Energy-Aware Tile Size Selection for Affine Programs on GPUs

Malith Jayaweera*, Martin Kong†, Yanzhi Wang‡, David Kaeli§
*Department of Electrical and Computer Engineering**‡§, *Northeastern University*, Boston, MA, USA
Department of Computer Science and Engineering†, *The Ohio State University*, Columbus, OH, USA
{malithjayaweera.d, yanz.wang}@northeastern.edu*‡, moreno.244@osu.edu†, kaeli@ece.neu.edu§

***Abstract*—Loop tiling is a high-order transformation used to increase data locality and performance. While previous work has considered its application to several domains and architectures, its potential impact on energy efficiency has been largely ignored. In this work, we present an Energy-Aware Tile Size Selection Scheme (EATSS) for affine programs targeting GPUs. We automatically derive non-linear integer formulations for affine programs and use the Z3 solver to find effective tile sizes that meet architectural resource constraints, while maximizing performance and minimizing energy consumption. Our approach builds on the insight that reducing the liveness of in-cache data, together with exploiting automatic power scaling, can lead to substantial gains in performance and energy efficiency. We evaluate EATSS on NVIDIA Xavier and GA100 GPUs, and report median performance-per-Watt improvement relative to PPCG on several affine kernels. On Polybench kernels, we achieve 1.5× and 1.2× improvement and obtain up to 6.3× improvement on non-Polybench high-dimensional affine kernels.**

***Index Terms*—loop tiling, energy optimization, affine transformations, GPUs**

I. Introduction

Loop tiling is a critical transformation used to exploit spatial and temporal locality, while also exposing parallelization opportunities. It has been effectively used to improve cache utilization [1]–[3] in a wide variety of architectures, including CPUs [4], GPUs [5], [6] and FPGAs [7], [8].

Finding tile sizes that maximize performance, i.e., floating point operations per second, has traditionally been the driving objective. However, the increase in use of GPU-based heterogeneous platforms also adds new challenges when trying to maximize performance per unit of energy [9], [10]. There has been a growing use of the *performance-per-Watt* (PPW) as a key performance indicator by CPU and GPU vendors [11]. Unlike older GPUs, recent designs are more energy-optimized and use dynamic voltage and frequency scaling (DVFS) to manage power consumption [12].

Power consumption can vary dramatically on a GPU, based on the problem size. As an example, in Fig. 1 we show the power consumption of a gemm kernel run on a GA100 GPU. As we increase the size of the problem (increasing M, N, and K), due to the use of more multiprocessors and the generation of additional memory traffic, we find that the total power consumption tracks the dynamic and static power components, as opposed to remaining constant [12]. As we increase M, N, and K from 2,000 to 6,000, the total power is dominated by dynamic power. Dynamic power depends on GPU utilization (Streaming Multiprocessors / cache), which in turn depends on tile size choices. Therefore, there is the potential to improve energy usage through the proper selection of the tile size.

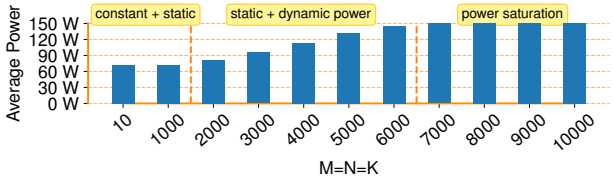


Fig. 1: Power consumption of GEMM kernel on GA100 across increasing problem sizes. At smaller scale constant + static power dominates the total power. Then, as sizes increase, the total power is dominated by static + dynamic power.

There is also a high demand from large-scale software service providers [13] to improve the energy efficiency of their systems to minimize the carbon footprint of computationally expensive deep learning pipelines [14]. While there have been efforts to conserve energy in server environments, these studies have focused mainly on server power capping [15], [16] or reducing Dynamic Voltage Frequency Scaling (DVFS) interference [17]–[20]. Adopting these approaches typically requires granting administrative access to the computing environment; an impossible privilege in shared cloud environments due to security concerns.

In this paper, we introduce a new *tile size selection scheme*, EATSS, that optimizes both power/energy efficiency and for performance of affine programs on GPUs, and is based on a novel non-linear integer formulation. We use the Z3 solver [21], [22] to find effective tile sizes based on predictive energy modeling. We incorporate several critical resource constraints, GPU execution model features, internal knowledge of affine scheduling, and insights from previous work [23], resulting in a simple, fast, and accurate model. A distinctive feature of our work is the ability to explore the trade-off of using varying degrees of shared memory and L1 / L2 per streaming multiprocessor (SM), and how to select the SIMT length as a fraction of the *warp size*, two factors known to strongly impact the energy budget on GPUs. The tile sizes found are fed to the PPCG [24] compiler to produce efficient CUDA code.

In summary, we make the following contributions: i) We develop, to the best of our knowledge, the first energy-aware tile-size selection scheme for affine programs on GPUs. ii) We

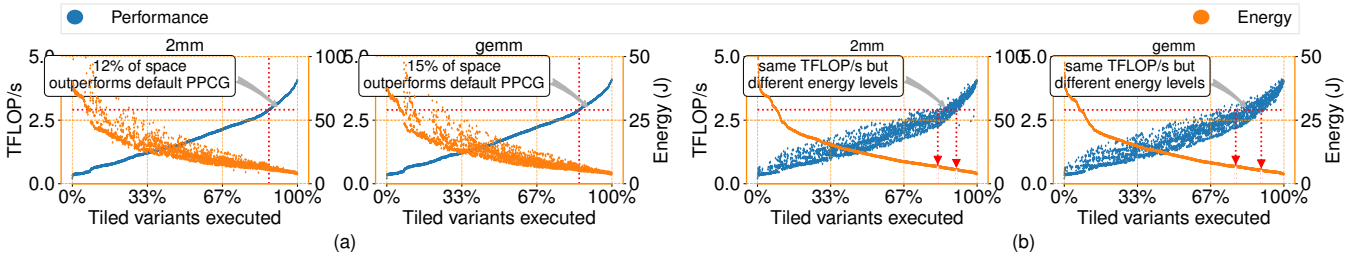


Fig. 2: Performance and energy distribution of 3,375 tiled variants on a GA100, N=4000. Left Y-axis is TFLOP/s, and right Y-axis is energy (J). The baseline (PPCG) is shown as a horizontal line. (a) Tile sizes are sorted by performance. (b) Tile sizes are sorted by energy.

conduct an in-depth study demonstrating the untapped performance and energy space, which was not addressed in previous work. iii) We demonstrate that our energy-aware formulation is capable of improving both performance (FLOPS) and energy. iv) We conduct an extensive evaluation on an NVIDIA Ampere (GA100) and Xavier, showing the effectiveness of our approach on several widely used affine kernels.

The remainder of this paper is organized as follows. Sec. II discusses the motivation for our work. Sec. III reviews several background concepts. Sec. IV introduces the design of our energy-aware tile size selection formulation for affine programs on GPUs. We then present our extensive evaluation in Sec. V, and conclude the paper with the related work (Sec. VI) and provide closing remarks in Sec. VII.

II. Motivation

The premise of our work is that a judicious selection of tile sizes can deliver not only strong/high performance, but also energy efficiency. We observe that there is a *reuse trade-off* overlooked by previous approaches. To illustrate this trade-off, we conduct an extensive exploration of tile sizes of the *2mm* kernel (two back-to-back MatMul), involving 3,375 variants, and evaluate their performance and energy-savings on the GA100 GPU. Fig. 3 shows the performance and energy metrics for *2mm*. We note the room for performance improvement (around 30%), and energy savings (around 20%) between both platforms. Our findings suggest we can either improve the kernel performance, its energy efficiency, or achieve both objectives. Upon closer inspection of Fig. 2(a), we see that only a few variants improve the default tiling values used by PPCG [24] (i.e., only 12% of the variants for *2mm* and 15% for gemm on an GA100). Thus, the challenge is to find a good tiling configuration within this large tile-size space of program candidates.

Generally, it is assumed that higher performance leads to lower energy consumption due to faster execution times. This is often the case, but it is not the only scenario where energy is reduced. The energy reduction depends on both the reduction in power and the run-time. Therefore, if the power is constant (regardless of the execution schedule), the energy should be directly proportional to the execution time. However, our observations in Fig. 1, as well as Fig. 2, show that the power is variable and depends on the size of the problem

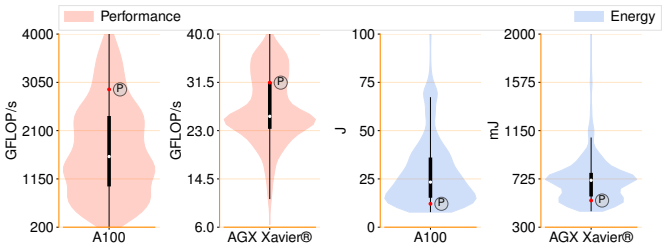


Fig. 3: Performance and energy distribution of `2mm` tile-space, on GA100 and Xavier. Location of default PPCG [24] performance and energy marked with P.

and the tiling strategy. In fact, Fig. 2(b) shows that there are tile sizes that deliver the same level of performance as the default configuration of PPCG [24], while differing in energy consumption. **Our insight motivates us to incorporate energy modeling as a primary objective in tile size selection engines**.

