Experimental evaluation of memory optimizations on an embedded GPU platform

Innocenzo Mungiello
University of Naples Federico II

Abstract—This paper presents the experimental evaluation of data mapping techniques in the shared memory of an embedded GPU. The evaluated technique, previously presented in the literature in other contexts, aims at partitioning an array across the shared memory physical banks, so as to increase parallel accesses, resulting in appreciable gains in terms of both performance and energy efficiency. The paper presents the experimental setup used for characterizing physically the behavior of the platform, allowing a validation and a closer understanding of the evaluated memory mapping technique.

I. INTRODUCTION

Because of the end of Dennard Scaling, since the early 2000s manufacturers have been forced to explore new architectural paradigms, including multi- and many-core solutions as well as Graphics Processing Units (GPUs), now being used for a variety of applications including the embedded domain [1]. In addition, Field-Programmable Gate Arrays (FPGAs) also provide a solution for building customized computing platforms, although they pose non-trivial challenges mostly involving complexity, high-level design as well as interplay with suitable programming models [2], [7], [8], [12], somewhat mitigated by GPUs. A crucial aspect of recent GPU and FPGA architectures is their power-efficiency, i.e., the rate of computation that can be delivered by a system for every watt of power consumed, measured in GFLOPS-per-Watt, today considered a primary performance metric for the scalability of computing platforms.

An interesting observation is that in current computing technologies a major component of power consumption is due to data movement rather than mere processing, which is indeed a widely studied subject in a number of different contexts [11], [5] In particular, looking at today’s GPUs, the power contribution of data movement compared to processing can be as high as 85%. This work thus addresses the impact on performance and energy efficiency of memory optimization techniques for heterogeneous architectures, particularly GPUs, in an embedded setting. In particular, the paper focuses on the experimental evaluation of data mapping approaches previously developed in the area of non-uniform memory architectures and transferred to the case of GPUs. The emphasis is on the multi-bank organization of the on-chip GPU shared memory, where the whole shared addressing space is partitioned in a cyclic way and is potentially subject to an access conflict problem. We evaluate a data layout transformation impacting access patterns, which enables a significant gain in terms of both performance and energy efficiency, as confirmed by an experimental set-up used for performing measures on a physical embedded platform.

The paper is structured as follows. Section II discusses the background and the motivation of this work. Section III recapitulates the approach adopted for data partitioning. Section IV shows the set-up used for the experimental evaluation. Section V concludes the paper with some final remarks.

II. BACKGROUND

Memory mapping has traditionally been an important optimization problem for high-performance parallel systems [11]. Today, these issues are increasingly affecting a much wider range of platforms. In fact, many medium/high-end embedded systems are now based on parallel compute architectures while, at the opposite end of the spectrum, large datacenters currently play a central role for popular cloud-based applications, with a whole range of new disparate challenges, from architecture optimization to security as well as workflow management and validation [19], [17], [29], [32]. Although here we are fundamentally interested in the embedded architecture level, all such platforms are characterized by inherently the same issue concerning the memory infrastructure organization, i.e. the fact that, at the low-level, they are based on non-uniform memory access (NUMA) which, depending on the application access patterns, may be critical to the overall performance. In fact, the NUMA model reflects a scenario where multiple independent processing cores/nodes with local memory modules are connected by some form of interconnect, causing the access time to depend on the location relative to the processor placing the access operation [6]. A closely related concept, distributed shared memory (DSM), is a form of memory architecture where physically separate memories can be addressed as one logically shared address space. DSM systems combine the best features of shared-memory and distributed-memory machines. They support the convenient shared-memory programming model on scalable distributed-memory hardware, exposing a simpler abstraction for data passing to the application programmer. Furthermore, many distributed parallel applications execute in phases, where each computation phase is preceded by a data-exchange phase. The time needed for the data-exchange phase is often dictated by the throughput limitations of the communication system. Distributed shared memory algorithms typically move data on demand as they are being accessed, eliminating the data-exchange phase, spreading the communication load over a longer period of time, and allowing for a greater degree of concurrency. Also, the total amount of memory may be increased proportionally, reducing paging and swapping activity [26], [31]. However, although many DSM systems have been proposed and implemented (see Bal et al. [16], Bershad et al. [20], Chase et al. [21], Dasgupta et al. [22], Fleisch and Popek [23], Li and Hudak [26], Minnich and Farber [27], and Kirk L. Johnson et al. [33]), achieving good performance on DSM systems for a sizable class of applications has proven to be a major challenge [15]. One of
the key problems in building an efficient software DSM system is to reduce the amount of communication needed to keep the distributed memories consistent. Often, the proposed solutions result in a trade-off between performance and consistency models, with the aim of enhancing the concurrency available in the distributed shared memories [37]. Another problem is to avoid access conflicts to physically different memory banks from multiple threads/processes running concurrently. This problem can impact greatly the performance of the system, especially in distributed systems, since it causes serialized accesses and a significant interconnect overhead. A large number of works addressed this problem, e.g. Das et al. [28] considered the star-template access on two specific host topologies, tori and hypercubes, enabling conflict-free mappings using an optimal or provably good number of memory modules. Monchiero et al. [30] propose a mechanism for data allocation on a distributed shared memory space, dynamically managed by an on-chip hardware memory management unit. Sung et al. [10] present automatic data layout transformation as an effective compile-time performance optimization for memory-bound structured grid applications.

