Evaluating the Impact of Data Layout and Placement on the Energy Efficiency of Heterogeneous Applications

Apan Qasem
Dept. of Computer Science
Texas State University
San Marcos, TX 78666-4684
Email: apan@txstate.edu

Samuel Teich
Dept. of Computer Science
Texas State University
San Marcos, TX 78666-4684
Email: steich@txstate.edu

***Abstract*—Heterogeneous compute nodes have become an integral component of today's HPC systems. Recent research has established the importance of data layout and placement on such systems. This paper explores the power and energy aspects of data layout and placement on heterogeneous systems. We present results of an experimental study that evaluates the impact of data layout and placement on candidate HPC node architectures for kernels that exhibit a wide variety of performance characteristics.**

The results of the study show that data layout and placement can have a significant impact on the energy efficiency of heterogeneous applications. On some platforms, selecting the appropriate layout can yield up to an order-of-magnitude improvement in energy efficiency. The study shows that the conventional approach of using a structure-of-arrays for device-mapped data structures is not always profitable and that in addition to memory divergence, data layout choices are impacted by a variety of factors including arithmetic intensity and task granularity. The results of the study are used to establish a set of energy imperatives to guide data layout and placement across different architectures.

I. Introduction

There is mounting evidence that the first exascale systems will be highly heterogeneous where conventional high-performance CPUs work in concert with energy-efficient accelerators to yield scalable performance at prescribed power budgets [1], [2]. These emerging architectures embody heterogeneity in both computation and memory resources. The memory system contains many SRAMs, DRAMs and NVRAMs with different capacity, bandwidth and perceived latency. Heterogeneous memory systems add a new dimension to the already challenging problem of programming scalable systems.

Task (and data) mapping must consider not only the available computational resources but the capabilities of the memory units as well. For instance, if the computation patterns in a thread are such that the data is being accessed at a relatively high rate then ideally it should be mapped to a memory unit with higher bandwidth (e.g., HBM [3]). This placement must also be weighed against the computation capabilities and power demands of the processors attached to that memory. Furthermore, care must be taken to ensure that data is laid out in a way that favors the memory unit to which it has the highest affinity. For instance, an array of aggregate data types can be laid out with elements of the aggregate in consecutive memory locations (such as Array of Structures, AOS) which will favor the latency-optimized CPU memory hierarchy. On the other hand, a layout where elements from different aggregates are grouped and placed consecutively (Discrete Arrays, DA) will favor a GPU memory hierarchy with coalesced access for adjacent threads. In this context, we term the combined task of (i) selecting suitable layouts for data structures and (ii) controlling the placement of data segments in different memory units as data organization.

This was work was supported by the National Science Foundation through awards CNS-1305302 and CNS-1253292.

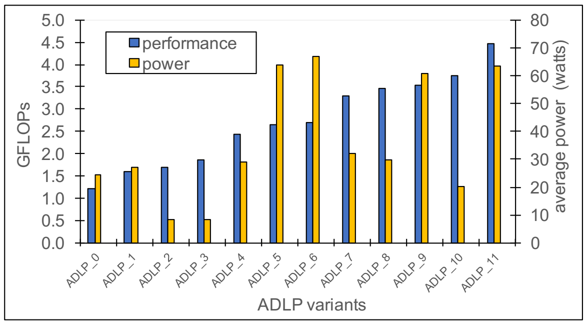


Fig. 1. Performance and power variation across ADLP (Adaptive Layout and Placement) configurations on a single node. ADLP_i refers to a particular layout and placement configuration on the target node

Consequences for inefficient data organization on heterogeneous memory systems can be dire, for performance and power. Fig 1 shows the performance and power variations of a heterogeneous image processing application for different data organization configurations. We observe up to a factor of five difference in performance and up to a factor of seven difference in power (ADLP_6 vs ADLP_2). Not surprisingly, no particular organization provides the best power-performance combination. It should be noted, that these numbers are for a single node only. The difference in performance and power will quickly escalate as we scale to larger systems.

This paper presents an extensive cross-platform empirical study that aims to understand the effects of data organization decisions on the energy and power consumption of emerging heterogeneous applications. The study considers two classes of heterogeneous memory systems mainly in use today: (i) integrated CPU-GPU with shared DDR memory, and (ii) discrete GPU with HBM or GDDR memory that is attached to the main DDR memory via PCIe. The discrete GPU systems provide limited software support for a shared virtual address space. We study the effects of traditional data layout schemes such as array of structures (AoS), discrete arrays (DA) and structure of arrays (SoA). We also consider advanced layout techniques that have been recently proposed for heterogeneous memory architectures. We analyze these data layouts along with the supported placement methods on each target platform. We characterize the main considerations for power and identify key program attributes that can potentially have the most impact on data layout and placement decisions. We construct a parameterized search space around these attributes and explore each dimension to determine how changes in each dimension affect power and energy consumption of the application.