III. Background

GPU Architecture: GPUs adopt a Single-Instruction Multiple-Thread (SIMT) programming model. GPU programming presents an abstraction of a 3D-parallel architecture. At the top level, GPUs consist of a set of in-order *Streaming Multiprocessors (SM)*. A computation offloaded to a GPU is decomposed into a grid of two dimensions. The outer dimension of the grid specifies the number of *thread blocks*, whereas the inner dimension specifies the number of threads mapped to each thread block. SMs execute thread blocks at the granularity of *warps*, normally 32 threads. All threads in a warp execute the same instruction on different operands. The memory hierarchy of modern GPUs consists of registers, SM-level L1 caches, `shared memory` (accessible to all threads in the SM and managed by software), a shared L2 cache across the GPU, and global memory (DRAM). Coalesced Memory Access (CMA), a special type of efficient global memory access, is achieved when threads in a warp (which run in parallel) access contiguous elements in memory, effectively reducing the number of memory transactions (relative to the number of threads issuing the request). Threads, thread blocks, and SMs have limits on the number of registers that can be used. The more registers a thread uses, the fewer threads that can fit into a thread block. Similarly, the more registers used by a

TABLE I: GPU-specific (GA100) input parameters to model.

Abbreviation	Description	Example	Abbreviation	Description	Example
T_P_B	Threads per Thread-Block	1024	R_P_T	Registers per Thread	255
T_P_W	Threads per Warp	32	$L1_{SH}$	L1 + Shared Memory	192KB
R_P_S	Registers per SM	64K	L2	L2 Memory	40MB
R_P_B	Registers per Thread-Block	64K			

thread block, the fewer warps that can run concurrently within a single SM, limiting parallelism.

Loop Tiling: Fig. 4 shows the tiled `matmul` kernel using tile sizes 32, 64 and 16. Tile sizes control the granularity of the parallel loops, and also the data footprint accessed within each atomic execution of a tiled loop (see the three innermost loops in Fig. 4). In general, larger tile sizes improve locality (both intra- and inter- thread level), but also reduce the number of outer-parallel iterations, the ones that traverse the tiles.

Polyhedral GPU Compilers In this work, we use the *Polyhedral Parallel Code Generator* (PPCG) [24] to apply *fixed tiling* and to produce exploratory spaces consisting of hundreds of tiled variants. PPCG uses the parallelism exposed by tiling to map loops onto GPUs, while also permitting the user to enable/disable the usage of shared memory, and allowing to provide customized shared-memory budgets.

```
for(ti=0; ti<M; ti+=32) // ti maps to BID.y
 for(tj=0; tj<N; tj+=64) // tj maps to BID.x
  for(tk=0; tk<P; tk+=16) //
  //-------- Intra Tile Execution -----------
  for(i=ti;i<min(M,ti+32);i++)// i maps to TID.y
   for(j=tj;j<min(N,tj+64);j++)// j maps to TID.x
    for(k=tk;k<min(P,tk+16);k++)// Intra-thread reuse
      Out[i][j] += In[i][k] * Ker[k][j];
```

Fig. 4: Tiled `matmul` with 32, 64, 16. Standard CUDA variables abbreviated as BID (blockIdx), TID (threadIdx).

IV. Energy Aware Tile Size Selection

A. Overview

Our approach uses properties of the computation, together with the architectural characteristics and resource constraints of the GPUs, to optimize the efficiency and performance of energy / power simultaneously. Our formulation selects tile sizes that benefit more from higher intra-SM and inter-thread data sharing versus intra-thread locality. Our approach constrains the time that data solely used by individual threads remain resident in the cache. At the same time, cache resources are better used when multiple threads share and access the same data, leading to higher reuse of inter-thread data, creating a similarity of *resource partitioning*. To achieve this type of reuse, we maximize the tile sizes of the loop dimensions that are parallelized (e.g., loops `ti/i` and `tj/j` in Fig. 4) and carry spatial data reuse (i.e., that induce CMA). In

[†]We refer to the tile sizes used in `matmul`, one per input loop dimension, as T_i, T_j and T_k.

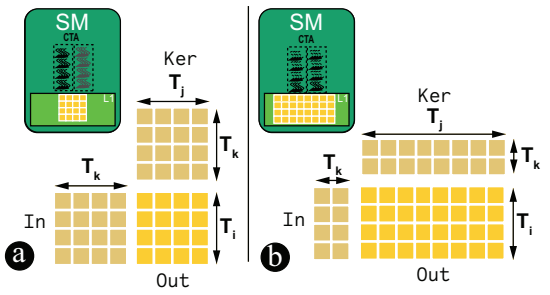


Fig. 5: Schematic showing the trade-off between intra-thread and inter-thread locality. Our objective function (See Sec. IV-K) maximizes the product of tile sizes (i.e., T_i and T_j)[†] associated to parallel loop dimensions. In turn, the tile size of loop dimension T_k (not parallel) and which carries reuse, is constrained. Smaller T_k values reduce the lifetime of tile `Out` in L1. Larger values of T_i and T_j increase intra-SM inter-thread data sharing as T_i determines the reuse on data tile `Ker`, while T_j controls the reuse on data tile `In`.

turn, this reduces the tile sizes associated with non-parallel loops, which carry data reuse as shown in Fig. 5(b). This optimization criteria effectively trade off intra-thread locality for inter-thread data reuse. However, we also have to maximize overall GPU resource utilization. Hence, our formulation also aims to maximize the number of threads per computational tile to be distributed over SMs. Our model is further parameterized by the amount of available GPU resources: *a warp fraction size*, the thread block size, the number of available registers per thread and per SM, the L1 and L2 cache sizes and the available shared memory. We adapt resource usage based on floating-point precision (i.e., FP32 (Single Precision) and FP64 (Double Precision)). Lastly, a unique feature of our tile size selection scheme is that it permits us to explore the trade-offs between mapping arrays to hardware-managed caches versus user-controlled shared memory in the GPU. These parameters are listed in Table I.

TABLE II: `matmul` array properties: CMA (Coalesced Memory Accesses), Reuse Type (T: Temporal, S: Spatial).

Array Reference	Memory Type	CMA Capable	Reuse Type (Loop Dim)
`Out[i][j]`	L1	Yes	T-reuse (k), S-reuse (j)
`In[i][k]`	Shared-Mem	No	T-reuse (j), S-reuse (k)
`Ker[k][j]`	L1	Yes	S-reuse (j)

Access Patterns: EATSS selects tile sizes by leveraging memory access patterns of array references. For the `matmul` example, a summary of the patterns and associated properties is shown in Table II. We first classify each array reference as either *cache mappable* or *shared memory mappable*. Arrays of the former type are those that can be accessed via CMA or are repeatedly and frequently updated (hence, they are guaranteed to remain in the cache). Arrays of the latter type may exhibit limited (spatial) reuse, but are not capable of CMA access (e.g., reference `In` of `matmul`). Thus, we assign a heavier weight to tile sizes that benefit cache-mappable references. This choice will, in turn, constrain tile sizes that possess temporal reuse or

no reuse at all. For example, in `matmul`, this constrains tile size T_k. We also harness the ability of modern GPUs to shift memory resources between the L1 cache and shared memory. This resource is modeled as a combined memory where the split between the two memory types is controlled via an input parameter in the range of 0.0 to 1.0.

Upon classifying array references by type, we favor tile sizes of dimensions amenable to CMA over those that are not CMA capable. We aim to estimate the energy consumed per array reference using the number of cache lines and the number of times that these are accessed. This assumption holds for threads in an SM when CMA-capable references exhibit O(n) reuse and when the data footprint of cache-mappable references fits in the L1 cache. If tile sizes are carefully chosen, the data for each cache-mappable reference will target a disjoint cache portion. In the `matmul` example, threads in each computational tile, mapped to an SM, will access cached data tiles of size $T_i \times T_j$ and $T_k \times T_j$ elements for arrays `Out` and `Ker`, respectively, as shown in Fig. 6. Further, the sum of both footprints must not exceed the L1 cache capacity per SM to avoid cache capacity misses and uphold our previous assumption for approximating energy. At the same time, the reuse in these references will be T_k for `Out` and T_i for `Ker`, higher being better. On the other hand, as array `In` is mapped to shared-memory, its data footprint, measured as $T_i \times T_k$ elements, will be limited to the shared memory capacity.

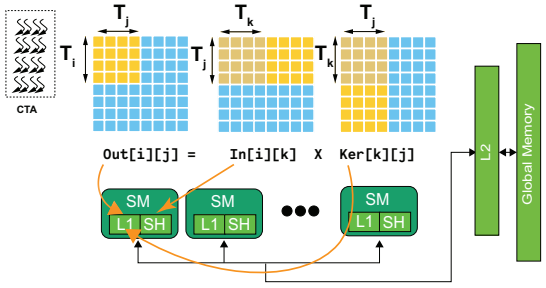


Fig. 6: Schematic of how is chosen the memory type for each array reference. Loop dimension j is selected as CMA (See Sec. IV-D). Thus, references `Out` and `Ker` are mapped to L1, while reference `In` is mapped to shared memory. (Sec. IV-E).

Resource Partitioning: Our model uses the hardware resource limits of Table I to enforce partitioning of GPU resources. In general, we model resource usage as some function of the tile sizes associated to iterators indexing array references. The number of threads in a thread block (modeled with B_{size}), upper-bounded by parameter T_P_B, will be the product of the tile sizes associated with the (parallel) loop dimensions used to access written arrays. For the `matmul` example, this would be $T_i \times T_j \leq T\_P\_B$. Similarly, the registers used per SM are modeled as the product of the size of the *thread block* and the number of references in the computation, while the memory footprint is estimated as the sum of the footprints of all arrays assigned to a memory type. For `matmul`, targeting the NVIDIA GA100, using 50% of the combined L1 + Shared Memory for each type of memory, FP_{factor} = 2 (for FP64), and a `WARP_ALIGNMENT_FACTOR` of 16 (0.5 of warp size T_P_W), these last hardware constraints would then be added to our formulation: $B_{size} \times 3 \times 2 \leq R\_P\_S$, $T_i \times T_j + T_k \times T_j \leq M_{L1}$ (L1 cache capacity constraints) and $T_i \times T_k \leq M_{SH}$ (shared memory capacity constraints).