A. Memory in heterogeneous parallel systems

As implied by the above introduction, traditional solutions for memory mapping optimization in NUMA contexts such as parallel computer/datacenters might play a key role also in today’s embedded platforms. In fact, many of such computing platforms are increasingly being provided with high-end GPU and/or FPGA units, where parallel processing elements access simultaneously several independent memory banks through complex interconnects. This potentially provides an opportunity for improving the memory bandwidth available to the application, provided that one adopts suitable memory partitioning strategies based on the actual access patterns [5]. In particular, GPUs represent today a major paradigm shift in computing architecture focusing on increasing the execution throughput of parallel applications. A current representative instance of this trend is the NVIDIA GeForce GTX680 graphics processing unit (GPU) with 16384 threads, executing in a large number of simple, in-order pipelines. As of 2012, the ratio between many-core GPUs and multi-core CPUs for peak floating-point calculation throughput was about 10 to 1 [4], mainly due to the differences in the fundamental design philosophies between the two types of processors, as illustrated in Figure 1. This approach is natively supported by the Compute Unified Device Architecture (CUDA) programming model, introduced by NVIDIA in 2007, and is reflected by more recent programming models such as OpenCL (now at version 2.0), OpenACC, C++ AMP and OpenMP 4.0.

For performance purposes, the on-chip shared memory of a GPU device is normally divided in banks, which can be accessed in a parallel way from all the threads in a warp. The number of banks is strictly dependent on the architecture. As an example, in an NVIDIA Kepler architecture the number of banks is 32 and each bank has a word of 8 bytes. Data allocated in the shared memory are cyclically distributed over the banks in two ways:

- **4-byte access**: Successive 4-byte words are mapped to successive banks. The memory can be seen as made of 32 banks, each 4-byte wide. If the data that we use is 8 bytes wide, the access mode becomes an 8-byte access.
- **8-byte access**: Successive 8-byte words are mapped to successive banks.

We can easily compute the bank where a data is mapped to, as follows:

- \((8\text{B word index}) \mod 32\);
- \((4\text{B word index}) \mod (32 \cdot 2)\);
- \((\text{byte address}) \mod (32 \cdot 8)\)

Figure 2 shows an example of data mapping on shared memory with both modes. In this example the data are 4 byte-word index and, for simplicity, we show only four banks.

---

**Fig. 1.** CPU architecture vs. GPU architecture

---

**Fig. 2.** Comparing bank mode mappings. On the left-hand side we have a 4-byte access. On the right-hand side we have an 8-byte access.

The 4-byte access is the default mode. We can change it by using the CUDA function `cudaDeviceSetSharedMemConfig(param)` where `param` can be:

- `cudaSharedMemBankSizeEightByte`, or
- `cudaSharedMemBankSizeFourByte`.

The access mode can affect the performance of a kernel. In fact, allowing all the threads in a warp to fetch data in parallel from this multi-bank memory can lead to great performance improvements, but it requires explicit, bank-aware organization of the data layout. There are three main mechanisms for shared memory access which guarantee improved performance:

- **Unicast**: each thread in a warp tries to access a different location stored in a different bank.
- **Multicast**: One or more groups of threads in a warp try to access the same location stored in one of the banks. The other threads perform a unicast-style access.
- **Broadcast**: Every thread in a warp will access exactly the same location, obviously stored in the same bank.
These three mechanisms are enabled by an interconnection network which links each core of a Streaming Multiprocessor to the shared memory. By using this interconnection network and performing one of these access patterns, data can be retrieved at low latency as they were stored in the registers. If the pattern is different from those described above, shared memory performance may easily decrease. In particular, in the case where two or more threads in the same warp try to access different words stored in the same bank, the interconnection network is no more able to provide the required data to all the threads in parallel. This situation, called bank conflict, is a major problem related to the use of the on-chip shared memory. In particular, if two threads try to access different words stored in the same bank, we have a 2-way bank conflict. If three threads try to access different words stored in the same bank, a 3-way bank conflict occurs, and so on. The worst case involves all 32 threads in a warp trying to access different words stored in the same bank, causing a 32-way bank conflict. Whenever a conflict occurs, it is resolved by serializing the accesses in time. As an example, the serialization in a 2-way scenario leads to doubled latency and can increase the energy consumption considerably.