To facilitate the exploration of the search space, we develop a micro-kernel generator and custom profiler. The kernel generator creates synthetic micro-benchmarks, each of which represent a specific point in the search space. Hardware performance counter based profiles are collected for each run and pushed to a profile database. The performance analyzer analyzes the collected data to identify patterns along each dimension.

The study reveals several interesting effects of data organization on power and energy. These include the interacting effects of memory divergence, arithmetic intensity and task granularity.

The rest of the paper is organized as follows: Section II provides some background on heterogeneous architectures and programming models; Section III presents related work on data layout optimizations on heterogeneous platforms; Section IV describes the data layout and data placement methods studied in this paper; Section V discusses application characteristics that are investigated in this study and the rationale for their inclusion; Section VI describes the experimental setup; Section VII presents experimental results and finally the key findings are summarized in Section VIII

II. Background

This section briefly reviews design of heterogeneous node architectures and corresponding programming models.

A. Heterogeneous node architectures

Today, heterogenous architectures for HPC systems can be mostly divided into two main classes: (i) integrated and (ii) discrete GPU architectures. In general, integrated GPUs and discrete GPUs differ in the physical memory organization and the underlying interconnect between the compute and memory elements in the system. Integrated GPUs typically share the physical memory space between the CPU and GPU cores and the memory is cache coherent. For added performance capability, some integrated GPU systems reserve a portion of the system memory to be exclusively accessed by the GPU without being cache coherent with the rest of the system. In contrast, discrete GPUs have a distributed physical memory architecture, where the GPU has its private non-coherent memory space and it communicates with the system memory via PCIe. Sharing the system memory directly with the GPU is done with driver support on demand by pinning parts of the host memory.

Furthermore, discrete GPU systems have two physical memory spaces – (1) system memory and (2) GPU memory, where the system memory can be shared with the GPU cores but the GPU memory is not accessible by the CPU cores directly. AMD software stack allows the system memory to be accessed by the GPU in either a fine-grained or a coarse-grained manner, whereas the Nvidia software stack allows the system memory to only be shared in a coarse-grained manner, where memory synchronization is explicitly performed with the `threadfence()` API[4].

B. Programming models

AMD systems use the Radeon^{TM}Open Compute Runtime (ROCR)[5], which is based on the Heterogeneous Systems Architecture (HSA^{TM}) specification [6], for program execution. HSA runtime is a thin, user-mode API that directly exposes the graphics hardware capabilities to the end user. HSA includes low-latency user-mode task dispatch, lightweight signaling to trigger kernel execution and query task completion, a rich memory management API and many more capabilities targeted at heterogeneous computing and memory environments. The memory API is used to allocate, move and de-allocate memory from various physical memory spaces.

CUDA is Nvidia's most widely used software platform, whose API can be used to manage memory on the device as well as share system memory between the host and the device. In this paper, we study the traditional host-device explicit data transfer model and the Unified Virtual Addressing (UVA) model that allows the pinned system memory to be directly accessed by the GPU cores. However, the very latest features of CUDA, such as Unified Virtual Memory and host-device page migration, are not studied in this paper and are considered as future work.

III. Related Work

Data layout transformations have long been a staple for compiler writers for improving data locality [7], [8], [9], and have seen use in additional homogenous memory architecture contexts such as the compression of sensor data [10]. We divide further discussion of recent work in data organization based on their applicability to discrete GPU and integrated CPU-GPU nodes.

A. Data Layout for Discrete GPU Nodes

Data re-organization techniques have been developed to reduce non-coalesced memory access on discrete GPU systems. Baskaran *et al.* present a polyhedral technique for determining suitable padding factors to improve shared memory locality [11]. Lee *et al.* describe a matrix-transpose transformation for improving memory coalescing in *thread-private* data structures. This layout conversion is implemented as a source-level transformation in their OpenMP-to-CUDA compiler framework [12]. Liu *et al.* describe a layout transformation that attempts to address channel skewing, non-coalescing and bank conflicts. The key idea is to divide arrays into blocks and map them to processing units with minimal overlap [13]. Wu *et al.* looks at the problem of data re-organization for irregular GPU applications. They show that in general the problem of minimizing non-coalesced memory accesses with data re-organization is NP-complete [14]. They propose two new algorithms that make appropriate trade-offs among time, space and complexity. One algorithm uses padding to avoid duplication while the other takes advantage of shared memory.