Objective Function: Ultimately, we produce a single objective function that balances SM parallelism and inter-thread (but intra SM) locality with the selected tile sizes. For the `matmul` example, the objective function used is: $T_i \times T_j + [(0 \times T_i) + (2 \times 16 \times T_j) + (0 \times T_k)]$, which is iteratively optimized using Z3, and produces the solution: T_i=16, T_j=384, and T_k=16. The first term, $T_i \times T_j$, represents the parallelism component as the number of threads per computational tile, while terms 2-4 (in square brackets) are the tile size variables with a weight proportional to the CMA benefit achieved through the tile size of each loop dimension. We note that Z3 finds satisfiable solutions to our formulation. Therefore, our scheme finds a first solution and then attempts to maximize it, iteratively, until no better solution is found.

B. Bounds on Tile Size Variables

Let L be the maximum dimensionality of the loop nest. We define L tile size variables T_i, with $i \in \{1, 2, \ldots, L\}$. Each of the T_i variables is bounded by the integer interval $[1, T\_P\_B]$, defined in Table I. When the problem sizes are known, the upper limit of the previous range becomes `min(`T_P_B, N`)`. We also leverage Z3's expressiveness to force the selection of tile sizes to be multiples of a WARP size by introducing the constraint $T_i \% T\_P\_W = 0$. However, we note that this constraint can be adapted to smaller values. For example, instead of the typical size of 32 threads per warp, we use multiples of 16, 8, or 4. Leveraging this *knob*, which we call `WARP_ALIGNMENT_FACTOR`, becomes necessary since forcing tile sizes to be multiples of some large number could render an empty solution space. In practice, this occurs with higher-dimensional arrays, 3D and larger. Furthermore, the problem is exacerbated in smaller GPUs that are equipped with fewer resources.

C. Volume per Reference

Next, we model the required memory per array reference. Let k_i be the loop iterator associated with the loop level i, we represent the *memory used by a data tile* of reference f^A as $V^{f^A} = \prod_i tile\_extent(f^A, i)$, where `tile_extent()` is a helper function that returns T_i if iterator k_i is used in f^A, and 1 otherwise. For example, this model will produce the product $T_{i1} \times T_{i3}$ for a reference such as `C[i1][i3]` when surrounded by a 3D-nested loop. To accelerate and simplify the solution process, we lower-bound all V^{f^A} variables by each of the T_{k_i} tile size variables associated to iterators used to access f^A.

D. Selecting the Loop Dimension for Coalesced Memory Accesses (CMA)

Our approach prefers parallel loop dimensions that exhibit stride-1 access across as many array references as possible. This criterion is similar to the requirements for vectorization in

CPUs, with the difference that the CMA loop is instead used as the thread-id. When presented with two or more choices, we select the loop dimension used in most arrays and refer to this loop level as l_{s1}. For our `matmul` example, consisting of references `Out[i1][i2]`, `In[i1][i3]` and `Ker[i3][i2]`, the selected loop for CMA would be `loop-i2`, as it allows us to exploit CMA on two of the three references.

E. Splitting Memory References

Next, we split the set of array references into two sets: 1) those that are capable of effectively exploiting *coalesced memory accesses* and 2) those that do not. The first set will consist of the references that will exploit the L1 cache ($L1_{set}$), while the references in the second set will exploit *shared memory* (SH_{set}). Any array reference that produces a sequence of contiguous memory accesses in the loop l_{s1} will use the L1 cache, while those that do not will use the shared memory local to an SM. The following constraints model how much memory of each type is used in a kernel:

$$M_{L1} = \sum_{f^A \in L1_{set}} V^{f^A} \quad \wedge \quad M_{SH} = \sum_{f^A \in SH_{set}} V^{f^A} \qquad (1)$$

We note that the rationale for this decision is to better model CMA-incapable accesses, which in order to avoid repeated non-CMA accesses, are mapped to shared-memory. Intuitively, any non-CMA array reference would repeatedly access global memory, without the benefit of prorating this cost among the threads in the thread block. If non-CMA accesses are allowed into the L1/L2 caches, their accessed data – not shared by threads in the SM – would remain longer in the L1 and L2 caches, increasing the memory traffic and energy expenditure. Furthermore, mapping non-CMA data to shared-memory frees valuable resources to CMA-capable references.

Continuing with the `matmul` example, references `Out` and `Ker` would be chosen to exploit the L1 and L2 memories, while reference `In` will use shared memory.

F. Estimating the Thread-Block Size

To properly balance the selection of tile sizes, we estimate the number of threads per thread block by computing the product of the tile sizes used for up to the first three outer-parallel loop dimensions, $B_{size} = \prod_{i \ is \ par} T_i$. This choice is rooted in the fact that if a loop dimension is not parallel, it will not be selected as a mappable dimension for the GPU. Consequently, it will have no impact on the thread-block size, but rather only impact the locality and energy.

G. Controlling the Number of Registers per SM

A critical feature of our model is to limit the usage of the register per SM. While for most practical purposes the constraints on shared memory, L1 and L2 cache will prevent us from exceeding the limit on the number of registers used per SM, it can happen that owing to our L1+shared-memory split strategy, the tile size dimensions carrying temporal reuse will be allowed to grow unchecked. This phenomenon increases the memory footprint in the L2 cache, which together with the internal scheduling of thread blocks and warps (a black-box scheduler), will increase the time that data are live in the L2 cache. This can incur higher memory traffic between L2 and global memory. Moreover, as observed in prior studies [23], higher liveness of data at this level of the memory hierarchy in GPUs is one of the main sources of wasted energy. Our approach tries to avoid this scenario, favoring the reduction of tile sizes that only carry temporal reuse. The reader will notice that maximizing the thread block size will indirectly minimize the tile sizes associated with the loop dimensions that carry temporal reuse exclusively, as well as any other tile size of any parallel loop dimension not used for mapping to the GPU resources. In addition, to favor more equal sharing of registers (and of the shared/L1/L2 memories), we estimate the number of registers per SM as $REG_{SM} = B_{size} \times no.references$, where $no.references$ is the number of references accessing distinct cache lines. The REG_{SM} variable is upper-bounded by the GPU parameter R_P_S (Registers per SM). For example, the number of references for the `matmul` kernel would be 3, while for the `fdtd-2d` kernel it would be 4 (two references typically lie in the same cache line).

H. Modeling the L2 Cache

In some scenarios, we can prefer to use all combined L1 + shared memory resources only for shared memory. Consequently, the constraints about the L1 cache become irrelevant, and we skip their introduction. Afterward, while the L1 cache may no longer be a factor, the amount of L2 cache per SM effectively replaces the L1 constraints. The decision to adopt this modeling strategy for the L2 cache is based on the property that, when loading data from global memory into shared memory, most GPU architectures still have to go through the L2 cache. To the best of our knowledge, only the NVIDIA GA100 bypasses the L2 cache when loading data into shared memory.

I. Resource Adaptation to Single/Double Precision

In several architectures, including GPUs, it is common for the double precision (DP) peak performance to be half of the single precision (SP). We use this common GPU characteristic and introduce a *scaling precision factor* to adjust our model, thus making it aware of the implications of choosing SP or DP. To model this property, we refine the estimation of registers per SM, REG_{SM}, as $REG_{SM} = B_{size} \times no.references \times FP_{factor}$, where FP_{factor} is 1 for single precision and 2 for double precision. This refinement is inspired by the fact that in most CPU and GPU architectures, wider vector registers are produced by using multiple smaller ones. The impact of this decision halves the register budget per SM.

J. Establishing Resource Limits

Our formulation requires that a *split factor* between 0 and 1.0 be assigned as input to the model generator. This factor will be used to divide the combined L1 + shared memory per SM. A zero value for the *split factor* represents the full usage of L1 + shared memory for the L1 cache (if the architecture

allows it), while a value of 1.0 allocates all resources to shared memory. Values between 0 and 1.0 will be used to explore the design space of tile sizes. Typical values of interest are 0.25, 0.5, 0.75 for the GA100 and Xavier. Memory resource limits of the GPU are scaled down based on the byte width of the corresponding floating point datatype, for the model to use units compatible with the number of *loop iterations*.

$$M_{SH} \leq SPLIT\_FACTOR \times L1_{SH} \quad \wedge$$
$$M_{L1} \leq (1 - SPLIT\_FACTOR) \times L1_{SH}$$
$$\wedge \quad M_{L2} \leq L2\ (input) \quad \wedge \quad REG_{SM} \leq R\_P\_S$$

K. Tile Size Selection Objective Function

We use a single objective function (Eq. 2) to select tile size configurations that optimizes both performance and energy. It consists of two terms, the *parallelism term* (left) and a *spatial locality term* (right), producing a weighted sum of tile size variables.

$$OBJ = \prod_{i\ is\ par} T_i + \sum (H_i \times T_i) \tag{2}$$

Maximizing OBJ leads to **higher inter-thread data sharing** by preferring it over **intra-thread locality**. This is achieved by considering only parallel dimensions in the left term. Ultimately, OBJ is used as a proxy for active cache lines. Via dependence analysis, before applying the tiling transformation (but after computing it), loops are identified as parallel or serial (including permutable-only loops). The *parallelism* term is represented by the product of tile sizes that contribute to the size of the thread block. We also note that we only include up to the first three parallel loop dimensions, given the fact that threads in most GPUs only use a maximum of 3 dimensions for their index representations, and often only rely on 2 or 1. In most cases, this product will only involve two loop dimensions. The reader will also notice that when the loop chosen for CMA, l_{s1} is parallel, it will be chosen to define the thread-block size and also contribute to the objective function. The *weights* used in the *spatial locality term*, H_i, are statically computed fixed factors. They model the number of times each loop iterator appears in the fastest varying dimension (or stride-1 dimension) among all of the array references. Further, we consider here three sub-cases; First, when the loop iterator is associated to a loop dimension that can yield CMA, H_i is scaled by `WARP_ALIGNMENT_FACTOR`; Second, in loop nests with dimensionality 3D or higher, H_i is nullified if the loop is not parallel to favor CMA; Lastly, in 2D loop nests (when often only one loop dimension is parallel), the parallel loop is ignored (since it is already mapped), and we instead prefer to increase the tile size of the nonparallel loop.

