Optimal 2D Data Partitioning for DMA Transfers on MPSoCs

Selma Saidi*†, Pranav Tendulkar*, Thierry Lepley†, Oded Maler*
*VERIMAG Lab (CNRS,University of Grenoble), France
†STMicroelectronics Grenoble, France

***Abstract*—Reducing the effects of off-chip memory access latency is a key factor in exploiting efficiently embedded multi-core platforms. We consider architectures that admit a multi-core *computation fabric*, having its own fast and small memory to which the data blocks to be processed are fetched from external memory using a DMA (direct memory access) engine, employing a double- or multiple-buffering scheme to avoid processor idling. In this paper we focus on application programs that process two-dimensional data arrays and we determine automatically the size and shape of the portions of the data array which are subject to a single DMA call, based on hardware and applications parameters. When the computation on different array elements are completely independent, the asymmetry of memory structure leads always to prefer one-dimensional horizontal pieces of memory, while when the computation of a data element shares some data with its neighbors, there is a pressure for more "square" shapes to reduce the amount of redundant data transfers. We provide an analytic model for this optimization problem and validate our results by running a mean filter application on the CELL simulator.**

***Index Terms*—Data parallelization, Direct Memory Access (DMA), CELL Processor, Double Buffering.**

I. INTRODUCTION

Multiprocessor system on chip (MPSoC) such as the CELL processor [5] or the more recent Platform2012 [1] are heterogeneous multi-core architectures, with a powerful host processor and a computation fabric consisting of several smaller cores whose intended role, somewhat similar to graphics processing units (GPUs) [12], is to replace dedicated hardware accelerator. Computation-intensive (and parallelizable) parts of the application are offloaded to the multi-cores for execution. Theses parts of the application are often data intensive, operating on large arrays of data initially stored in a remote off-chip memory whose access time is about 100 times slower than that of the cores local memory. A major characteristic of the CELL and P2012 is a *software controlled* local storage rather than a *hidden* cache mechanism. Indeed, the local scratch-pad memory is the only memory directly available to each core, and to access data in the off-chip memory a processor must issue explicit Direct Memory Access (DMA) requests. This simplifies hardware design but at the cost of a greater program complexity for managing explicit memory transfers through the memory hierarchy. To handle these issues new programming models have been introduced such as *Sequoia* and *Cellgen* [4], [16], [17]. DMA can transfer large amounts of data between memory locations without processor intervention hence it offers another level of parallelism by overlapping computations and data prefetching [5], [15]. However, to exploit efficiently these new capabilities, the programmer has to make decisions about the granularity of data transfers and the way they are scheduled to achieve optimal performance. This paper is part of an effort to provide tools that help developers in making such decisions, and ultimately, to automate the whole process of data parallelization for such platforms.

We focus on data parallel applications that exhibit a regular computation and data transfer pattern for which explicit control of data transfers can be more efficient than implicit unpredictable low granularity caching mechanisms [14]. In [13] optimal data transfer granularity for programs that work on *one-dimensional* arrays of data has been derived. In this paper we tackle the more challenging case of two-dimensional arrays which is common in applications such as video/image processing and certain types of scientific computations. As we explain below, when the computations on array elements are mutually *independent*, the structure of memories dictates a solution similar to the one-dimensional case. Hence we move on to the situation where the computation for one data block depends also on data from its *neighbors*. We assume that this shared data is replicated and sent to all processors that process array elements that need it. We compute optimal size and shape for these data transfers.

While a lot of work has been done in the past to successfully parallelize such applications such as [2], [3], [9], contemporary heterogeneous architectures with a limited on-chip memory and a high latency main memory change the formulation and parameters of the problem and call for new solutions in order to achieve high performance on these platforms.

The rest of the paper is organized as follows. In Section II we define the applications, the double buffering technique, the DMA cost model and the problem of optimal granularity. In Section III we solve the problem for completely independent computations on a single processor. The solution is extended to multi-processors and shared data in Section IV and then validated experimentally on a simulator of the CELL architecture in Section V. A discussion of this work and its continuations closes the paper.