B. Data Layout for Integrated GPU Nodes

Work on data organization techniques for heterogeneous memory systems with shared memory is less common. Sung *et al.* propose a tiled layout for arrays of aggregate types and a runtime marshaling technique for in-place data-layout conversion [15]. They are able to achieve performance gains on four kernels on both NVIDIA and ATI GPUs. Although the strategy is aimed at heterogeneous shared memory systems the evaluation is done on discrete GPUs only. Che *et al.* describe Dymaxion, an extension of the CUDA API, that allows programmers to remap data layout structures to match the GPU memory [16]. They also provide an efficient method for performing remaps by overlapping them with PCIe transfers. The actual decisions for data mapping are left to the programmer, however. Majeti *et al.* describe a compiler framework for selecting between *AoS* and *SoA*. The selection heuristic is guided by meta information supplied by the programmer [17]. In more recent work, Majeti *et al.* present a greedy algorithm for selecting between AoS and SoA for array sections. They provide an automatic method for generating these layouts for CPU-GPU hybrid applications. However, the framework is evaluated on only one platform with a discrete GPU [18].

Two key aspects distinguish this work from earlier work in data organization. First, this is the first study to focus on data layout effects on power and energy. Second, unlike previous approaches, this work consider data layout and placement in conjunction.

IV. Data Layout and Placement

A. Data Layout Methods

Data layout refers to the organizational structure of data in memory and, while optimizing data layout with regards to various performance metrics is still an open problem, there are three data layouts for aggregate types that are commonly used in heterogeneous memory systems. These are (i) array of structures (AoS), (ii) structure of arrays (SoA) and (iii) discrete array (DA).

We provide brief descriptions of these data layouts next. To describe each layout, we consider an aggregate data structure containing n observations of k integer values.

1) Array-of-Structs (AoS): The AOS data layout bundles each observation as a discrete structure in series. Disregarding any padding added to align structure members along word boundaries, the AOS data layout ensures that features of any given observation are adjacent in memory, while subsequent iterations of any given feature will be separated in memory by a distance the size of an observation.

```
struct observation {
  int feature_1;
  int feature_2;
  int feature_3;
  ...
  int feature_k;
  };

struct observation AOS[n];
```

2) Structure-of-arrays (SoA): The SOA data layout, by contrast, bundles series of features. In memory, this guarantees that subsequent iterations of any given feature will be adjacent, while the constituent features of any given observation i will each occupy index i of that feature array. In addition feature arrays are adjacent in memory.

```
struct SOA {
  int feature_1[n];
  int feature_2[n];
  int feature_3[n];
  ...
  int feature_k[n];
  };
```

3) Discrete Array (DA): The DA data layout, similarly to the SOA data layout, bundles series of features, but does not encapsulate the data itself within a c structure. This implies that there are no guarantees about the location of a feature array in memory. Often the DA data layout is used rather than the SOA data layout due to language restrictions on dynamic allocation of arrays within structs.

```
struct DA {
  int * feature_1;
  int * feature_2;
  int * feature_3;
  ...
  int * feature_k;
  };

da.feature_1 = (int *) malloc(sizeof(int) * n);
da.feature_2 = (int *) malloc(sizeof(int) * n);
da.feature_3 = (int *) malloc(sizeof(int) * n);
...
da.feature_k = (int *) malloc(sizeof(int) * n);
```

B. Data Placement

Discrete GPU systems have two physical memory spaces, which allows for two data placement options for heterogeneous applications. A data structure can be explicitly copied to GPU device memory or it can be be mapped to page-locked system memory and accessed directly from the GPU using demand paging. We call these two placement method *DEV* and *HOST*, respectively.

Shared memory spaces may support *fine-grained* or *coarse-grained* accesses, where updates to memory in a fine-grained region are immediately visible to all devices that can access it, but only one device can have access to a coarse-grained allocation at a time. Synchronization in coarse-grained memory is performed through explicit memory fence operations, either by the driver or by the programmer. We refer to these two placement methods as *FINE* and *COARSE*, respectively.

V. Application Characteristics

In this study, we explore a range of application characteristics that have potential interplay with data layout and placement decisions. We develop a micro-kernel generator (Section VI) to create kernels that exhibit one or more of these properties in varying degrees.

A. Memory divergence

Most accelerators are equipped with coalescing units that attempt to combine memory requests emanating from threads in the same warp. If threads request data from the same cache line then the requests are coalesced into a single memory transaction. Un-coalesced memory requests increase memory traffic, potentially increasing power draw and simultaneously causing performance degradation. Below we present three scenarios for memory divergence and discuss how these can be addressed with data layout changes.

1) Indirect indexing: In the code example below, values in *indices[]* are being used to access into the *val* array.

```
  for(int k = 1; k < bound; k++) {
    j = indices[k];
    val = values[j];
    /* ... */
  }
```

Values in the indices array are not guaranteed to be contiguous, i.e. $indices[i] + 1 \neq indices[i + 1]$. Hence, *val* may not be accessed contiguously by consecutive threads.