### III. Data Layout Transformation

As already mentioned above, heterogeneous platforms provide some form of customizability that can be effectively exploited to improve performance and power efficiency. While FPGAs offer the possibility to jointly customize the memory infrastructure architecture and the application task mapping, by using one of several approaches in the literature [5], [14], GPUs have still enough flexibility to expose the physical shared memory bank structure to the programmer, enabling bank mapping to be tailored on the application access patterns. Here we shortly review a literature approach to data mapping in parallel architectures [11], that can be effectively applied to the case of the GPU shared memory, as shown in this paper.

The results presented here apply to affine static control parts (SCoPs), i.e., code segments in performance-critical loops where loop bounds, conditionals, and subscripts of memory references are affine functions of the surrounding loop iterators and of constant parameters possibly unknown at compile-time. For each reference to an array $A$ in the loop nest, call memory access function a correspondence $F$ associating each element of $A$ with a value of the iteration vector $\overline{\nu}$, which is the vector having as elements the indices of the loop nest containing the reference. Since the subscripts in SCoP code are affine functions, $F$ can always be expressed as $F = F \cdot \overline{\nu} + \overline{\tau}$, where $F$ is an integer matrix and $\overline{\tau}$ is a constant displacement.

Furthermore, we need a mathematical formulation of the bank mapping in the GPU shared memory. The cyclic scheme described in the previous section can be seen as a special case of an allocation expressed as a modular mapping function $\sigma(\overline{\nu}) = M \cdot \overline{\nu} \mod m$, associating each index $\overline{\nu}$ of an array element having $p$ components with the corresponding bank [11]. $M$ is a $p \times n$ integer matrix, $\overline{m}$ is $p$-dimensional array of integer moduli,1 and the modulo operation is component-wise. We regard the physical banks making up the GPU shared memory as a linear array, hence $(p = 1)$. Assume that we have a bi-dimensional array to allocate and let $\begin{bmatrix} x \\ y \end{bmatrix}$ be the indices of its elements. The mapping problem can thus be expressed as:

$$ Bank(x, y) = M \cdot \begin{bmatrix} x \\ y \end{bmatrix} \mod \overline{m} $$

where $\overline{m}$ is in fact mono-dimensional and coincides with the number of available banks, denoted banks. The value of this constant depends on the specific GPU architecture. For instance, banks = 32 for the NVIDIA Kepler family.

An example of matrix $M$ is:

$$ Bank(x, y) = \begin{bmatrix} 1 & N \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \mod \text{banks} $$

where $N$ is equal to the size of the array along the $x$ dimension.

Table I provides an example for a $52 \times 52$ array, highlighting the cyclic scheme followed by data allocation.

<table>
<thead>
<tr>
<th>Table I: Mapping of a $52 \times 52$ Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>52</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Table II: Example of a Conflict

The repetition of the values 0, 1, 2, 3 causes here a 2-way conflict. To avoid conflicts, no repetitions must occur in the rectangular domain. Equivalently, the bank mapping function corresponding to the memory reference in the threads must be injective in the rectangular domain covered by the warp.

We consider the class of transformations to SCoP code that change the memory access function by multiplying its expression by a matrix $T$ [3]:

$$ T = \begin{bmatrix} a & b \\ c & d \end{bmatrix} $$

which also implies changing the layout in memory of the locations concurrently accessed by the threads in a warp. A new allocation can be defined as:

$$ Bank(x, y) = \begin{bmatrix} 1 & N \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \mod \text{banks} $$

1In general, the set of banks may have a dimensionality equal to $p$, so that $\sigma$ returns a $p$-dimensional bank index. Modular mappings can change the dimensionality of the data address.
whether a given transformation induced by accesses conflict-free dimension of a warp, the transformation ensures injective function over the rectangular domain identified by the transformation matrix $T$. We rely on the formal treatment in [3] to check whether a given transformation induced by $T$ is injective.

