Efficient OpenMP Data Mapping for Multicore Platforms with Vertically Stacked Memory

Andrea Marongiu, Martino Ruggiero, Luca Benini
DEIS - University of Bologna
Viale Risorgimento, 2 - 40136 Bologna - Italy
Email: {a.marongiu, martino.ruggiero, luca.benini}@unibo.it

Abstract—Emerging TSV-based 3D integration technologies have shown great promise to overcome scalability limitations in 2D designs by stacking multiple memory dies on top of a many-core die. Application software developers need programming models and tools to fully exploit the potential of vertically stacked memory. In this work, we focus on efficient data mapping for SPMD parallel applications on an explicitly managed 3D-stacked memory hierarchy, which requires placement of data across multiple vertical memory stacks to be carefully optimized. We propose a programming framework with compiler support that enables array partitioning. Partitions are mapped to the 3D-stacked memory on top of the processor that mostly accesses it to take advantage of the lower latencies of vertical interconnect and for minimizing high-latency traffic on the horizontal plane.

I. INTRODUCTION

Three-dimensional (3D) stacking technology has recently risen to the research forefront as one of the most high-potential technology innovations for many-core integrated platforms, both in general purpose and embedded computing [11], [12], [6], [4]. 3D integration technology provides a number of means to overcome the scalability limitations imposed on many processor designs as 2D technology reaches the nanometer scale. It gives the opportunity to revisit the traditional architectural tradeoffs based on the evidence that the processor and memory sub-systems had to be placed side by side. In 3D stacking they can be placed on top of each other, and linked through vertical interconnects which are more than two orders of magnitude more energy-efficient and denser than the most advanced off-chip I/O channels.

The main benefit of this disruptive technology in high-end embedded computing is to enable the construction of many-core data-processing systems with low latency and high bandwidth access to multiple, large DRAM banks in close spatial proximity. The availability of such an efficient physical layer for processor-to-memory communication and of an enormously increased amount of space in tightly coupled memories will trigger deep changes in high-performance embedded programming. In a nutshell, 3D integration enables distribution in space not only of computation but also of main memory storage to an unprecedented level. Clearly, this brings new distinctive compile- and run-time software development challenges which are just starting to be assessed by the scientific community.

Our first goal is to define a conceptual framework to address these challenges. We model a vertically stacked memory system with the abstraction of memory neighborhood: each physical processing element in a large many-core array has fast, large-bandwidth access to a vertical stack of memory banks on top. The processor can also address (in a globally shared memory model) vertical stacks on top of other processors, but corresponding memory transactions will have to be transported through a horizontal on-chip interconnect fabric, typically a Network-on-chip (NoC). This implies a notion of distance: the cost (increased latency and decreased bandwidth) of a memory access sharply increases as we move to memory neighborhoods to far away processors. Fig. 1 depicts a high-level view of a 3D integrated architecture and its memory neighborhoods. In this work we focus on a concrete embodiment of this model targeting embedded computing, namely a 3D-integrated platform for multi-dimensional array processing (e.g. antenna arrays, radar images, video images) with explicitly managed data memories.

Typical applications in the domain of array and image processing require the implementation of algorithms for enhancement, analysis, synthesis of multidimensional arrays of “pixel” data. Many of these algorithms are amenable to SPMD (Single Program, Multiple Data) parallelization, based on decomposition of data array processing across parallel threads. Mapping this computation model onto a 3D-stacked memory architecture requires careful data placement across multiple physical memories. Placing entire arrays onto a single shared memory encounters scalability problems. Moreover, accessing remote memory staks induces severe latency overheads. These issues can be efficiently addressed by partitioning data and placing each partition onto the DRAM neighborhood of the processor that mostly references it.

The main technical contribution of this work is to give clear evidence that 3D-memory aware programming model and application development environment is critically required to achieve high execution efficiency on a vertically integrated embedded multicore platform. We precisely limit the scope of our approach to SPMD-type parallel applications targeted on a MPSoC with explicitly managed memory hierarchy. In this context, our work addresses three key issues: (i) providing a high-level OpenMP-based programming model which supports array data partitioning across clustered vertical DRAM stacks; (ii) developing compile-time support (based on the GCC compiler infrastructure) for feedback-based data placement on 3D-stacked main memory; (iii) streamlining application execution with an efficient run-time support. Experimental results demonstrate that a neighborhood-aware software development environment can boost application execution efficiency by up to 6,25×.