2) Transposed access: This kernel code below uses two square matrices with column-major indices.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = 0;
for (j = 0; j < dimsize; j++)
if (i < dimsize) {
  B[i * dimsize + j] =  A[i * dimsize + j];
  /* ... */
}
```

As with the first example, cache misses occur due to non-contiguous access. In particular, the array access is strided within each warp, where the size between access is equal to dimsize. For a fully coalesced access pattern, the code would need to use row-major indices. That is for the above example that following must hold

$$B[j * dimsize + i] = A[j + dimsize + i]$$

3) Aggregate data access: Here we have code that manipulates images whose pixels are stored as an array of structs.

```
int tidx = threadIdx.x + blockDim.x * blockIdx.x;
for (int j = 0; j < NUM_IMGS; j++) {
  dst_images[j].r[tidx] = x;
  dst_images[j].g[tidx] = y;
  dst_images[j].b[tidx] = z;
  /* ... */
}
```

When a warp on the GPU stores the red value, the access pattern is once again strided, skipping the green and blue fields in between. This strided access pattern invokes memory divergence in the same way as example 2.

B. Arithmetic Intensity

Arithmetic intensity is defined as the number floating-point operations relative to the amount of data movement [19]. Arithmetic intensity is a key performance indicator for many classes of applications. It can also be a good indicator of power consumption. In general, higher intensity implies increased power consumption. On heterogeneous memory systems, however, the relationship between arithmetic intensity, data organization, and power-performance is more complex.

The GPU programming model relies on hiding memory latency with overlapped computation. If there is sufficient computation in the kernel to hide the excess latency caused by increased bandwidth pressure then no discernible performance loss will be observed. Thus, to clearly identify performance problems with memory divergence, the rate of computation in the kernel must be considered. Higher arithmetic intensity implies more computation in the kernel that can potentially hide the latency resulting from un-coalesced access. Thus, for high intensity kernels, memory divergence caused by data layout may not be as significant and layout decisions will be more heavily influenced by the cost of conversion.

C. Data Access Patterns

Data access patterns in the kernel dictate which segments of a shared data structure will be accessed by a thread or a thread block. Independent of data locality, access patterns can determine how much data is touched in a thread block (e.g., requirements for shared/scratchpad memory) and the amount of memory divergence. In this study, we focus on a particular class of data access patterns known as *sparsity*. Sparsity is defined as the distance between memory references in consecutive threads. Two threads are considered consecutive if they belong to the same warp/wavefront and their thread indices are separated by one. A sparsity value of k implies that consecutive threads are accessing values that are k bytes apart in memory.

D. Task Granularity

The next consideration for data layout efficiency is *task granularity* which refers to the amount of work done by each thread. Increased granularity implies that more data is being fetched and more computation being performed with the actual arithmetic intensity remaining constant. However, depending

TABLE I
CONFIGURATION OF EXPERIMENTAL PLATFORMS.

	AMD Kaveri	AMD Carrizo	Radeon Fiji	Nvidia Kepler
Architecture	integrated	integrated	discrete	discrete
Software	HSA	HSA	HSA	CUDA
CUs	8	8	64	14