Below we exemplify the procedure used to address the problem of bank conflicts. We consider an algorithm performing matrix multiplication. In our quantitative experiments, we choose $52 \times 52$ tile dimensions. Consequently, the code snippet is as follows:

```cpp
__shared__ double AS[2704];
__shared__ double BS[2704];
//Calculate the row index of the C element and A
int Row = blockIdx.x*blockDim.x+threadIdx.x;
//Calculate the column index of C an B
int Col = blockIdx.y*blockDim.y+threadIdx.y;
if ((Row < WIDTH) && (Col < WIDTH)) {
    double Cvalue = 0;
    // each thread computes one element
    // of the block sub-matrix
    #pragma unroll
    for (int k = 0; k < WIDTH; k++)
        Cvalue += AS[(Row*WIDTH)+k]*BS[k*WIDTH+Col];
    C[Row*WIDTH+Col] = Cvalue;
}
```

We can actually define a block, not a warp. But if the block is composed by only 32 threads, then, we are defining the shape of the warp too.

This transformation completely clears the conflict problem. As an example, the previous two accesses performed by iterations with $k=0$, Col=0, and Row=4/Row=12 now become $AS[212]/AS[636]$, accessing banks 212 mod 32 = 20 and 636 mod 32 = 28, respectively, but the formally proved injectivity condition ensures that this occurs for any thread pair in the rectangular warp. Notice that the conflict-free condition comes at the cost of stretching the memory region covered by matrix AS, requiring interleaved placement of other data structures for efficient utilization of the shared memory. Furthermore, the number of arithmetic operations involved in address computation might increase, making the trade-offs with time and energy efficiency less obvious. The experimental evaluation carried out in the next section provides some insights about the overall benefit of the adopted technique.

IV. EXPERIMENTAL EVALUATION

In this section we present a set-up used to carry out the experimental evaluation of the above optimization technique and collect performance/power data from a physical platform.

A. System Overview

The diagram shown in Figure 3 is a schematic representation of our environment. We have a host PC running a WMware Virtual Machine with Ubuntu 14.04 and the JetPack installed on it. In this environment, we use NVIDIA Nsight to write CUDA code and, then, to compile and remotely run it on a Jetson TK1 development board. On the same machine we have a Windows Operating System with Digilent WaveForms application installed on it. We use this application as a data logger. The data are collected by the Digilent Analog Discovery suitably connected through Channel 1 wire probes to the $R5C1$ resistor, available on the Jetson board in order to control the power consumption of the overall platform, by measuring the voltage across this resistor. Figure 4 shows the real system used in our experimental setup.
B. Results

In this section we discuss the transformation results. By using the system described in IV-A it was possible to evaluate the impact of the transformation on power consumption and execution time. Figure 5 shows the results collected using the Analog Discovery.

The instrument does not allow us to differentiate the consumption, in terms of watt, of the two versions, as both versions reach a value of around 0.80 W. On the other hand, we can appreciate that the execution time is drastically reduced. Figure 5 also shows that the optimized kernel takes about 5 seconds, instead the non-optimized takes 6.5 seconds. This means that the optimized kernel consumes around 0.80 W × 5 s ≃ 4 J, instead the non-optimized consumes around 0.80 W × 6.5 s ≃ 5.2 J, totalling a difference of 1.2 J, i.e. a gain of 23%.

As mentioned in the previous section, the transformation solves the conflict issue, but at the cost of increasing the arithmetic operations to be performed and the amount of shared memory to allocate. Table III shows the differences, in terms of number of instructions executed by the two kernels. The difference is 1040000 instructions, i.e. a 3.84% overhead. As for the amount of memory to allocate, the optimized kernel requires 808 bytes, or 3.73% more. In terms of performance per watt, the kernel executes 56243200 floating point operations in double precision. So, for the non-optimized kernel we have a value of 10.816 MFLOPS/watt (referred to the subset of the GPU actually used), while the optimized one has a value of 14.0608 MFLOPS/watt, i.e. we have an increase of 30% with this transformation.

V. Conclusions and future work

GPUs have become extremely important for today’s HPC and Cloud Computing as they provide an effective answer for increasingly stringent energy constraints. This work presented a practical experience centered around the evaluation of an optimization technique for GPU on-chip memory. The experimental results collected pointed out the significant incidence that such techniques may have on both execution time and energy efficiency. As a part of our future work, we plan to build a complete heterogeneous platform pairing embedded GPUs with FPGA units for a range of applications including multimedia, testing, and security [25], [24], [34], allowing an extensive evaluation of optimization techniques combining special-purpose acceleration and software heterogeneous programming.

The presentation of this work is supported by the European Commission in the framework of the H2020-FETHPC-2014 project n. 671668 - MANGO: exploring Manycore Architectures for Next-Generation HPC systems.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Number of Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatMul_32_Optimized</td>
<td>28070952</td>
</tr>
<tr>
<td>MatMul_32_No_Optimized</td>
<td>27030952</td>
</tr>
</tbody>
</table>

TABLE III. COMPARISON OF THE NUMBER OF INSTRUCTIONS

REFERENCES