L. Repeatedly Finding Solutions with Z3

We use the Z3 solver to find solutions to the *non-linear integer* formulation produced with EATSS. As Z3 does not find optimal solutions for this type of problem, we resort to introducing an additional constraint of the form $OBJ_{n+1} > OBJ_n$ and use it to find optimal solutions in an iterative fashion, and terminate when no more solutions are found. Hence, our approach finds a first solution and then proceeds to gradually improve upon it.

M. Advantages and Application of our Model Generator

Our model provides us with several features. First, it enables one to easily switch between single and double precision, while also allowing us to explore the effects of splitting the combined resources of the L1 and shared memory. Second, the formulation is *problem size agnostic*, but can also benefit from using fixed problem sizes when these are known. Third, the formulation presented in this section forms the basis for a *Model Generator* which can be used in a variety cases: i) it can be integrated into an *auto-tuning* framework to determine effective tile sizes for parametrically tiled code (when the transformation is applied before the selection of tile sizes) or to choose them before applying the transformation with *fixed tile sizes*; ii) it can be used in tandem with a GPU-polyhedral compiler such as PPCG [24] to determine *kernel-wide* tile sizes that control the granularity of work units in the GPU; iii) it can be integrated into toolchains that perform JIT compilation, which is commonplace in deep learning frameworks.

V. Experimental Evaluation

A. Experimental Setup

We evaluate the effectiveness of our tile size selection scheme in the Polybench / C 3.2 benchmark suite [25]. We complement our study with three non-Polybench kernels that are commonly used in linear algebra and machine learning, *mttkrp*, *heat-3d*, *convolution-2d*. Experiments were conducted on two NVIDIA GPUs that represent the high and low levels available in the market. Their characteristics in terms of performance and power consumption vary, and their details are shown in Table III. Given the different GPUs, we employ a different Polybench dataset for each system: EXTRALARGE for the GA100, and STANDARD for the Xavier GPUs.

TABLE III: GPU Testbed Specifications

	GA100	**AGX Xavier**
Architecture	Turing	Volta
Multiprocessor count	108	8
L1 / L2 cache	192 KB / 40 MB	128 KB / 512 KB
Shared-mem per block & SM	48 KB / 164 KB	48 KB / 96 KB
Registers per block	65536	65536
Global memory	40 GB	32 GB
CUDA version	11.4	10.2
Peak FP64 (GFLOPS)	9700	44†
Thermal design power	250W	30W

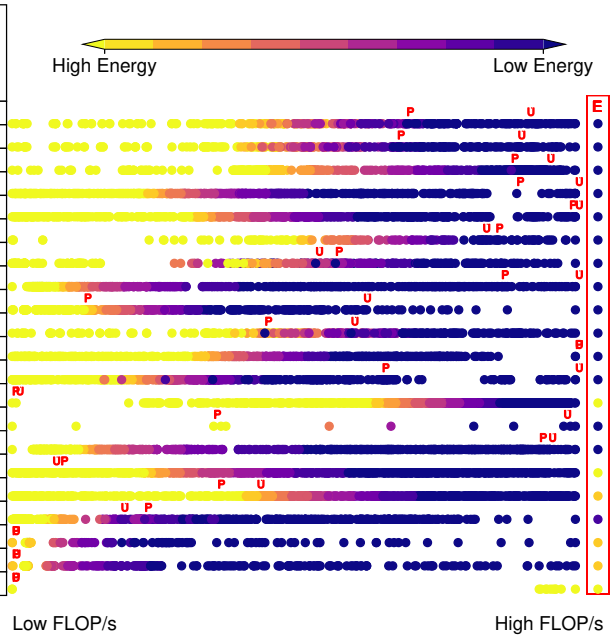
† cuBLAS FP64 performance

Exploration Space: We conduct seven sets of experiments. In the first set (Sec. V-B), we evaluate the benchmarks on all GPUs. For each benchmark, we explore a space of tile configurations (approximately 200-800 variants), depending on the maximum loop dimensionality of the kernel. We then generate with EATSS‡ three tile size configurations per benchmark (corresponding to three levels of shared memory), and compare *our* best result against the entirety of the tiled

‡Artifacts at: https://doi.org/10.5281/zenodo.10362265 [26], https://github.com/mkongiv/eatss.git [27].

Med PPCG Perf /Watt	Def PPCG Perf /Watt	Best PPCG Perf /Watt	Our Perf /Watt	Med PPCG (J)	Def PPCG (J)	Best PPCG (J)	Our (J)	Med PPCG GFLOP	Def PPCG GFLOP	Best PPCG GFLOP	†Our GFLOP	Bench
39.3	54.4	66.7	70.7	13.02	9.42	7.67	7.24	2603.9	2911.0	3993.0	3682.9	2mm
39.2	52.0	66.9	70.5	19.57	14.77	11.48	10.89	2597.4	2920.2	4097.3	3717.1	3mm
0.6	0.7	0.8	0.7	0.81	0.69	0.62	0.67	27.3	33.5	35.6	34.7	adi
1.6	2.5	3.8	3.1	0.32	0.2	0.14	0.17	74.1	158.5	176.6	191.2	atax
1.7	2.4	3.4	3.1	0.29	0.21	0.15	0.17	78.5	152.5	154.1	191.5	bicg
45.1	56.2	62.5	54.8	2.84	2.28	2.05	2.34	2152.9	2610.6	2920.5	2559.8	corre
34.5	53.7	56.3	48.7	3.7	2.38	2.27	2.63	2344.1	2483.3	4203.7	2341.2	covar
1.9	6.3	7.5	7.2	0.17	0.05	0.04	0.04	86.5	303.8	346.8	349.2	fdtd-2d
0.5	0.3	1.3	0.9	144.71	194.92	52.23	79.77	30.3	23.6	81.8	56.7	fdtd-apm
40.2	56.9	68.6	78.0	6.37	4.5	3.73	3.28	3676.0	2897.3	5865.3	3721.2	gemm
2.6	4.1	5.4	5.1	0.49	0.31	0.24	0.25	121.2	289.3	289.3	353.8	gemve
1.1	2.4	3.4	4.2	0.46	0.22	0.15	0.12	73.7	146.5	222.4	258.7	gesum
0.06	0.01	0.07	0.01	0.02	0.12	0.02	0.11	4.1	0.4	5.0	0.4	grams
0.7	0.6	0.9	0.9	0.001	0.001	0.001	0.001	31.9	27.6	40.9	40.5	jacob-1d
3.2	8.2	9.1	8.3	0.05	0.02	0.02	0.02	147.1	393.6	419.4	400.2	jacob-2d
0.5	0.2	0.9	0.1	0.002	0.007	0.001	0.008	32.2	7.4	61.9	6.5	lu
0.9	0.6	1.5	0.7	0.15	0.23	0.08	0.19	59.6	39.3	105.6	46.5	mvt
7.9	7.8	27.1	6.7	40.3	41.0	11.79	48.06	870.8	730.4	2802.1	616.5	symm
7.9	4.4	34.9	4.0	48.54	87.72	11.01	96.18	699.3	237.8	3575.7	239.1	syr2k
7.5	3.1	34.0	2.9	17.02	41.86	3.77	43.74	645.7	157.9	2820.7	161.8	syrk
0.01	0.01	0.01	0.01	0.001	0.001	0.001	0.001	0.5	0.3	0.5	0.3	triso
2.35	2.42	4.9	2.82	gmean								

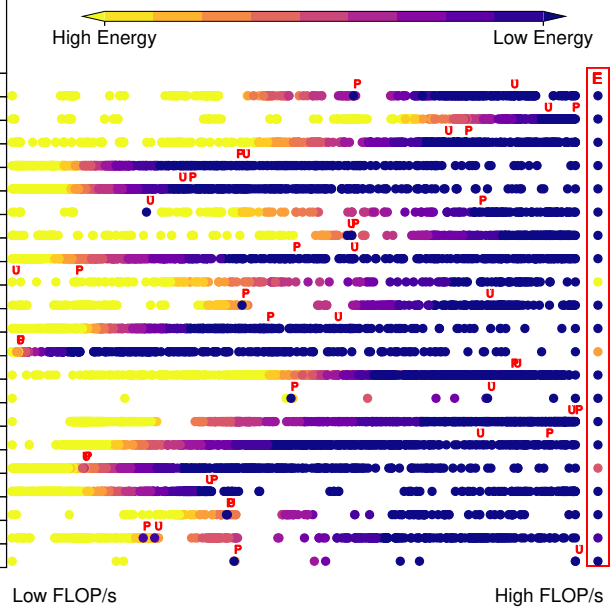
†Note: our performance per watt corresponds to our most performant tile size



(a) Performance and energy distribution on NVIDIA GA100.