II. PRELIMINARIES

A. Software Pipelining

Consider the sequential algorithm (Program 1) for computing $Y = f(X)$ which *uniformly* applies f to a two-

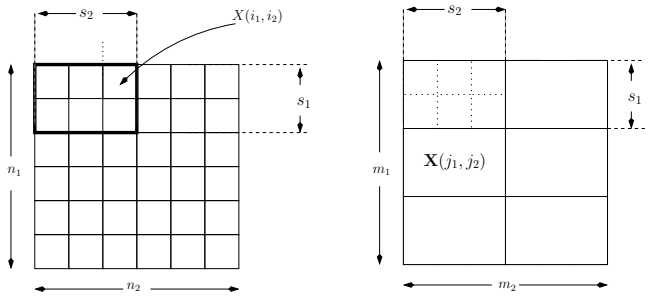


Fig. 1. Basic blocks $X(i_1, i_2)$ and super blocks $\mathbf{X}(j_1, j_2)$ (logical view).

dimensional input array X of $n_1 \times n_2$ elements to produce an output array Y of the same dimension. Array X is stored initially in the off-chip memory and Y is to be written on this memory after computation.

Program 1 (Sequential):
for $i_1 := 1$ **to** n_1 **do**
  for $i_2 := 1$ **to** n_2 **do**
    $Y(i_1, i_2) := f(X(i_1, i_2))$
  od
od

We assume that an array element represents the *minimal granularity* for which the computation of f can be carried out. In image processing it can be a pixel, a block or a macroblock. We refer to such granularity as a *basic block*. For the moment we talk only on the logical structure of the array and defer the discussion of the physical memory layout to the sequel. One could handle data transfers at the basic block level by putting explicit DMA calls *get* and *put* before and after the computation in the main program loop. However, as is common with other slow memories such as disks, these transfers are applied on a coarser granularity by clustering together several array elements which are brought into a buffer via a *single* DMA call. We call such clusters *super blocks* and assume they are rectangular blocks consisting each of $s_1 \times s_2$ basic blocks, see Figure 1. One can view the super blocks as arranged in an $m_1 \times m_2$ array $\mathbf{X}$ (and $\mathbf{Y}$) with $m_1 = n_1/s_1$ and $m_2 = n_2/s_2$. We use

$$\mathbf{X}(j_1, j_2) = \left\{ X(i_1, i_2) : \begin{array}{l} (j_1 - 1)s_1 + 1 \le i_1 \le j_1 s_1 \\ (j_2 - 1)s_2 + 1 \le i_2 \le j_2 s_2 \end{array} \right\}$$

to denote the set of basic blocks associated with a super block indexed by (j_1, j_2). It is sometimes more convenient to view two-dimensional arrays as one-dimensional and this is done by a flattening function $\phi : [1..m_1] \times [1..m_2] \to [1..m]$, for $m = m_1 m_2$. We will sometime refer to super block $\mathbf{X}(j_1, j_2)$ as $\mathbf{X}(j)$ for $j = \phi(j_1, j_2)$. We use buffers B_x and B_y for input and output super blocks and rewrite Program 1 as follows.

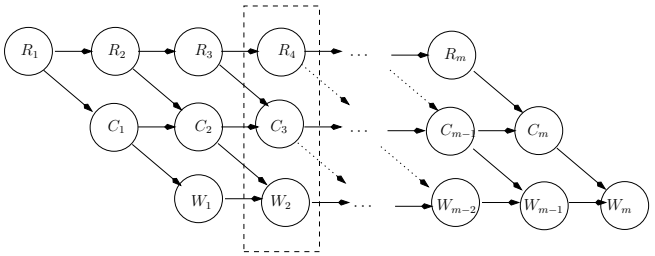


Fig. 2. A schematic description of a pipeline: Read, Compute, Write.

Program 2 (Buffering):
for $j := 1$ **to** m **do**
  $get(B_x, \mathbf{X}(j))$; read super block
  for $i_1 := 1$ **to** s_1 **do**
    for $i_2 := 1$ **to** s_2 **do** compute for all blocks
      $B_y(i_1, i_2) := f(B_x(i_1, i_2))$ in super block j
    od
  od
  $put(B_y, \mathbf{Y}(j))$; write super block
od

In the following, we use $comp(j)$ as a shorthand for the inner double loop.