Cores/CU			4 (SIMD)	192
GPU clock	720 MHz	800 MHz	1 GHz	811 MHz
GPU Mem	1 GB	none	4 GB	3 GB
Peak GPU BW	25 GB/s	31 GB/s	512 GB/s	134 GB/s
L1 cache	16 KB	16 KB	16 KB	16 KB
L1 cache line	64 B	64 B	64 B	128 B
PCIe BW	none	none	13 GB/s	8 GB/s

on the occupancy and wavefront scheduling, the effective rate at which data is requested can change. In the degenerate case, if computation from all warps occur in parallel (i.e., low occupancy) then an increase in task granularity will see a proportional increase in bandwidth pressure. The rate of increase in bandwidth pressure will drop as the occupancy increases. Increased bandwidth pressure, and consequently increased memory traffic, will increase application power draw.

Task granularity can be controlled by increasing the problems size, modifying the number of threads or by directly changing the amount of work done per thread. In our framework, we measure granularity as the number of objects processed per thread.

VI. EXPERIMENTAL SETUP

A. Evaluation Platforms

Table I describes the four heterogeneous platforms used in this study. Systems were selected with both integrated and discrete GPUs from two major vendors, AMD and Nvidia. The selected platforms also exhibit variations in the memory sub-system.

B. Micro-kernel generator

To characterize impact of data organization on heterogeneous memory, we developed a micro-kernel generator.

At its core, the kernel-generator contains a synthetic microbenchmark that performs 3 FP add and 2 FMA operations on an array of aggregate types. This baseline kernel is implemented with alternate data layout, placement and partition configurations. Each variant is then parameterized to expose several tunable parameters as listed in Section V. Seven of these are controlled by directly modifying the kernel while the rest are controlled externally through a driver program. The variants are constructed such that each dimension in the parameter space can be controlled independently, allowing us to examine the impact of each dimension in isolation, as well as explore the space in a non-orthogonal manner. Many of the parameters are controlled by manipulating loop headers. Loop bounds determine the degree while a custom function determines the thread index. This method can introduce overhead for some variants. This is accounted for during profiling.

Problem (N), grid (T) and workgroup (W) size are controlled in the driver program by replacing launch configuration parameters. T is chosen for a given N and W is selected to evenly divide T. Specific constraints imposed by the target architecture (e.g., $W > 32$) are also accounted for.

C. Profiler

The micro-kernel generator is coupled with a profiling tool which consists of an execution harness, a a HW performance counter reader and a performance analyzer. The tool is used to discover performance patterns in the parameter space and evaluate the relative effectiveness of a pair of data layout configurations. On invocation, the profiler first gathers pertinent architectural information such as memory capacity and available bandwidth. This information is then used to establish the range of each dimension in the parameter space. Some values are obtained directly (e.g., maximum number of threads), others via experimentation (e.g., when problem size exceeds memory capacity). The tool then proceeds to generate alternate kernel variants by systematically adjusting parameter values.

VII. EVALUATION

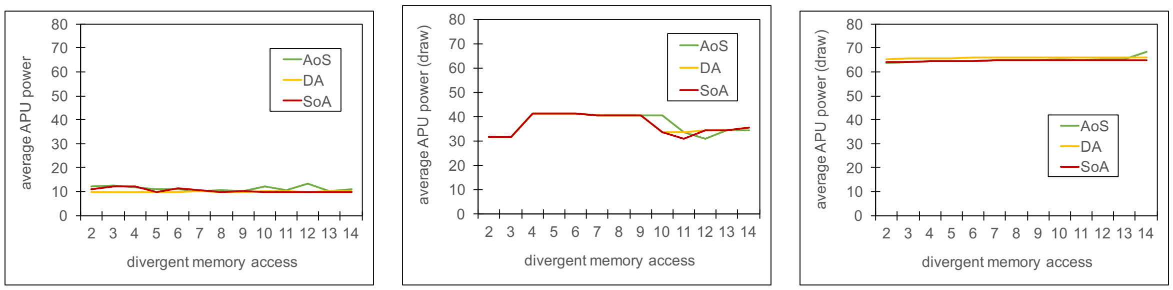
A. Memory Divergence