II. RELATED WORK

Recently, several 3D memory designs have been announced, confirming the benefits of 3D technology for high-efficiency next-generation memory systems[1][2][3]. The benefits of 3D memories have mostly been explored for high performance systems [4][5][3]. Kgil et al. [4] present a high performance server architecture where DRAM is stacked on a multicore
processor chip. Overall power improvements of 2 – 3× with respect to a 2D multi-core architecture are reported. Similarly, [5] presents a 3D stacked memory architecture for CMP. By changing the internal DRAM architecture (based on true-3D memory organization proposed by [3]), the author claims a 75% speedup. On the industrial front, many companies, including industry leaders IBM and Intel are active in technology and architecture exploration [6][7][8][9]. On the research side, Li et. al investigate in [10] the challenges for L2 design and management in 3D chip multiprocessors. Their term of comparison is 2D NUCA systems, which employ dynamic data migration to place more frequently-accessed data in the cache banks closer to the processor. Experiments show that a 3D L2 memory design with no dynamic data migration generates better performance than a 2D architecture that employs data migration.

3D memory integration is also actively explored in the embedded computing domain. All major players in the mobile wireless platform markets are very actively looking into how to integrate memories on top of MPSoC platforms for next-generation hand-held terminals [11]. More in general, the system size reduction, coupled with orders-of-magnitude improvements in memory interface energy efficiency are key enablers for disruptive innovation in embedded computing [12], possibly even more than in performance-centric general-purpose computing. In [13], Ozturk et al. explore core and memory blocks placement in a 3D architecture with the goal of minimizing data access costs under temperature constraints. Using integer linear programming, the best 2D placement vs the best 3D placement are compared. Experiments with both single-core and multi-core systems show that the 3D placement generates much better results (in terms of data access costs) under the same temperature bounds.

Programming models to explicitly take into account the memory hierarchy at the application level are presented in [15][17]. Sequoia[15] is a programming language that abstractly exposes hierarchical memory in the programming model and provides language mechanisms to localize computation to particular memory locations. Unified Parallel C[17] is a parallel extension of the C programming language intended for multiprocessors with a common global address space (GAS). The shared space is partitioned into portions each of which has affinity (or logical association) with a given thread. The programmer can declare a partitioning scheme for arrays, and specify the affinity under which to execute loops.

Shared data partitioning in programming environments for NUMA multiprocessors has been widely studied in the past[18], and in particular there is a vast amount of literature dealing with the integration of such techniques in OpenMP[19][20][21]. Here OpenMP API extensions to enable distribution of shared arrays over multiple memories in a NUMA architecture are presented, which are similar to ours. However, the differences at the architectural level between traditional NUMA multi-processors and embedded many-cores with vertically stacked memory, are very significant and have far-reaching consequences. Traditional NUMA machines were organized as clusters of computing nodes (e.g. the SGI Origin), where inter-node communication has orders-of-magnitude lower speed than local operations. Remote memory access took place under heavyweight software abstractions such as virtual memory paging. The cost associated to such a memory management layer is hidden behind the huge communication cost. In embedded MPSoCs paged virtual memory is not supported in hardware as it would be way too expensive to replicate a complex MMU for all the element of a large-scale on-chip data-processor array. Moreover, all communication is through tightly coupled tiers, where latency is much lower and bandwidth is much higher. These major differences lead to completely different set of implementation choices, that will be described in details in the following sections.