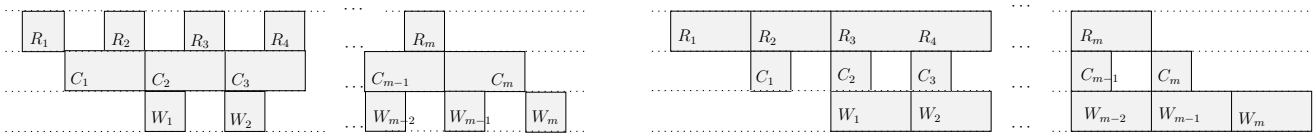
In program 2, data transfers and computations are performed *sequentially* and the processor is idle during reading and writing. Using *double buffering* for input and output blocks $B_x[0]$, $B_x[1]$, $B_y[0]$ and $B_y[1]$, the processor can work on a super-block j residing in one buffer while the DMA brings *in parallel* the super-block $j + 1$ to the other buffer. Program 3 defines a *software pipeline* with 3 stages: input of super block $(j+1)$, computation on super block j and output of super block $(j-1)$, see Figure 2. Reading the first block and writing back the last block are, respectively, the *prologue* and *epilogue* of the pipeline.

Program 3 (Double Buffering):
$c := 0$; $c' := 1$;
$get(B_x(0), \mathbf{X}(1))$; first read
$get(B_x(1), \mathbf{X}(2)) \parallel comp(1)$;
for $j := 2$ **to** $m - 1$ **step** 1 **do**
  $get(B_x(c), \mathbf{X}(j+1)) \parallel comp(j) \parallel put(B_y(c'), \mathbf{Y}(j-1))$;
  $c := c \oplus 1$; $c' := c' \oplus 1$;
od
$comp(m) \parallel put(B_y(0), \mathbf{Y}(m-1))$;
$put(B_y(1), \mathbf{Y}(m))$; last write

To reason about the optimal granularity of data transfers we need first to analyze the performance of the pipeline and to this end we need to refine the qualitative description of Figure 2 which describes the obvious precedences between the computations and data transfers but tells us nothing about their relative durations.

Without loss of generality we assume that a basic (input and output) block consists of a contiguous chunk of b bytes[1], that

[1]The model can be easily adapted to the case where it is a $b_1 \times b_2$ rectangle.



(a) Computation regime

(b) Transfer regime

Fig. 3. Pipelined execution using double buffering for one processor in both regimes

the array is organized contiguously in a lexicographic order and that transfer costs in both directions are identical. We denote the transfer time and computation time per super block by T and C, respectively. Based on the balance between T and C, the behavior of the software pipeline splits into two cases: the *computation regime*, where $C > T$ and the *transfer regime* when $C < T$, illustrated in Figures 3-(a) and (b), respectively. It is not hard to see that the total execution time is:

$$\begin{array}{ll} m \cdot C + 2T & \text{when } C > T \\ (m+1) \cdot T & \text{when } C < T \end{array} \tag{1}$$

The relation between the computation time of a super block and its transfer time is not fixed but can be controlled to some extent by varying the *size* and *shape* of the super block. To this end we need to characterize the DMA behavior. In the following we detail the DMA command flow in the CELL architecture on which we base our experiments and derive a model of its performance.

B. DMA Performance Model

We start with features that are common to most DMA mechanisms. To copy data from one memory location to another, a processor issues a command to the DMA which takes charge of the transfer. Such a command typically consists of a *source* address, a *destination* address and a block *size*. The command execution decomposes into two major phases:

1) *Command initialization* phase: including the command issue time, the time to write the command in the queue and potentially some address translations. Note that this phase is *independent* of the amount of data to transfer.
2) *Data transfer* phase: when a command is ready, data transfer begins and the block is divided into smaller packets that travel from source to destination through an interconnect (bus, NoC). The duration of this phase is *proportional* to the amount of data.

The initialization cost is typically significant and is amortized when the blocks are large. This makes the DMA more attractive for coarse data granularity than for load/store instructions. For one-dimensional data arrays the super blocks are stored as *contiguous* memory segments but this is no more the case for two-dimensional data arrays that require usually rectangular data block transfers. Bringing non-contiguous (but regularly structured) pieces of data is possible using *strided* DMA commands that in addition to the source and destination address specify the stride which is an offset to access the next contiguous block in memory. Strided transfers are more expensive than contiguous transfers of the same size.

In the CELL architecture, a strided DMA command is implemented using a *DMA list*, that is, *one* DMA command composed of a list of contiguous transfers. It can be viewed as an array whose entries are pairs consisting of a main memory address and a contiguous transfer size. Each list element can refer to a different location in the external memory. However, all elements of the list move data in the *same* direction (*get* or *put*) and the transferred data should form a contiguous block in local memory.