Med PPCG Perf /Watt	Def PPCG Perf /Watt	Best PPCG Perf /Watt	Our Perf /Watt	Med PPCG (mJ)	Def PPCG (mJ)	Best PPCG (mJ)	Our (mJ)	Med PPCG GFLOP	Def PPCG GFLOP	Best PPCG GFLOP	†Our GFLOP	Bench
21.2	32.3	32.3	39.3	405.65	265.63	265.63	218.58	34.1	31.4	40.9	38.2	2mm
25.5	42.9	42.9	41.3	505.36	300.65	300.65	311.72	39.7	42.8	42.9	41.5	3mm
3.0	5.2	5.2	5.0	10.6	6.1	6.1	6.35	3.9	4.7	5.7	4.6	adi
2.8	6.6	12.6	6.9	46.32	19.43	10.19	18.5	3.7	7.4	17.8	7.6	atax
3.5	6.2	14.8	6.0	36.59	20.52	8.64	21.3	4.1	6.0	18.0	5.7	bicg
31.6	49.8	49.8	37.1	63.95	40.57	40.57	54.37	35.9	37.0	39.8	27.4	corre
30.0	36.7	36.7	36.3	66.7	54.44	54.44	55.08	33.8	26.8	40.8	26.5	covar
2.9	8.2	9.9	9.5	6.97	2.45	2.03	2.11	3.9	7.6	13.4	8.8	fdtd-2d
3.1	2.0	4.3	1.8	365.23	577.65	265.53	647.2	3.5	2.6	4.7	2.3	fdtd-apm
26.4	34.5	34.5	49.1	162.93	124.42	124.42	87.45	30.6	24.9	39.3	35.5	gemm
4.4	11.6	18.6	13.8	72.18	27.64	17.21	23.2	5.6	10.4	21.8	12.9	gemve
0.3	0.2	7.6	0.2	447.59	586.84	16.93	605.81	0.4	0.2	9.8	0.2	gesum
2.7	3.0	3.6	3.0	0.38	0.35	0.29	0.35	2.1	2.6	2.9	2.6	grams
1.8	2.2	2.2	2.7	0.03	0.03	0.03	0.02	2.2	2.0	2.7	2.5	jacob-1d
6.4	13.5	13.5	13.4	1.57	0.74	0.74	0.75	8.2	12.1	12.1	11.9	jacob-2d
3.3	10.2	10.2	9.0	0.32	0.1	0.1	0.12	4.2	9.1	9.6	7.9	lu
3.3	2.7	14.0	2.6	38.26	47.38	9.17	48.7	4.2	2.7	17.7	2.7	mvt
9.3	14.5	24.8	14.2	579.18	371.28	216.14	377.34	11.9	13.2	31.1	12.9	symm
13.1	16.6	21.5	16.6	493.32	387.31	299.42	389.38	17.7	15.5	29.3	15.6	syr2k
9.0	11.1	14.8	11.4	238.32	193.88	145.67	188.92	12.3	10.4	20.1	10.6	syrk
0.2	0.3	0.3	0.4	0.03	0.03	0.03	0.02	0.3	0.3	0.3	0.3	triso
4.97	7.34	11.49	7.56	gmean								

†Note: our performance per watt corresponds to our most performant tile size



(b) Performance and energy distribution on Jetson AGX Xavier®.

Fig. 7: The X-axis shows the normalized FLOPs and each dot represents a tiled variant. P is default PPCG, U is EATSS and E shows our energy separately. (**Table key**: Med PPCG: program variant with median performance in the space, Def PPCG: program variant with default tile configuration (32^n), Best PPCG: program variant with best performance in the space.)

space across the two metrics of interest, performance, and energy. Tile sizes are passed to PPCG to produce efficient CUDA-tiled code for each GPU. We set the amount of available shared memory per GPU to match our best configuration and keep all PPCG default options. As a second experiment (Sec. V-C), to understand the underlying source of our energy efficiency, we analyze the correlation between the number of L2 cache lines (sectors) and the average power across four benchmarks of representative computational classes. In Sec. V-D, we evaluate three non-polybench kernels often used in computer vision and machine learning applications on the GA100. Subsequently, in Sec. V-E, we compare our

performance against native CUDA libraries, in Sec. V-F we study the impact of varying the problem size and in Sec. V-G we briefly discuss the low overhead associated with Z3. Lastly, in Sec. V-H we compare EATSS with *ytopt*, a state-of-the-art auto-tuner.

Methodology for Data Collection: Power measurements are obtained with *nvidia-smi* [28] for GA100 and *tegrastats* [29] for Xavier. Each tiled kernel variant is run 100 times with samples collected at 10ms intervals to allow for a sufficient number of samples to be collected during execution (similar to recent work [12]). Execution times and cache metrics at the kernel level are obtained with NVIDIA NSight [30]. Having obtained the execution time (in seconds) and the average power (in Watts) during execution, we estimate the energy consumption in joules ($energy = power_{average} \times time_{exec}$).

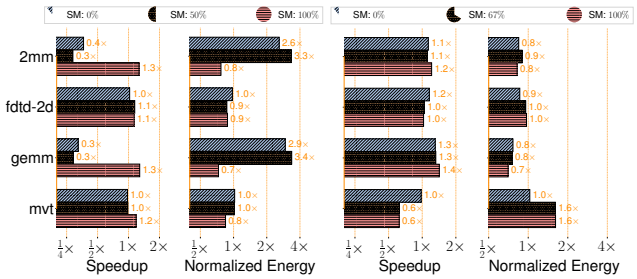
B. Results on Polybench

Fig. 7a-7b show the performance and energy expenditure range achieved over the evaluated space. We include the "default tile configuration" performance, i.e., 32^d (d: maximal loop depth of the kernel), and mark it with 'P' in the spectrum. The tables on the left also show the median and best empirically found data points in the space for performance-per-Watt (PPW), performance, and energy. The 32^d tile sizes often yield good out-of-the-box performance, **but still under-perform owing to the criterion of only using a large fraction of 32KB caches without over-spilling, largely ignoring energy efficiency implications.** EATSS achieved performance and energy within the spectrum is denoted with 'U' (us).

To capture the effects of performance and energy improvement, we use the *performance-per-Watt* metric [31]–[33], as computed in Eq. V-B. The intuition of this metric is that given the same number of flops performed by two tiled variants of the same benchmark in their respective executions, the *performance-per-Watt* represents the number of FLOPs that can be computed with a unit of energy.

$$\textit{performance / watt (PPW)} = \frac{\text{(Floating-point ops per second)}}{\text{average power (W)}}$$

Overview of expected results: In general, we should hope for stronger improvements on dense linear algebra kernels that exhibit two or more levels of parallelism, an order $O(n)$ of data reuse, and at least one vectorizable loop dimension. Kernels with these characteristics are mainly of the BLAS3 type (e.g., *gemm*), but also *covariance* and *correlation*. We find a second class in 2D-kernels that exhibit constant reuse $O(1)$, e.g., *mvt*. In these kernels, our forecast changes to modest improvements given that there are fewer trade-offs between locality and parallelism to exploit. Lastly, kernels of a third type, mostly consisting of iterative stencils (*jacobi-2d, fdtd-2d*), are interesting in that there is often a single, extremely obvious loop dimension used to exploit stride-1 and CMA. However, since PPCG does not exploit interstep data reuse (i.e., *time-tiling*), and only the space dimensions are tiled, we expect only a moderate level of improvement.



(a) GA100 (b) Xavier

Fig. 8: Performance and energy achieved with EATSS under different splits of shared-memory and L1-cache. **Speedup: higher (> 1) the better** and **energy: lower (< 1) the better** (SM: Shared Memory); metrics normalized to Default PPCG.

Analysis of evaluation: Our results on the GA100 (Fig. 7a), exhibit the following trends. For *2mm*, *3mm* and *gemm*, we outperform default PPCG in terms of performance and are close to the best performance observed in our search space (for 2mm & 3mm). Although we may not achieve the best performance in terms of FLOPs, the *performance-per-Watt* for our scheme is higher than that of the best PPCG result (which uses *our chosen* amount of shared memory). This is because EATSS trades intra-thread data reuse for inter-thread data sharing while cooperating with the GPU's DVFS.

In the stencil category – *fdtd-2d, fdtd-apml*, *jacobi-1d* and *jacobi-2d* – we outperform *default PPCG* moderately in terms of performance and *performance-per-Watt*. Our scheme can exploit CMA along a single-loop dimension. In low-dimensional kernels (*atax*, *bicg*, *mvt* and *gemver*), EATSS outperforms default PPCG in *performance-per-Watt*. Selecting an energy-efficient tile size for these kernels is difficult since they only have one parallelizable loop mapped to CTAs, thus limited data reuse.

On the Xavier GPU, Fig. 7b, we achieve near-peak performance the *gemm, 2mm, 3mm and covariance* kernels. We greatly outperform the default PPCG performance for the *2mm* and *gemm* benchmarks, but observe slightly poorer performance for *3mm*. For *gemm*, we obtain a performance improvement of 45% and an improvement of about 42% in *performance-per-Watt*. Turning our attention to stencil computations, we can see that *fdtd-2d* & *jacobi-1d* outperform default PPCG. However, *fdtd-apml* and *jacobi-2d* experience a slight slowdown. On the *gemver* kernel, we outperform default PPCG in terms of both performance and *performance-per-Watt*. However, *mvt* and *atax* experience no performance gain, while *bicg* experiences a slight slowdown.