III. TARGET 3D ARCHITECTURE

The platform template targeted by this work is the 3D-stacked MPSoC depicted in Fig.1. The bottom layer hosts the processing elements of the chip, while the others are composed by DRAM memory banks[5]. Each vertical stack features a bank of private memory, only accessible from the local processor, and a bank of shared memory. The collection of all the shared segments is organized as a globally addressable NUMA portion of the address space. In the considered 3D template, memory and CPUs are allocated onto different layers, but our software framework can be applied to different stacking approaches. The bottom layer in Fig.1 illustrates the block diagram of the 2D multi-core subsystem. It is made by several computational tiles composed by a RISC-like CPU and a small amount of local memory (SPM, caches). The interactions between CPUs and memories are done via the overall 3D interconnecting system, which is composed by two main orthogonal and heterogeneous facilities:

• on-layer communication network (NoC), for horizontal communication on the bottom layer;
• fast vertical DRAM controller with TSV DRAM physical interface for vertical communication to upper layers.

The whole memory sub-system is accessible from the bottom layer by every tile through this homogeneous 3D interconnection. Every CPU can reach every SPM memory through the bottom-layer horizontal communication network, but also every sub-bank memory allocated in the upper layers via the on-layer communication network and the appropriate vertical memory controller interface. However, different CPU-to-memory paths have different communication latencies. Shorter

---

1Virtual memory may be supported by the host processor, which usually runs a complex OS and performs high-level orchestration [22]
paths provide faster communication, while multi-hop paths imply higher latency. Fig.1 shows two memory access examples: the black path denotes a fast communication since the CPU is accessing a memory region which is right above it, while the white path has a higher latency because the transaction has to travel across two links on the bottom layer before reaching the right TSV. Multi-hop transactions are affected by the delay of the links that they need to cross. Moreover, they cause congestion of the overall system interconnect.

IV. NEIGHBORHOOD PROGRAMMING

Data intensive embedded applications are designed to handle and perform on large data structures. In these programs a high degree of data parallelism is available, where the SPMD model is used to replicate the computational kernels in parallel threads working on different subsets of the target data array. In general terms, in the SPMD model each instance of a parallel task (i.e. each thread) accesses only a subset of the shared data. Part of the shared data-set may overlap among different threads, but typically only few border elements are actually accessed by multiple threads. The benefits in placing frequently accessed data close to the processor are well known, particularly when dealing with complex memory hierarchies with NUMA organization. From a practical point of view this has been historically dealt with by means of data transfer to constantly keep in small and fast on-chip SRAM memories such data, with the goal of satisfying most memory references from there. In 3D architectures, three-dimensional stacking of DRAM (main) memory greatly mitigates the memory size limitation and thus the need for frequent data movements, but it does not solve the problem of efficiently mapping data to physical memory banks. Naïve topology-agnostic data placement techniques which allocate entire arrays on a single memory neighborhood may create interconnect bottlenecks and suffer significant latency penalties. Stacking-aware allocation schemes are needed, where different parts of a shared data structure (hereafter called tiles) can be mapped to different physical memory stacks with the goal of minimizing accesses to non-neighbor stacks.

The process of partitioning an array introduces addressing difficulties. Since data tiles can become not-contiguous in physical memory, we can no longer reference the data structure as a whole by simple offset computation. Addressing an element involves finding its physical address, specified by a memory bank number and an offset within that bank. In our view of the neighborhood programming model, the programmer can express at the application level the necessity for distributed placement of a shared data structure. Partitioning is then triggered in the compiler, which transforms the program to synergistically interact with the runtime environment to find the most convenient placement.

A. Distributed Data Placement

Our 3D MPSoC features both private and shared regions of the address space. Our compiler efficiently deals with thread-private variables by allocating them onto the private block of their host processor’s neighborhoods. Dealing with shared data is trickier. The OpenMP memory model assumes a single memory space and provides no facilities to specify how data is to be arranged within the memory space. This calls for language features to specify data placement onto specific memory blocks. To this aim, we extended the OpenMP API with a custom distributed directive.

```c
int A[N][M];
#pragma omp distributed (Af, mem_id);
```

Declarations of distributed variables are changed by the compiler into pointers, which point to a region in the shared address space. The optional `mem_id` parameter specifies a particular memory neighborhood for placement.