The cost of transferring a super block of $s_1 \times s_2$ basic blocks (physically a rectangular chunk of memory of s_1 lines and $b \cdot s_2$ columns) can be approximated by the affine function

$$T(s_1, s_2) = I_0 + I_1 \cdot s_1 + \alpha \ (b \cdot s_1 \cdot s_2) \tag{2}$$

This function assumes a fixed initialization cost I_0, independent of the size and form of the super block, an additional initialization cost I_1 associated with each element (contiguous line) in the DMA list, and the transfer itself which is proportional to the size (area) of the super block. Note that in this model we assume a *fixed* transfer latency α. This assumption is imprecise for two major reasons:

- We do not model the characteristics of the external DRAM memory such as the scheduling policy of the memory controller, the effect of page misses and data refreshment latencies.
- The speed of transfer in the interconnect, especially in a multi-processor setting, depends crucially on the number of simultaneous transfer requests from the processors.

The first issue is too complex to handle precisely as memory controllers vary among vendors. We can assume, however, that page misses are distributed more or less evenly and their effect does not favor or disfavor a specific choice of granularity. Moreover, the preference to contiguous blocks captured by our cost model holds also on the memory controller side. As for the influence of demand patterns on the latency of the interconnect, we will use later a model where α is parameterized by the number p of active processors with $\alpha_p < \alpha_{p'}$ whenever $p < p'$.

We assume the algorithm for computing f to have a fixed (data independent) computation time ω per basic block, once the block is in local memory. This is the time to perform one iteration in Program 1. The cost of computing a super block is therefore

$$C(s_1, s_2) = \omega s_1 s_2 \tag{3}$$

In the next section, based on (2) and (3), we derive optimal granularity for super blocks in Program 3.

III. Independent Computations, Single Processor

As mentioned previously we can control to some extent the relation between T and C by controlling the size and shape of super blocks, but which relation is preferred? The answer depends on which resource is more stressed by the application, computation or communication, a fact characterized by

$$\psi = \omega - \alpha b.$$

Condition $\psi < 0$ means that regardless of the choice of data granularity, transfer time always dominates computation time. In this case we prefer large data blocks (which corresponds to the maximal buffer size allowed by the local store capacity) to amortize the DMA initialization time and fully utilize the interconnect bandwidth. In the sequel, we focus on the other case where $\psi \geq 0$, that is, more computation than transfer. In this case we opt for a shape that yields a computation regime.

Assuming $\psi > I_1$, the dependence of T and C on their arguments is illustrated in Fig. 4. The intersection of these two surfaces separates the domain of (s_1, s_2) into two sub-domains, the *computation domain* $T \leq C$ where the computation of a super block dominates the transfer of the next one and the *transfer domain* where $T \leq C$, see Fig. 5-(a).

Computation regime yields a better performance because the processor does not stall, waiting for data, between two iterations. Therefore we orient the super block selection towards (s_2, s_2) such that $C(s_1, s_2) \geq T(s_1, s_2)$ and $m \cdot C(s_1, s_2) + 2T(s_1, s_2)$ is minimal. Since the processor is always busy, all shapes satisfying $T < C$ admit roughly the same total computation time $m \cdot C(s_1, s_2) \backsimeq \omega n_1 n_2$. Hence, it remains to optimize the length of the prologue and epilogue $2T(s_1, s_2)$ with computation regime viewed as a constraint. Obvious additional constraints state that a super block is somewhere between a basic block and the full image, provided its size does not exceed the maximum local buffer size M imposed by the local store limited capacity. This leads to the following constrained optimization problem.

$$\begin{aligned} &\min\ T(s_1, s_2) \quad \text{s.t.} \\ &\\ &T(s_1, s_2) \leq C(s_1, s_2) \\ &(s_1, s_2) \in [1, n_1] \times [1, n_2] \\ &b \cdot s_1 \cdot s_2 \leq M \end{aligned} \tag{4}$$

Since the computation domain is convex, the candidates for optimality are restricted[2] to the the *intersection* $T = C$. These points are of the form $(s_1, H(s_1))$ where

$$H(s_1) = (1/\psi)(I_1 + I_0/s_1)$$

and their transfer time is expressed as a function of the number of clustered horizontal blocks s_1:

$$T(s_1, H(s_1)) = c\ (I_0 + I_1 s_1)$$

[2]Because for any point s inside the domain, we can always find another point s' ($s'_2 = s_2$ and $s'_1 < s_1$) on the boundary with a smaller transfer time.