Impact of Shared-Memory quotas: We now demonstrate that, to improve performance and energy efficiency, the optimization space of tile size selection must consider the possible splits between shared-memory and L1 cache. To show this, we compare EATSS against PPCG's default performance and energy under the same amount of shared-memory as *our best* (See Fig. 8). For each platform, we consider three levels of shared-memory usage from the set – 0%, 50% or 67%, and

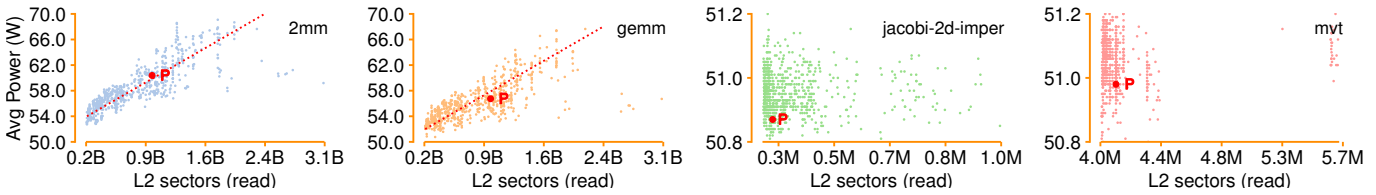


Fig. 9: Correlation between average power of tiled program execution with the number of L2 cache lines (sectors) read; 2mm and gemm (BLAS3) show a strong correlation as opposed to jacobi-2d (stencil) and mvt (linear-algebra). P is default PPCG.

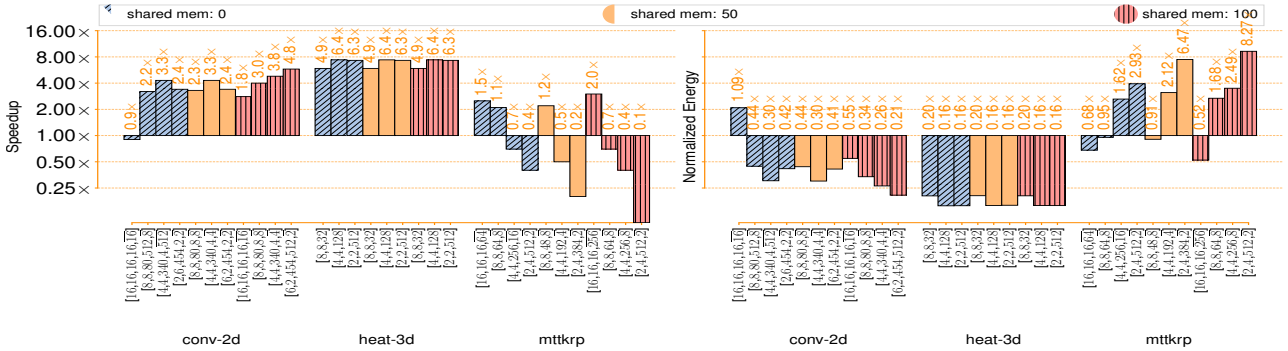


Fig. 10: Performance and energy improvements of non-polybench Benchmarks on GA100. **Speedup: higher (>1) the better** and **energy: lower (<1) the better**. The baseline is the default PPCG tiling. Note: We observed that the PPCG code generator ignores the tiling for the innermost loop when depth >3. Such dimensions are indicated with an overline.

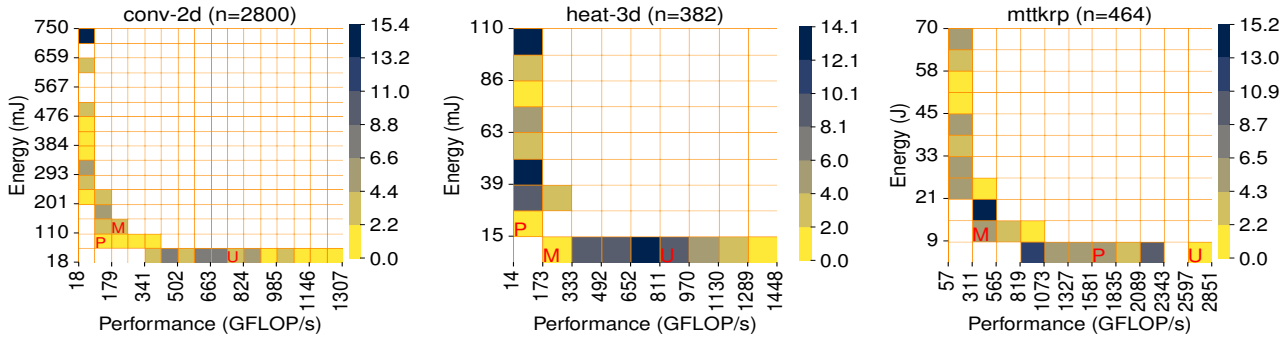


Fig. 11: Performance and energy distribution of non-polybench benchmarks. P: default PPCG tile sizes, M: median PPCG performance, and U: our best performance in Fig. 10. n: number of tile sizes evaluated in the baseline space.

100%. We observe that using 100% of shared-memory does not always yield the best performance or energy efficiency. Using less shared-memory results in increased L1 and L2 cache sharing, and in-cache data liveness. For the BLAS3-like kernels, we can see that exploiting more shared-memory will typically amount to better performance and lower energy usage. On the contrary, for lower-dimensional kernels, such as *mvt*, using less shared-memory (0% or 50%) often produces the best results on the Xavier.

C. Relation between L2 Cache Usage and Average Power

Fig. 9 shows cache statistics of 700+ tiled variants for *2mm*, *gemm*, *jacobi-2d* and *mvt*. We note the relation between the number of L2 cache lines read by the kernel and the average power during execution. We use the number of L2 cache lines read (extracted using ***lts__t_sectors_srcunit_tex_op_read.sum*** keyword in Nsight Compute profiler) as a proxy to data-liveness in the L2 cache, as this metric can only be computed through simulation, as shown in previous work [23]. We observe a high correlation between the number of L2 cache lines, also referred to as sectors, read by *2mm*, *gemm* program variants, and the average power reported. In Fig. 9 Pearson's correlation coefficient is 0.85 and 0.75 for *2mm* and *gemm*, respectively. The same trend does not hold in kernels with O(1) reuse, e.g., *jacobi-2d* and *mvt*. In summary, these observations support our choice of objective function formulation, which captures the effects of the L2 cache utilization to generate energy-aware tile sizes. Our results also explain the reasons behind the improvements achieved by EATSS.

D. Case Study - Leveraging different warp fractions

Next, we present results on three non-polybench kernels, *a 2D-convolution*, a *3D-heat stencil* and *mttkrp* (matricized tensor times a Khatri-Rao product). We note that these kernels are notoriously hard to optimize, owing to larger transformation spaces resulting from higher loop dimensionality (4D), larger

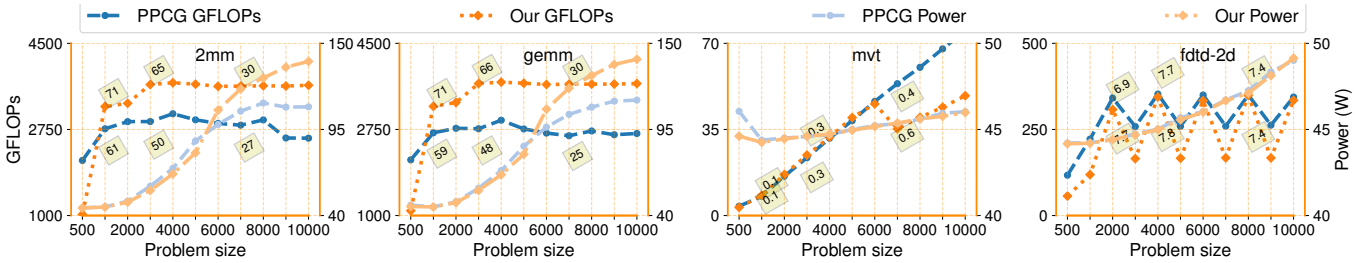


Fig. 12: Relationship of performance and average power with input size. The highlighted labels show the PPW.

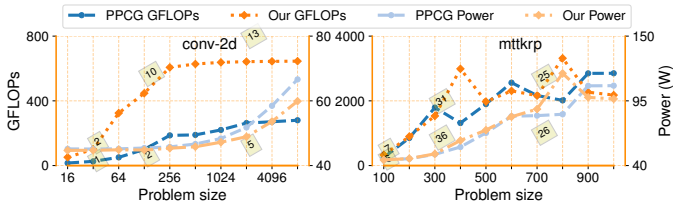


Fig. 13: Performance and average power relation as a function of input size, GA100. Highlighted labels show the PPW.

memory space requirements, complex data access patterns and several choices for parallelization that affect intra-SM locality. This is evidenced by the performance distribution shown in Fig. 11. We see that *PPCG's default tile size* (P marker) and *median* (M) perform rather poorly. Simply selecting a tile size by respecting resource constraints largely ignores its effect on the granularity of parallelism, as larger tile sizes yield fewer units of work to distribute across the 2D-parallel architecture of GPUs. These experiments are conducted on the GA100, and use 0% or 50% of shared-memory for baseline variants. The two shared-memory quotas used are chosen based on the results of Sec. V-B. The higher data dimensionality of these kernels (e.g., 3D arrays) requires allowing for tile sizes smaller than a *full warp* (32). Otherwise, several configurations could exceed shared memory and L1 capacity. We set the parameter *warp fraction* to 0.125, 0.25, 0.5, and 1.0 to produce a larger exploration space with EATSS. This allows us to explore tile sizes as small as 4. The defined space will consist of 11 tile configurations for *conv-2d* and *mttkrp*, and 9 tile sizes for *heat-3d*. Missing configurations do not satisfy resource constraints, as all tile sizes would need to be multiples of 16. These cases correspond to the set *warp_frac=1.0* with shared-memory=50% for *conv-2d* and *mttkrp*, and all scenarios of *heat-3d* with *warp_frac*=1.0.