B. Array Data Partitioning

OpenMP provides work-sharing directives to divide computation among parallel threads, but it lacks means to specify an affinity between the data set touched by a thread and its physical placement on a memory block. We provide API extensions to add this feature. The partitioning process can be triggered within a parallel region by annotating the shared array with the custom `split` clause. If the `split` clause is specified, the programmer assures that all references issued by a thread fall within a single tile.

```c
#pragma omp parallel for split (a)
for (i=0; i<N; i++)
a[i] = foo();
```

This functionality is similar to those found in High Performance Fortran (HPF) or OpenMP extensions[18][20] for data distribution. Anyhow, our partitioning technique is much more flexible, in that – unlike cited approaches – the granularity of array blocking can be arbitrarily small at a very contained cost (see Sec. IV-C2).

Similar parallelization schemes in which each processor touches distinct portions of an array are quite common in OpenMP programs. In this case data tiles are straightforwardly placed close to processors hosting the logically associated thread. In less regular programs often happens that threads need to reference non-local data. Typical implementations of data distribution techniques for NUMA machines rely on heavyweight virtual memory paging techniques to fetch remote data. Here the cost for virtualization is hidden behind the high latencies of the communication medium. On our platform this solution is unfeasible, since we are lacking both the hardware (i.e. MMUs) and software (a full-fledged operating system) support for virtual memory. Furthermore, the high cost for such a virtualization layer would no longer be paid off, due to the low cost for communication in our interconnection system. Thus, our implementation rather relies on a streamlined support for address translation, based on metadata for array indexing that is explicitly managed and allocated (see Sec. IV-C). At the application level, we provide the `tiled` clause, that can be coupled with a parallel directive to describe an irregular access pattern. In the following example each array element is placed on a distinct memory, and every thread accesses all of them. Thus, three out of four accesses are to remote memories.

```c
#pragma omp parallel tiled (a)
for (i=0; i<N; i++)
a[i] = foo();
```

Fig. 2 shows how the wanted array element is accessed through address translation. Every single reference to distributed array `a` is instrumented by our compiler with a call to the library function `GOMP_access_tiled_array()`. Based on the reference offset, the runtime computes a tile ID, then looks up in the `tiles` metadata array for the correct address.
C. Runtime Support to Data Partitioning and Placement

The partitioning technique described above relies on metadata (the tiles array) containing the base address of data tiles. Metadata is replicated onto each processor’s memory neighborhood for fast local inspection. By default, arrays are partitioned in a number of tiles that is equal to the number of worker threads (i.e., cores), and addresses for each tile are generated by the compiler according to a cyclic distribution onto memory neighborhoods. This default choice has three advantages: first, it captures the thread-to-memory affinity of static loop scheduling (the most common in OpenMP programs). Second, it enables the coarsest partitioning scheme, which generates metadata with very small memory footprint. Third, it requires almost no intervention from the programmer.

The default scheme provides good results for regular applications, that are amenable to static loop parallelization. For programs with strided or irregular memory patterns, cyclic placement and coarse-grained partitioning may lead to high rates of remote accesses. We show in the following subsections the solutions we provide to improve locality.

1) Automatic Generation of Affinity-based Data Layouts: Default cyclic tile placement can be overridden by providing a custom tile layout descriptor (metadata) in a specific header file. Devising an efficient placement requires insights on the application behavior on memory. To make this task easier we enriched our compilation toolchain with scripts that automatically find affinities between threads and data tiles, based on access count information gathered during a profile run of the application. Metadata representing a placement that minimizes the number of remote references in the program is automatically generated and included for compilation.

To conceptually show the benefits of affinity-based placement w.r.t. cyclic placement, let us consider the following example. A loop is statically parallelized among four threads, and default cyclic partitioning is enabled.

```
#pragma omp parallel for schedule(static) split(arr)
for (i=0; i<N; i++)
   arr[i] = foo();
```

We represent the footprint of threads on the array in Fig. 3. Threads are represented with dashed lines, and array tiles with thick black borders. Since the lower boundary of the iteration space is greater than zero, the portions of the array accessed by the four threads do not overlap completely with the array tiles. In Fig. 3.a) we show cyclic (default) tile placement. Color coding is used to associate a tile to a thread, so each tile is associated to a different thread in a cyclic fashion. We use plain and dashed filling to represent local and remote references, respectively. In Fig. 3.b) we show affinity-based tile placement. Here tiles are allocated onto the memory neighborhood local to the processor that mostly references it. It is possible to notice that affinity-based placement accounts for the irregularity in loop boundaries. Both tiles 1 and 2 are allocated local to thread 1, and the number of remote accesses is significantly reduced.