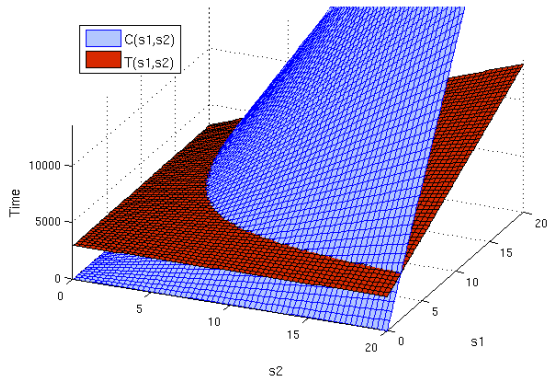


Fig. 4. The dependence of computation C and transfer T on granularity (s_1, s_2).

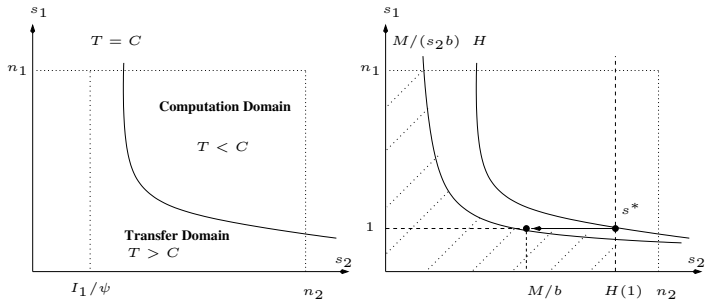


Fig. 5. (a) Computation and transfer domains, (b) Optimal granularity and local memory constraint.

where c is the constant $1+(\alpha b/\psi)$. Since T is linear and monotone in s_1, the optimal shape is $s^* = (s_1^*, H(s_1^*)) = (1, H(1))$, a contiguous block of one line of the physical data array. This is not surprising as the asymmetry between dimensions in memory access prefers "flat" super blocks with $s_1 = 1$. Without data sharing and memory size constraints the problem becomes similar to the one-dimensional case [13] where it is only the *size* of the super block that needs to be optimized. The memory size constraint $s_2 s_2 b \leq M$ excludes solutions which are above the hyperbola shown in Fig. 5-(b). If the optimal shape $(1, s_2^*)$ does not satisfy this constraint it is replaced by $(1, M/b)$. Note that in addition to these constraints, each specific DMA engine imposes additional constraints on the range possible values of s_1 and s_2.

IV. Shared Data

So far we considered data independent applications. In the rest of this paper we focus on applications where the computation for each block involves additional *input* data from neighboring blocks. In other words, the computation in the inner loop of Program 1 is replaced by

$$Y(i_1, i_2) = f(V(i_1, i_2))$$

where $V(i_1, i_2)$ denotes a set of basic blocks consisting of $X(i_1, i_2)$ and it neighbors. Without loss of generality, we

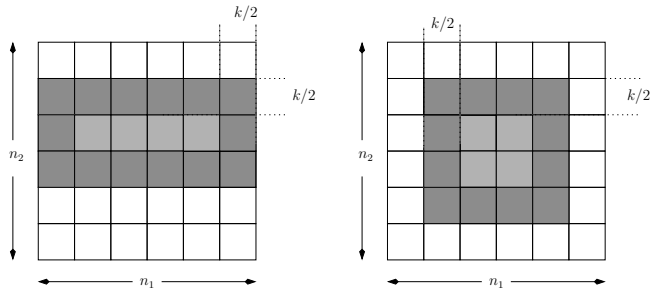


Fig. 6. A flat and a square super blocks of the same area. The shaded area represents replicated data overhead for $k = 2$.

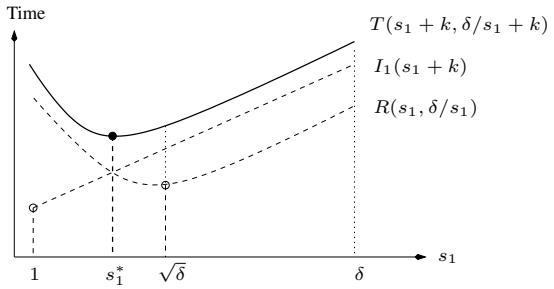


Fig. 7. DMA transfer cost with replicated area as we increase number of lines in a block for a given $\delta = s_1 \times s_2$.

assume $V(i_1, i_2)$ to be a square around $X(i_1, i_2)$, that is,

$$V(i_1, i_2) = \left\{ X(j_1, j_2) : \begin{array}{l} (i_1 - k/2 \le j_1 \le i_1 + k/2 \\ (i_2 - k/2 \le j_2 \le i_2 + k/2 \end{array} \right\}$$

We assume that shared data is *replicated* at each transfer to the local memory. In the sequel we explain how the shape of the block and its replicated area influence the transfer cost and then derive optimal granularity for shared data considering first one processor and then multiple processors.