Fig. 2 shows how data layout choices impact power on three heterogeneous platforms. On *Carrizo* and *Kepler* there is very little variation in power consumption for different data layout choices. On *Fiji*, there is a band of memory divergence values for which the application operates at a higher power level with all three data layouts. This increase is mainly attributed to the increased rate of data transfer on the PCIE bus. Small values of memory divergence ($\leq$ 3) do not create sufficient pressure on the bus to substantially increase its power draw. On the other hand when the memory divergence is high, the application spends most of its time stalled in memory and is not able to consume data at a rate that would stress the bus. This is the reason for the higher plateau in the power values.

The impact of data layout on energy consumption can be seen in Fig. 3. Only *SoA* is able to maintain a consistent level of energy efficiency on all three platforms for the range of memory divergence values. *DA* maintains a consistent level on *Kepler* but experiences a gradual rise in energy consumption on *Carrizo* and *Fiji*. This increase was primarily due to performance loss due to increased register pressure causing spills to local memory. *AoS* performs worst in terms of energy efficiency. On *Carrizo* and *Kepler*, there is a linear increase in energy consumption as a function of the amount of memory divergence. On *Fiji*, the case is more severe, where its energy efficiency is worse by almost a factor of 10 for many values of the memory divergence.

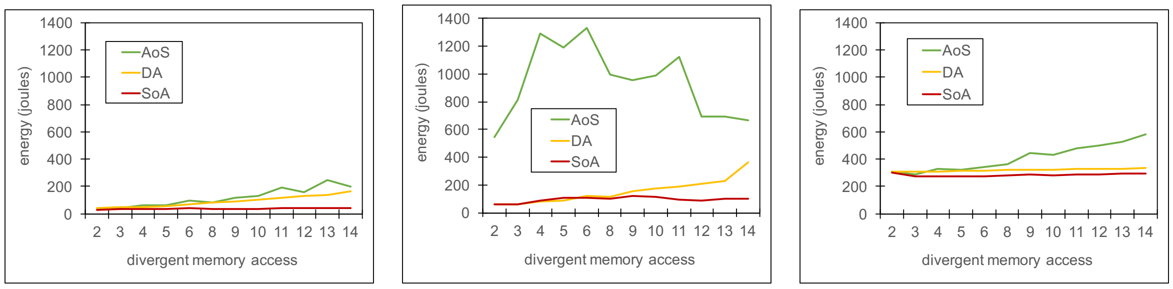
B. Data Access Patterns

To evaluate the impact of data access patterns, we consider kernels with sparsity values that range from 1 to 64. At the lowest sparsity value, contiguous memory locations are accessed consecutively, while at the highest value distance is equal to the size of the wavefront. On *Kepler*, the highest value is twice the size of a wavefront.



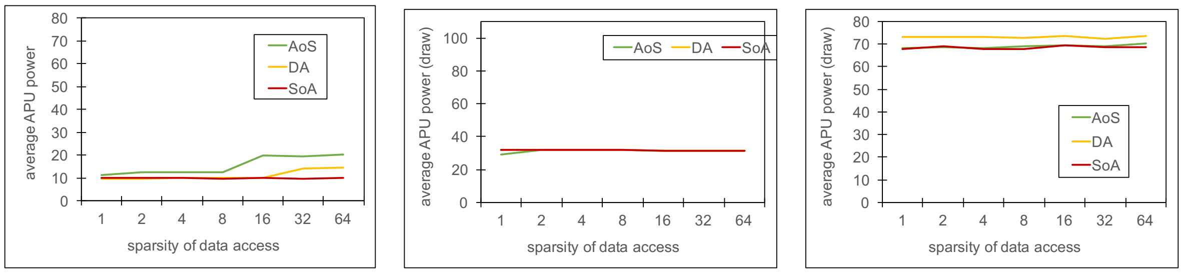
(a) Carrizo (b) Fiji (c) Kepler

Fig. 2. Impact of memory divergence on power



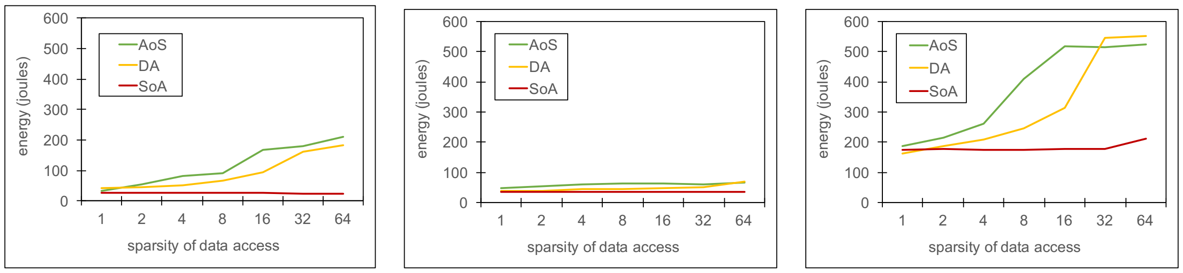
(a) Carrizo (b) Fiji (c) Kepler

Fig. 3. Impact of memory divergence on energy



(a) Carrizo (b) Fiji (c) Kepler

Fig. 4. Data access pattern and its impact on power

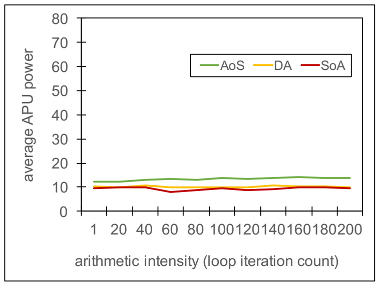


(a) Carrizo (b) Fiji (c) Kepler

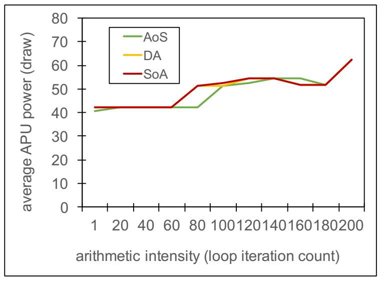
Fig. 5. Data access pattern and its impact on energy

Fig. 4 shows how data access patterns impact application power draw when *AoS*, *DA* and *SoA* data layouts are used. On *Fiji* and *Kepler*, application power draw is generally insensitive to data layout choice for all sparsity. Nevertheless, we do notice that *DA* incurs a higher power draw on *Kepler*. On *Carrizo*, there is a clear delineation point beyond which both *AoS* and *DA* incur a higher level of power draw. This rise can be attributed to increased local (for *DA*) and global memory (for *AOS*) traffic.

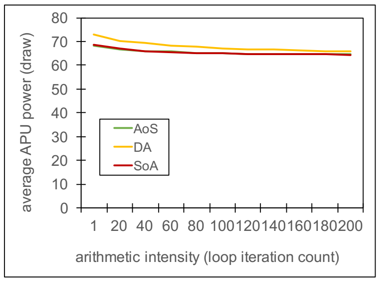
Fig. 5 shows how data access patterns impact application



(a) Carrizo

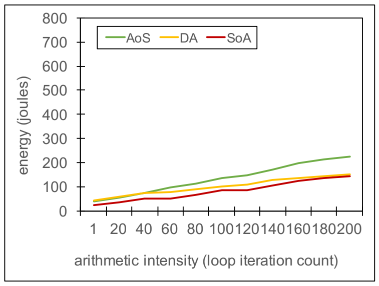


(b) Fiji

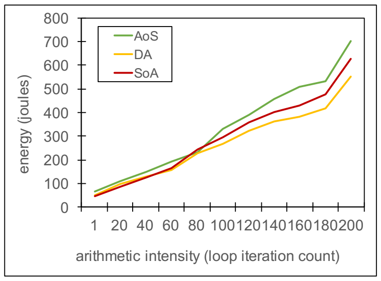


(c) Kepler

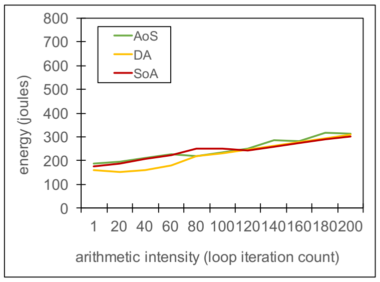
Fig. 6. Impact of arithmetic intensity on power



(a) Carrizo



(b) Fiji



(c) Kepler

Fig. 7. Impact of arithmetic intensity on energy

power draw when *AoS*, *DA* and *SoA* data layouts are used. Like memory divergence, *SoA* shows tolerance for different values of sparsity, delivering consistent energy efficiency across all three platforms. Both *AoS* and *DA* suffer from increased energy consumption for increased sparsity. This increase is less pronounced on *Fiji* than on *Kepler* and *Carrizo*. For large values of sparsity, *SoA*, can yield a factor of three energy improvement over the other two approaches.