2) Refining Partitioning Granularity: Keeping the size of metadata small is profitable when memory space is strictly constrained. On the other hand, in irregular programs exploiting few large tiles may result in poor approximation of the thread footprint on memory, resulting in poor locality. On our 3D architecture there are no strict memory space constraints, so we can refine the granularity of partitioning. The programmer can specify the number of tiles for partitioning. The finer the partitioning is done, the more overlapping of data tiles with accessed locations is achieved. This is shown in the comparison between coarse-grained affinity-placement (Fig. 3.b) and fine-grained affinity-placement (Fig. 3.c) for the example loop introduced in the previous section.

In Fig. 3.c the array is partitioned in eight tiles, each of which is placed locally to the processor with higher affinity. This significantly reduces the number of remote accesses.

It is worth underlining here that data distribution on traditional NUMA machines either only provide page granularity for partitioning[21] (which is often too coarse to be beneficial), or resorts to very tricky and expensive techniques (e.g. data padding at the page level) to provide finer partitioning[19]. We can support fine-grained partitioning with much better efficiency, with arbitrarily small data tiles at the same cost for address translation and at the only increased cost for memory footprint of metadata.
V. Experimental Results

We describe in this section the experimental setup used to evaluate our programming framework and the results obtained.

The OpenMP-based programming framework with the proposed extensions was implemented within the GCC 4.3 compiler (GOMP). The runtime library (libgomp) has been rewritten from scratch with no OS support. The library code is executed by every core. At system startup the processor with the highest ID is designated as the master processor, and it is responsible for orchestrating parallel execution by synchronizing slave processors and pointing them to parallel code and shared data.

We implemented an instance of the 3D platform template presented in Sec. III within a SystemC full system simulator. We simulate a 3D chip composed by three layers. The bottom level hosts 16 processor tiles, while memory stacks (16 MB each) reside on the topmost two layers. Each processor tile features a RISC-like CPU coupled with 16KB scratchpad memory (SPM) and a small unified cache (16KB) for private data and instructions. Caches only manage private data, therefore any coherence issue is prevented. The network on chip on the CMP die is based on the ST STBus protocol.

In the image on the left of Fig. 4 we show the layout of processing tiles on the CMP die. Processor IDs increase with the pattern indicated by the arrow. The master core is kept in a central position in the CMP die to balance communication costs among cores towards its memory stack. The memory access time is not constant for the entire hierarchy, but depends on the transaction path. Accesses to local SPM are subjected to only 1 cycle latency. For remote SPMs this cost depends on the internal memory interface latency (≈ 2 cycles), the number of hops to the target memory controller, the contention level on the network, the neighborhood interface latency (≈ 2 cycles), the neighborhood memory latency (1 cycle for SPM, ≈ 5 cycles for 3D stacked DRAM). The zero-load latencies for each core to traverse the interconnect for remote memory access are modeled as depicted in the image on the right. For instance, in absence of contention accessing data on the memory neighborhoods of processors 4, 14 or 10 from processor 12 is subject to a latency of 20 cycles. If interconnect resources are shared with other concurrent transactions, the latency will be higher.

We show results obtained with two benchmark applications. The first is a normalized cut clustering (NCC) image processing kernel, and the second is a JPEG decoding algorithm. Performance plots with a breakdown of parallel execution time for each processor (on the X-axis) highlight:

1) Mem port congestion - Idle time due to congestion on the memory port (serialization of transactions)
2) Latency - Time spent in delivering/retrieving data through the network (zero-load latencies)
3) CPU time - Time spent on computation

Such plots are drawn for three program configurations:
Baseline: All shared data is placed in the memory neighborhood of the master processor.
Basic Tiling: Coarse-grained partitioning with affinity-based placement.