A. Replicated Area and Transfer Cost

To process a super block of shape $s_1 \times s_2$, one needs to load

$$R(s_1, s_2) = (s_1 + k) \times (s_2 + k)$$

basic blocks. In other words, the overhead of replicated external data[3] is $k(s_1 + s_2) + k^2$ which, among all the super blocks of the same area, is minimal for *square* super blocks as illustrated in Figure 6. This fact is in conflict with the DMA issue overhead, optimized for flat blocks and there is a balance to be found between the two.

The DMA transfer cost under sharing becomes

$$T(s_1 + k, s_2 + k) = I_0 + I_1(s_1 + k) + \alpha b \cdot R(s_1, s_2) \quad (5)$$

Figure 7 illustrates this function for a fixed value of $\delta = s_1 \times s_2$ along with the DMA issue time overhead optimized for flat block transfer ($s_1 = 1$) and the replicated data transfer overhead optimized for square shapes $(\sqrt{\delta}, \sqrt{\delta})$. Among all combinations (s_1, s_2) satisfying $s_1 \times s_2 = \delta$, the transfer cost is minimal for the point $(s_1^*, \delta/s_1^*)$ where $s_1^* = \sqrt{\alpha bk\delta/(I_1 + \alpha bk)}$. This point represents *the* balance between initialization phase overhead (number of lines) and transfer phase cost (amount of replicated data). In the following section we derive optimal granularity for shared data applications taking this fact into account.

B. Optimal Granularity for Shared Data Applications

1) Single Processor: With data replication, the constrained optimization problem 4 becomes

$$\begin{array}{l} \min\ T(s_1 + k, s_2 + k) \ \text{ s.t.} \\ \\ T(s_1 + k, s_2 + k) \le C(s_1, s_2) \\ (s_1, s_2) \in [1, n_1] \times [1, n_2] \\ b \cdot (s_1 + k) \cdot (s_2 + k) \le M \end{array} \quad (6)$$

As for independent computations, candidates for optimal granularity are restricted to the points $(s_1, H(s_1))$ satisfying the equality $T = C$ and the problem is reduced to minimizing $T(s_1 + k, H(s_1))$ where

$$H(s_1) = (c_2 s_1 - c_3)/(\psi s_1 - c_1)$$

c_1, c_2 and c_3 are positive integer constants that depend on I_0, I_1, α, b and k such that,

$$\left\{ \begin{array}{l} c_1 = \alpha bk \\ c_2 = c_1 + I_1 \\ c_3 = I_0 + I_1 k + \alpha bk^2 \end{array} \right.$$

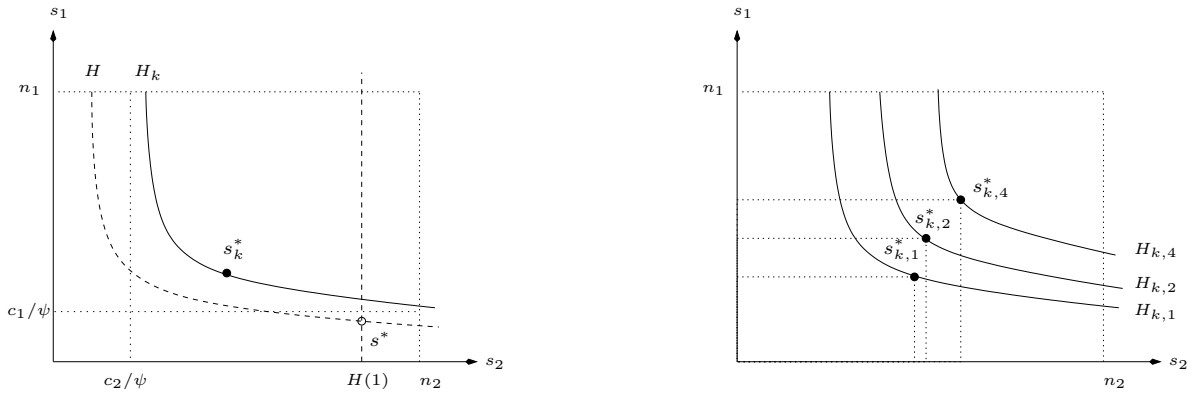
$T(s_1, H(s_1))$ is a second order function with one variable. By computing the derivative, we get one negative point that is not interesting for us and another positive point that is the optimal. To simplify the reading of the formulas, let $\Delta = (c_1/\psi)[1+D]$ where $D = \sqrt{c_3\alpha/c_1c_2}$, then

$$\left\{ \begin{array}{l} s_1^* = \Delta + (c_1/\psi)(1/D) \\ s_2^* = \Delta + (I_1/\psi)(1 + D) \end{array} \right.$$

Fig. 8-(a) illustrates the evolution of the computation domain and the optimal granularity while considering shared data. As discussed in the previous section, we can clearly see that optimal granularity is somewhere between a flat and a square block as s_1^* and s_2^* are both equal to Δ plus a different offset each.