Fig. 10 shows an overall speedup of 4.8× for *conv-2d*, 6.3× for *heat-3d* and 2.0× for *mttkrp*, relative to PPCG's default configuration using the same quantity of shared-memory as our best variant. Our selected tile sizes lead to the same improvements on energy, i.e., 4.8× for *conv-2d*, 6.3× for *heat-3d* and 1.9× for *mttkrp*. Fig. 11 shows the explored tiled-space as a 2D histogram using the Freedman Diaconis Estimator to take data variability and data sizes into account when selecting bin sizes. Bins located closer to the bottom-right corner represent higher performance and better energy efficiency. Our study highlights that EATSS not only beats PPCG's default and median performance, but that we are also reasonably close to the best empirically found variants at a very low exploration cost.

E. Comparison with cuBLAS / cuDNN

In Table IV, we compare the performance of EATSS with cuDNN (v8.1) / cuBLAS (v11.3). Unlike cuBLAS and cuDNN, PPCG generated code does not leverage tensor cores, and a direct comparison between PPCG and cuXXX is not possible. The peak FP64 machine performance of the GA100, with tensor cores, is 19.5 TFLOP/s, and without tensor cores is 9.7 TFLOP/s. Our results show that we can achieve 75% of the highly tuned cuBLAS/cuDNN PPW on the GA100, and more than 2.1x of the PPW on the Xavier GPU.

TABLE IV: Comparison against cuBLAS / cuDNN

Description	**cuBLAS (gemm)**		**cuDNN (conv-2d)**
Platform	**GA100**	**Xavier**	**GA100**
cuXXX Perf/Watt	105.94	23.32	24.8
PPCG Median Perf/Watt	40.2	26.4	2.9
Our Perf/Watt	78	49.1	18.9
cuXXX Energy	2.42 J	180 mJ	6.28 mJ
PPCG Median Energy	6.37 J	162.93 mJ	134 mJ
Our Energy	3.28 J	87.45 mJ	21 mJ
cuXXX GFLOP/s	18292	42.31	1405.55
PPCG Median GFLOP/s	3676	30.6	180
Our GFLOP/s	3721	35.5	813

F. A sensitivity study of the input size on power consumption

We present the impact of problem size on EATSS in Fig. 12. Our performance uses the best tile size determined by EATSS and we use the default tile size for PPCG. We do not use auto-tuning for tile selection of PPCG as the search space explodes when varying both input size and tile dimensions. As shown in Fig. 1, for smaller problem sizes in *2mm* and *gemm*, static + constant power is dominant. Intuitively, as the problem size increases, the GPU's SMs reach full utilization. Therefore, the power consumption will reach saturation for very large problem sizes. This observation is consistent with Fig. 1. *mvt* and *fdtd-2d* do not computationally saturate the GPU and predominantly use static power in terms of total power consumption. In the case of *fdtd-2d*, we observe a zigzag behavior due a high number of cache conflict misses. We investigated and found that padding improved performance and removed the zigzag effect. We end the sensitivity study by analyzing the problem size impact on non-polybench kernels when using EATSS in Fig. 13. For conv-2d, the *performance-per-Watt* is higher than observed using the baseline PPCG.

G. Compile Time Overhead due to Z3

Our iterative scheme works quickly. EATSS averages 1.3 seconds for the end-to-end iterative process across all kernel dimensionalities and architectures with the Z3 Python-solver. If grouping all configurations by classes of maximum kernel loop depth (2D, 3D, 4D, and 5D computations), we obtain average end-to-end times of 1.1 s, 1.4 s, 1.4 s, and 2.2 s, respectively. A single call to the Z3 solver, using our formulation in 391 different configurations, requires 0.29 seconds and between 4 to 7 solver calls, on average.

H. Comparison with Autotuners

We next provide a comparison with ytop, a state-of-the-art autotuner based on Bayesian optimization. We built ytop with Clang + OpenMP GPU device offloading support. Since *ytopt* relies on OpenMP, performance decreases compared to PPCG, which generates native CUDA code. However, for kernels with 3 nested loops, such as *2mm*, *gemm*, *heat-3d* and *mttkrp*, the tuning time was observed to be 17 minutes, whereas EATSS+PPCG generates CUDA code from selected tile sizes in seconds, as shown in Sec. V-G.

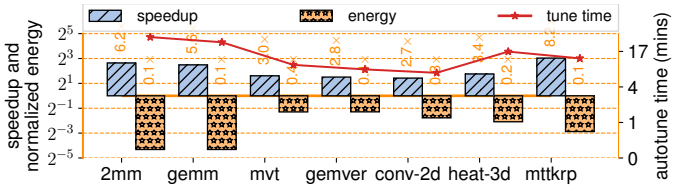


Fig. 14: Speedup (> 1 is better) and normalized energy (< 1 is better) of EATSS against *ytopt* [34] baseline on A100.

VI. Related Work

Tile Size Selection (TSS): Several polyhedral frameworks, such as PLuTo [1], Polly [35], PPCG [24], polly-ACC [36] and PolyMage [37], leverage loop tiling. Earlier work on TSS was strongly tied to cache characteristics [38]–[40]. Determining the best tile sizes and unroll factors simultaneously was explored by Kisuki et al. [41]. Rather than a static cost model, the authors use an iterative code generation approach to determine the best combination of tile size and loop unroll factor. Coleman et al. [38] chose tile sizes based on cache line size and fixed problem sizes for direct-mapped memories. However, direct-mapped caches are not widely used in modern microarchitectures. Recent work on TSS focuses on leveraging tiling and reuse by analyzing either the usage of the cache hierarchy or by reducing cache conflict misses [42], [43]. In the multicore / GPU era, techniques proposed for TSS on CPUs range from *dynamic programming* [44], through artificial neural networks [45], to directly incorporating them into ILP scheduling [46]. On GPUs, techniques such as optimizing to reduce uncoalesced global memory accesses [47] and performing a highly parallel reuse distance analysis [48], aim to improve performance. In contrast, earlier work on TSS on GPUs focused on do-across parallelism requiring the wavefront method. More recently, Abdelaal and Kong [5], building on the *legal space* formulation [49], proposed a joint-scheduling and TSS scheme for GPUs. In contrast, we introduce a non-linear integer programming problem that incorporates both GPU performance and GPU energy as objectives. Wang et al. [50] show the energy impact of auto-tuned polyhedral optimizations (i.e., tiling) on CPUs, while Pradelle et al. [51] proposed a compile-time energy model and energy optimizations that use polyhedral IR. Such techniques are limited to CPU platforms and do not consider both energy and performance. Orthogonally, empirical auto-tuning has been widely used for TSS [34], [52]–[54].

Energy Efficient Computing with Hardware / Software: Performance and energy modeling of GPUs has received much attention [12], [55], [56]. Adaptive cache management [31] and novel L2 cache designs [23] have recently aimed to improve the GPU energy efficiency. Although these studies consider the impact of architectural features on energy efficiency, they do not consider the impact that compilation techniques have on energy consumption. Our technique EATSS, incorporates variables such as warp size and register tiling in the code generation process to generate energy-efficient programs. Furthermore, the management of power caps and DVFS have also been proposed as effective energy optimizations on GPUs [15], [17] as well as CPUs [57]. Unlike explicit DVFS energy modeling, which might significantly differ depending on the hardware generation, we propose a generalizable model applicable to both server grade (GA100) and embedded (Xavier) GPUs.

Compiler Optimizations of GPU Energy Efficiency: Optimizations of this class can be broadly grouped into partitioning schemes [58], [59], runtime frequency adjustment [57], static compiler analyses and flag selection [60]–[62] and hardware-software codesign [63]–[65]. In contrast to existing work, to the best of our knowledge, we are the first to explore energy and performance optimization as a compiler objective in selecting tile sizes targeting GPUs.

VII. Conclusion

In this paper, we introduce a new tile size selection approach to exploit the trade-off between data reuse and data sharing, which leads to short intra-thread data liveness but higher inter-thread sharing. This translates to more energy savings, and by incorporating a combined performance-energy objective, we are able to choose tile sizes that deliver more performance per unit of energy consumed across 2 different GPU architectures. Our research will motivate the community to target performance per unit of energy metrics rather than solely throughput on future GPUs.

VIII. Acknowledgements

This research was supported in part by the Northeastern University Institute for Experiential AI and by the U.S. National Science Foundation through award 2234376.

REFERENCES

[1] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 101–113. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595

[2] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 319–329. [Online]. Available: https://doi.org/10.1145/73560.73588

[3] A. W. Lim, G. I. Cheong, and M. S. Lam, "An affine partitioning algorithm to maximize parallelism and minimize communication," in *Proceedings of the 13th International Conference on Supercomputing*, ser. ICS '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 228–237. [Online]. Available: https://doi.org/10.1145/305138.305197

[4] M. M. Strout, L. Carter, and J. Ferrante, "Compile-time composition of run-time data and iteration reorderings," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 91–102. [Online]. Available: http://doi.acm.org/10.1145/781131.781142

[5] K. Abdelaal and M. Kong, "Tile size selection of affine programs for gpgpus using polyhedral cross-compilation," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 13–26. [Online]. Available: https://doi.org/10.1145/3447818.3460369

[6] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 311–320. [Online]. Available: https://doi.org/10.1145/2304576.2304619

[7] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing the convolution operation to accelerate deep neural networks on fpga," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1354–1367, 2018.

[8] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 29–38. [Online]. Available: https://doi.org/10.1145/2435264.2435273

[9] B. Degnan, B. Marr, and J. Hasler, "Assessing trends in performance per watt for signal processing applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 1, pp. 58–66, 2015.