C. Arithmetic Intensity

We explore a range of the arithmetic intensity. The lowest value represents a 25% utilization of compute units while the highest value represent full saturation.

Fig. 6 shows how arithmetic intensity impacts application power draw when *AoS*, *DA* and *SoA* data layouts are used. Similar to results observed in the memory divergence experiments, there is little variation in power on *Carrizo* and *Kepler* for any of the three layouts. However, *SoA*, again emerges as the layout of choice on these two platforms. On *Kepler*, *DA* incurs higher power draw while on *Carrizo*, *AoS* incurs a higher power draw. On *Fiji*, the power draw for all three layouts increases in a staggered fashion, with two inflection points being observed in the range of intensity values over which experiments were conducted.

The impact of data layout on energy consumption can be seen in Fig. 7. There is a gradual increase in energy consumption on all three platforms. This is expected because a higher arithmetic intensity implies more computation for the kernel which adds to execution time, leading to increased energy. Interestingly, there does not appear to be a clear winner in this case. On *Fiji*, *DA* is the best candidate while on *Carrizo*, *SoA* holds a slight advantage over *DA*. On *Kepler*, *DA* is preferred for lower values of arithmetic intensity but this advantage diminishes for higher intensity.

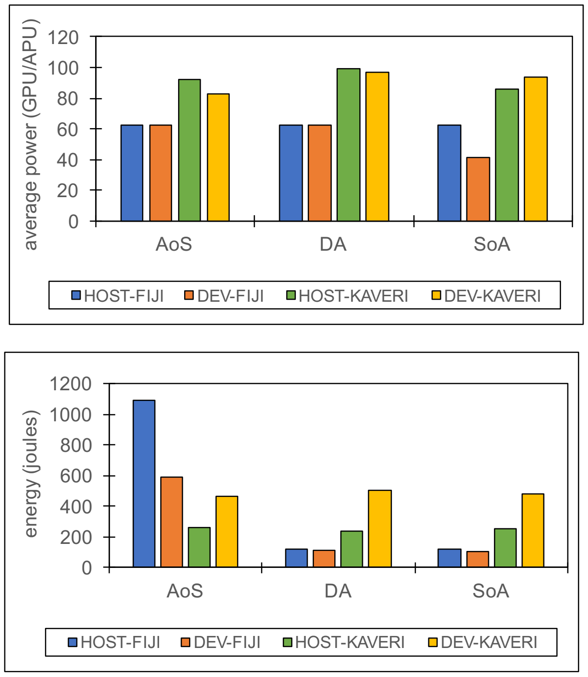


Fig. 8. Impact of data placement strategy on energy and power

D. Data Placement

In this section, we explore the effect of data placement. We assume the shared data structure is initially allocated in host memory. Then it is either copied to device memory (e.g., discrete GPU systems) or it is kept in host memory in a shared location. Since *Carrizo* does not have separate device memory, we evaluate the AMD *Kaveri* board.

Fig. 8 shows average power draw and energy consumption with different placement strategies. *Kaveri* incurs higher power draw than *Fiji*. We observe significant interplay between data layout and placement and application power draw. For instance, on *Fiji*, if the data is allocated to device memory then an *SoA* layout should be used. On the other hand, an *AoS* layout should be used for device-mapped data structures on *Kaveri*.

In terms of energy, *AoS* is clearly a poor choice on *Fiji* for both host and device-mapped data structures. Again, this is mainly due to the un-coalesced memory references that increases kernel execution time. Interestingly, on *Kaveri*, *HOST* placement is advantageous for all three data layouts.

VIII. Conclusions

This paper presented an experimental study that evaluated the impact of data layout and placement decisions on power and energy consumption. Applications were evaluated along several dimensions including memory divergence, arithmetic intensity and data access patterns. Experiments were conducted on four different heterogeneous platforms with different memory configurations.

Below are the key findings of this study

Data layout has low impact on application power draw: Generally, application power draw appears insensitive to changes in data layout and placement. *Fiji* is a notable exception, displaying increased levels of power draw with changes in memory divergence and arithmetic intensity.

Data layout and placement has significant impact on energy efficiency: Considerable variation in energy consumption was observed for different layout strategies on all platforms. On some platforms the best and the worst configuration differed by an order of magnitude.