2) Multiple Processors: Given p *identical* processors having the same processing speed and the same local store capacity, we assume for a given granularity that data blocks are allocated to the processors in a *cyclic* way. Since shared data are replicated among processors, the computations are completely independent. On the other hand, when the shared interconnect receives simultaneous transfer requests from several processors (*contentions*), it solves them on a packet by packet basis, serving the processors in a round robin fashion. Increase in number of processors does not influence the initialization

[3]The dark perimeter around the super blocks in Figure 6.



(a) One processor (b) Multiple processors

Fig. 8. Computation domain and optimal granularity for shared data.

phase as we assume a *distributed* DMA system, where each processor has its own DMA engine. However, contentions on shared resources induce a significant overhead that we model by parameterizing the transfer cost per byte α with the number of active processors such that α_p increases monotonically with p. We use T_p to denote (5) with α_p replacing α. [4] The total execution time of the pipeline becomes,

$$\begin{array}{ll} m/p \cdot C + 2T_p & \text{when } C \geq T_p \\ (m/p+1) \cdot T_p & \text{when } C < T_p \end{array}$$

The reasoning is similar to previous sections where function H becomes $H_p(s_1)$ yielding an optimal shape for each p.

Figure 8-(b) illustrates the evolution of computation domain and optimal granularity as we increase number of processors. Note that the difference between computation and transfer time represented by $\omega/p - b\alpha_p$ decreases as we increase p, reducing the computation domain. Note that optimal super block size increases with the number of processors because more data needs to be brought to each processor to keep it busy during the time it takes to fetch its next super block as well as the next super block of each of the other processors.

Table I summarizes the notations for the considered hardware and software parameters.

n_1, n_2	Array height and width in number of basic blocks
s_1, s_2	super block height and width in number of basic blocks
b	width of a basic block
k	shared neighboring data
ω	computation time per basic block
p	number of active processors
I_0	DMA initialization cost
I_1	DMA initialization overhead to issue a DMA list element
α	transfer cost in time per byte
α_p	transfer cost in time per byte of p concurrent requests
$T(s_1, s_2)$	transfer time of a super block
$C(s_1, s_2)$	computation time of a super block
M	Max local buffer size imposed by the local store capacity

TABLE I
PARAMETERS NOTATION

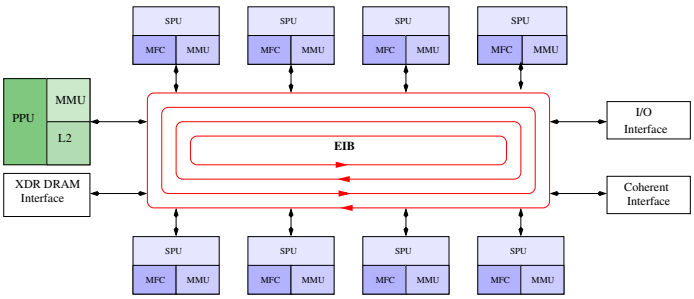


Fig. 9. The CELL Processor Architecture

V. EXPERIMENTS

We validate our results on the the CELL processor whose architecture is shown in Figure 9. It is by now a decade old architecture, still favorable for streaming applications. The main features of the architecture include a powerful general purpose processor (PPU-Power Processing Unit) along with 8 accelerators (SPU - Synergistic Processing Unit). Each SPU has a local scratchpad memory which is the only memory directly accessible using load/store instructions. Data in main memory and other processors memory is accessed using DMA. For more information about the architecture we refer to [6]–[8], [10].

