar et al. '09. CACTI 6.0: A tool to model large caches. *HP Labs'09*.
[11] M Hamzeh et al. '13. REGIMap: register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In *DAC'13*.
[12] M. Karunaratne et al. '17. HyCUBE: A CGRA with Reconfigurable Single-cycle Multi-hop Interconnect. In *DAC'17*.
[13] P. Cao et al. '17. Context Management Scheme Optimization of Coarse-Grained Reconfigurable Architecture for Multimedia Applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems'17*.
[14] SP Fekete et al. '04. A combinatorial characterization of higher-dimensional orthogonal packing. *Mathematics of Operations Research'04*.
[15] SP Fekete et al. '07. An exact algorithm for higher-dimensional orthogonal packing. *Operations Research'07*.
[16] SMAH Jafri et al. '16. Polymorphic configuration architecture for CGRAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems'16*.
[17] S. Yin et al. '16. Exploiting Parallelism of Imperfect Nested Loops on Coarse-Grained Reconfigurable Architectures. *TPDS'16*.