Layout and placement decision must be taken in concert: For improved energy efficiency, it is imperative that data layout decisions consider the placement of data. For instance, for *device* placement on discrete GPUs, an *SoA* layout should be chosen, while a discrete array layout is preferred for *host* placement.

SoA is not always the optimal choice for device-mapped data structures: When considering performance, it is generally advisable to use SoA for GPU data structures over other layouts. However, our study shows that, when considering energy efficiency, SoA may not always be the best choice. For instance, the study shows that for lower levels of arithmetic intensity a discrete array layout can produce more energy efficient kernels on heterogeneous systems with discrete GPUs.

References

[1] "Top 500 Supercomputer Sites : 2005-2013," http://www.top500.org.

[2] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers, "Achieving exascale capabilities through heterogeneous computing," *IEEE Micro*, vol. 35, no. 4, pp. 26–36, July 2015.

[3] J. Kim and Y. Kim, "HBM: Memory solution for bandwidth-hungry processors," in *A Symposium on High Performance Chips (HOTCHIPS)*, 2014.

[4] NVIDIA, "CUDA Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[5] AMD, "Radeon Open Compute Runtime 1.2," https://radeonopencompute.github.io/.

[6] HSA Foundation, "HSA Runtime Specification 1.1," http://www.hsafoundation.com/standards.

[7] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, ser. CC'11/ETAPS'11, 2011.

[8] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T. f. Ngai, "Data layout transformation for enhancing data locality on nuca chip multiprocessors," in *18th International Conference on Parallel Architectures and Compilation Techniques (PACT09)*, 2009.

[9] R. M. Yong Li, Ahmed Abousamra and A. K. Jones, "Compiler-assisted data distribution and network configuration for chip multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 11, pp. 2058–2066, 2012.

[10] W. O. C. Haifeng Xu, Yong Li, L. A. Schaefer, M. M. Bilec, A. K. Jones, and A. E. Landis, "Improving efficiency of wireless sensor networks through lightweight in-memory compression," in *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*, 2015, pp. 1–8.

[11] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for gpgpus," in *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008, Island of Kos, Greece, June 7-12, 2008*, 2008, pp. 225–234.

[12] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP09, 2009.

[13] J. Liu, W. Ding, O. Jang, and M. Kandemir, "Data Layout Optimization for GPGPU Architectures," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, 2013, pp. 283–284.

[14] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[15] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data Layout Transformation Exploiting Memory-level Parallelism in Structured Grid Many-core Applications," in *19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT10, 2010.

[16] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 13:1–13:11.

[17] D. Majeti, R. Barik, J. Zhao, M. Grossman, and V. Sarkar, "Compiler-Driven Data Layout Transformation for Heterogeneous Platforms," in *Euro-Par 2013: Parallel Processing Workshops*, 2013, pp. 188–197.

[18] D. Majeti, K. S. Meel, R. Barik, and V. Sarkar, "Automatic data layout generation and kernel mapping for cpu+gpu architectures," in *25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: ACM, 2016, pp. 240–250.

[19] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, April 2009.