To further evaluate the effectiveness of the granularity optimization we also provide plots that show the percentage of local memory accesses for decreasing tile sizes.

A. NCC benchmark

The main program loop is parallelized with static scheduling, but the overall number of iterations does not evenly divide the number of processors. An equal number of iterations is assigned to all processors but the one with the highest ID, which has a lighter workload. This behavior justifies the shorter execution time for processor 15 in the baseline plot (Fig. 5.a.1). When all shared data resides in the master core's memory stack – as expected – severe penalties due to memory port congestion are encountered. Latencies to access remote memory neighborhoods also lengthen significantly execution time. The congestion problem is completely removed when applying partitioning, even with coarsest granularity (Fig. 5.a.2). This yields a 6× speedup w.r.t. the baseline technique (the plots have different scales). For processors 0, 6 and 12, all memory references are satisfied from the local neighborhood, whereas other processors suffer varying degrees of penalty for accessing remote neighborhoods. Fine-grained partitioning (Fig. 5.a.3) with eight times smaller tiles allows a further significant reduction of the time spent on the interconnect (an additional 15% speedup). It has to be pointed out that the considered interconnect architecture has very low zero-load latencies, thus limiting the benefits of fine-grained partitioning. We expect it to be even more profitable when considering NoCs with higher zero-load latencies. The cost for fine partitioning is the increased footprint of metadata in memory. Coarse partitioning employs as many tiles as processors, which requires 64 bytes-metadata. The finest partitioning considered in these experiments generates metadata which has a footprint of 512 bytes (2.5% of the decoded image size). In Fig. 5.a.4 we plot the percentage of accesses satisfied from local neighborhood. On the X-axis the granularity of partitioning (1 corresponds to the basic technique. 1/2 means tile sizes halved and so on). Locality is improved by 11.43% for array 1 and by 23.60% for array 2 when 1/8 sized tiles are considered.

B. JPEG decoding benchmark

This benchmark is parallelized with dynamic scheduling. Chunks of iterations are distributed in a FCFS fashion to worker threads. We choose the chunk size to be a fraction (up to 1/8) of the tile size for the coarsest partitioning. This can be considered as a worst-case for the coarse-grained tiling technique. Indeed, even if affinity-based placement allocate tiles close to the processor with highest rate of accesses, still multiple threads insist on the same tile. When the entire image lays un-partitioned in the master processor’s memory neighborhood (Fig. 5.b.1) we experience the usual contention penalties. Due to the high contention on its local neighborhood, the master core is delayed in executing its work. This
results in a fewer number of invocations to the runtime library for work assignment, which justifies the shorter time spent on CPU computation. As expected, coarse-grained tiling (Fig. 5.b.2) suffers a high number of remote references, which also implies some interconnect congestion. This notwithstanding, a 2.7× speedup is achieved w.r.t. the baseline. Refined the granularity at 1/8 tile size leads to 15% reduction of the time spent on memory subsystem (Fig. 5.b.3), with a metadata footprint on memory which amounts to 1.3% of the image size. Fig. 5.b.4 shows that fine-grained partitioning allows almost perfect overlapping of thread and data space, thus leading to excellent locality.

VI. CONCLUSION

In this work we moved a first step toward the definition of 3D-aware programming abstractions and tools to enable effective exploitation of the large potential for increased computational efficiency offered by 3D-integrated memory architectures. We outlined the concept of memory 3D neighborhood programming, and we developed language extensions, compiler enhancements and run-time support for neighborhood programming within the standard OpenMP shared memory programming environment. Our approach is specifically focused to explicitly-managed memory architectures and applications with SPMD parallelism. Results are extremely promising, but much work remains to be done both in advanced optimization techniques and in extending the scope of applicability of the memory neighborhood concept to other classes of architectures (e.g. cache-coherent SMP) and application classes with different forms of parallelism.

ACKNOWLEDGEMENTS

This work was supported by projects SHARE (FP7-ICT-224170) and PREDATOR (FP7-ICT-216008), funded by the European Community.

REFERENCES