[10] R. Rodrigues, A. Annamalai, I. Koren, S. Kundu, and O. Khan, "Performance per watt benefits of dynamic core morphing in asymmetric multicores," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. Galveston, TX, USA: IEEE, 2011, pp. 121–130.

[11] Apple, "Apple unveils m1 ultra, the world's most powerful chip for a personal computer," Sep 2022. [Online]. Available: https://www.apple.com/newsroom/2022/03/apple-unveils-m1-ultra-the-worlds-most-powerful-chip-for-a-personal-computer/

[12] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T. G. Rogers, T. M. Aamodt, and N. Hardavellas, "Accelwattch: A power modeling framework for modern gpus," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 738–753. [Online]. Available: https://doi.org/10.1145/3466752.3480063

[13] I. User, "Facebook's new front-end server design delivers on performance without sucking up power," Dec 2018. [Online]. Available: https://engineering.fb.com/2016/03/09/data-center-engineering/facebook-s-new-front-end-server-design-delivers-on-performance-without-sucking-up-power/

[14] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in nlp," 2019. [Online]. Available: https://arxiv.org/abs/1906.02243

[15] K. Tang, D. Tiwari, S. Gupta, P. Huang, Q. Lu, C. Engelmann, and X. He, "Power-capping aware checkpointing: On the interplay among power-capping, temperature, reliability, performance, and energy," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Toulouse, France: IEEE, 2016, pp. 311–322.

[16] S. Reda, R. Cochran, and A. K. Coskun, "Adaptive power capping for servers with multithreaded workloads," *IEEE Micro*, vol. 32, no. 5, pp. 64–75, 2012.

[17] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra, "Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. San Francisco, CA, USA: IEEE, 2015, pp. 1–11.

[18] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas, "Gpgpu power modeling for multi-domain voltage-frequency scaling," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Vienna, Austria: IEEE, 2018, pp. 789–800.

[19] C.-H. Hsu and U. Kremer, "The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction," *SIGPLAN Not.*, vol. 38, no. 5, p. 38–48, may 2003. [Online]. Available: https://doi.org/10.1145/780822.781137

[20] W. Bao, C. Hong, S. Chunduri, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, and P. Sadayappan, "Static and dynamic frequency scaling on multicore cpus," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, dec 2016. [Online]. Available: https://doi.org/10.1145/3011017

[21] L. d. Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[22] Moura et al, "Z3 solver." [Online]. Available: https://github.com/Z3Prover/z3.git

[23] J. Tan, K. Yan, S. L. Song, and X. Fu, "Energy-efficient gpu l2 cache design using instruction-level data locality similarity," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 25, no. 6, aug 2020. [Online]. Available: https://doi.org/10.1145/3408060

[24] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400713

[25] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://polybench.sf.net*, 2012.

[26] M. Jayaweera, M. Kong, Y. Wang, and D. Kaeli, "EATSS: Energy-aware affine tile size selection." [Online]. Available: https://doi.org/10.5281/zenodo.10362265

[27] ——, "EATSS github repository: Energy-aware affine tile size selection," 2024. [Online]. Available: https://github.com/mkongiv/eatss.git

[28] "System management interface smi," accessed: 2023-11-18. [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface

[29] "Tegrastats utility," accessed: 2023-11-18. [Online]. Available: https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html\#page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegra\stats.html

[30] "Nsight compute cli::nsight compute documentation," accessed: 2023-11-18. [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

[31] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient gpu computing," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. USA: IEEE, 2014, pp. 343–355.

[32] A. Ilic, F. Pratas, and L. Sousa, "Beyond the roofline: Cache-aware power and energy-efficiency modeling for multi-cores," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 52–58, 2017.

[33] A. Lopes, F. Pratas, L. Sousa, and A. Ilic, "Exploring gpu performance, power and energy-efficiency bounds with cache-aware roofline modeling," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. USA: IEEE, 2017, pp. 259–268.

[34] X. Wu, M. Kruse, P. Balaprakash, H. Finkel, P. Hovland, V. Taylor, and M. Hall, "Autotuning polybench benchmarks with llvm clang/polly loop optimization pragmas using bayesian optimization," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 20, p. e6683, 2022.

[35] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[36] T. Grosser and T. Hoefler, "Polly-acc transparent compilation to heterogeneous hardware," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016, pp. 1:1–1:13. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926286

[37] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 429–443. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694364

[38] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, ser. PLDI '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 279–290. [Online]. Available: https://doi.org/10.1145/207110.207162

[39] J. Xue and C.-H. Huang, "Reuse-driven tiling for improving data locality," *International Journal of Parallel Programming*, vol. 26, no. 6, pp. 671–696, 1998. [Online]. Available: https://doi.org/10.1023/A:1018734612524

[40] J. J. K. Park, Y. Park, and S. Mahlke, "A bypass first policy for energy-efficient last level caches," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2016, pp. 63–70.

[41] T. Kisuki, P. M. Knijnenburg, and M. F. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," in *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*. IEEE, 2000, pp. 237–246.

[42] S. Mehta, P.-H. Lin, and P.-C. Yew, "Revisiting loop fusion in the polyhedral framework," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 233–246. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555250

[43] V. Ferrari, R. Sousa, M. Pereira, J. a. P. L. de Carvalho, J. N. Amaral, and G. Araujo, "Improving convolution via cache hierarchy tiling and reduced packing," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '22. New York, NY, USA: Association for Computing Machinery, 2023, p. 538–539. [Online]. Available: https://doi.org/10.1145/3559009.3569678

[44] A. Jangda and U. Bondhugula, "An effective fusion and tile size model for optimizing image processing pipelines," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: ACM, 2018, pp. 261–275. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178507

[45] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger, and K. O'Brien, "Automatic creation of tile size selection models," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 190–199. [Online]. Available: https://doi.org/10.1145/1772954.1772982

[46] U. Bondhugula, A. Acharya, and A. Cohen, "The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 3, pp. 12:1–12:32, Apr. 2016. [Online]. Available: http://doi.acm.org/10.1145/2896389

[47] R. Alur, J. Devietti, O. S. N. Leija, and N. Singhania, "Static detection of uncoalesced accesses in gpu programs," *Formal Methods in System Design*, vol. 60, no. 1, pp. 1–32, 2022. [Online]. Available: https://doi.org/10.1007/s10703-021-00362-8

[48] H. Cui, Q. Yi, J. Xue, L. Wang, Y. Yang, and X. Feng, "A highly parallel reuse distance analysis algorithm on gpus," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 1080–1092.

[49] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache, "Loop transformations: Convexity, pruning and optimization," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 549–562. [Online]. Available: https://doi.org/10.1145/1926385.1926449

[50] W. Wang, J. Cavazos, and A. Porterfield, "Energy auto-tuning using the polyhedral approach," in *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, S. Rajopadhye and S. Verdoolaege, Eds. Vienna, Austria: IMPACT, Jan 2014.

[51] B. Pradelle, M. Baskaran, T. Henretty, B. Meister, A. Konstantinidis, and R. Lethin, "Polyhedral compilation for energy efficiency," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. USA: IEEE, 2016, pp. 1–7.

[52] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 303–316. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628092

[53] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[54] X. Wu, P. Balaprakash, M. Kruse, J. Koo, B. Videau, P. Hovland, V. Taylor, B. Geltz, S. Jana, and M. Hall, "ytopt: Autotuning scientific applications for energy efficiency at large scales," *arXiv preprint arXiv:2303.16245*, 2023.

[55] C. Luo and R. Suda, "A performance and energy consumption analytical model for gpu," in *2011 IEEE ninth international conference on dependable, autonomic and secure computing*, IEEE. USA: IEEE, 2011, pp. 658–665.

[56] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent gpu architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. Cambridge, MA, USA: IEEE, 2013, pp. 673–686.

[57] R. Shrivastava and V. K. Nandivada, "Energy-efficient compilation of irregular task-parallel loops," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, nov 2017. [Online]. Available: https://doi.org/10.1145/3136063

[58] R. Barik, N. Farooqui, B. T. Lewis, C. Hu, and T. Shpeisman, "A black-box approach to energy-aware scheduling on integrated cpu-gpu systems," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 70–81. [Online]. Available: https://doi.org/10.1145/2854038.2854052

[59] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: Association for Computing Machinery, 2011, p. 12–23. [Online]. Available: https://doi.org/10.1145/2155620.2155623

[60] S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong, "Program energy efficiency: The impact of language, compiler and implementation choices," in *International Green Computing Conference*. Dallas, TX, USA: IEEE, 2014, pp. 1–6.

[61] Y. Arafa, A. ElWazir, A. ElKanishy, Y. Aly, A. Elsayed, A.-H. Badawy, G. Chennupati, S. Eidenbenz, and N. Santhi, "Verified instruction-level energy consumption measurement for nvidia gpus," in *Proceedings of the 17th ACM International Conference on Computing Frontiers*, ser. CF '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 60–70. [Online]. Available: https://doi.org/10.1145/3387902.3392613

[62] S. Puthoor and M. H. Lipasti, "Compiler assisted coalescing," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243176.3243203

[63] S.-C. Wang, L.-C. Kan, C.-L. Lee, Y.-S. Hwang, and J.-K. Lee, "Architecture and compiler support for gpus using energy-efficient affine register files," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, no. 2, nov 2017. [Online]. Available: https://doi.org/10.1145/3133218

[64] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. USA: IEEE, 2011, pp. 235–246.

[65] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on gpus," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. USA: IEEE, 2013, pp. 516–523.