As an application we use a *mean filter* algorithm that works on a bitmap image of 512×512 pixels. Each pixel is characterized by its intensity ranging over $0..255$. The output for a pixel is the average of the value of its neighborhood defined as a square (mask) centered around it. We have experimented with different mask sizes and focus on the presentation of the results for a 9×9 mask, that is, $k = 8$. In order to use SIMD operations to optimize the implementation of the code, we encode a pixel as an integer ($b = 4$ bytes). Based on profiling information, we are able to derive the DMA parameter values: fixed initialization cost $I_0 = 108$ and initialization cost per line $I_1 = 50$ cycles. The transfer cost per byte for p processors varies due to packet-level arbitration

[4] Architectures that have a centralized DMA can avoid this overhead by literally scheduling the transfer at the super block level.

	α_p		
p	min	max	avg
1	1.13	14.00	2.57
2	1.78	29.98	4.13
4	3.97	47.23	11.07
8	5.43	87.86	18.82

TABLE II
THE TRANSFER TIME PER BYTE AS A FUNCTION OF THE NUMBER OF PROCESSES.

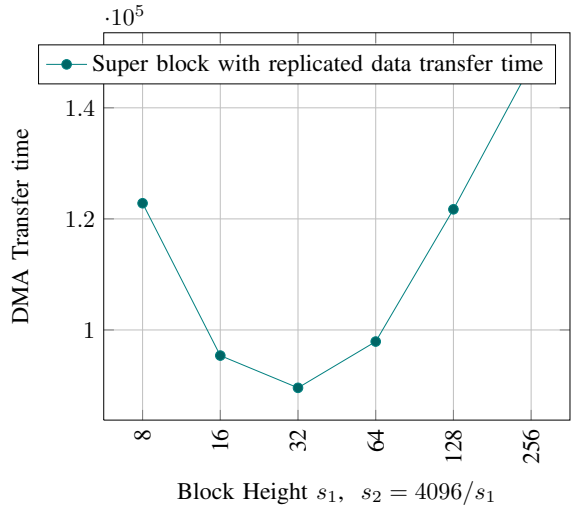


Fig. 10. Influence of block shape and its replicated data on the transfer time.

between request of different processors as well is reading and writing of the same processor. The minimal, maximal and average values of α_p are shown in Tab. II and we use the average value in our model. The computation workload per basic block is roughly $\omega = 62$ cycles (see remarks at the end of the section).

Note that due to the characteristics of the CELL not all combinations are possible. Indeed a DMA list can hold up to 2K transfer elements. Each element is a contiguous block transfer with maximum size 16KBytes (which corresponds in our case to $s_2 = 4096$). Furthermore, the CELL has a strict alignment requirements on 16-byte boundary for both DMA transfers and SPU vector instructions for which the processor is optimized. If this is not taken care of, the DMA engine aligns the data by itself causing erroneous results.

Figure 10 illustrates the influence of the shape of the block (and its implied replicated area) on the transfer time as explained in section IV. We consider in this plot different feasible combinations of (s_1, s_2) so that $s_1 \times s_2 = 4096$. A shape (s_1, s_2) yields a block of $s_1 + 8$ lines, each line corresponding to a contiguous transfer of $b \cdot (s_1 + 8)$ bytes. As argued in section IV, the optimal transfer time is obtained neither for square $(64, 64)$ nor the flattest possible $(8, 512)$ super blocks and the best trade-off in this case is $(s_1, s_2) = (32, 128)$.

Finally we evaluate the effect of the size and shape of the super blocks and the total execution time of the pipeline for different numbers of processors. Fig. 11-(a) compares

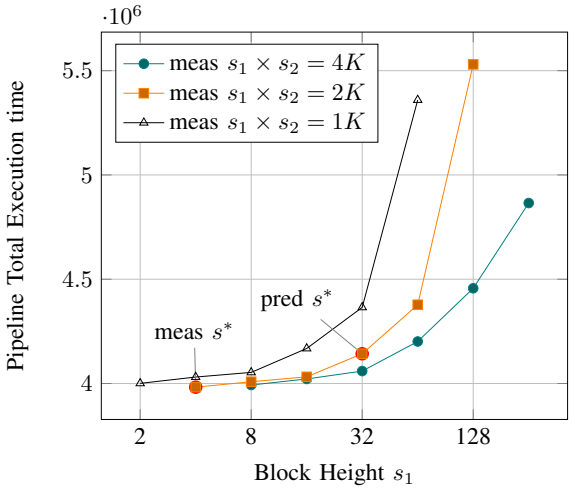


Fig. 12. Observed optimal granularity $s^* = (4, 256)$ and predicted optimal granularity $s^* = (64, 32)$.

the predicted and measured performance for different block shapes where $s_1 s_2 = 1024$ while Fig. 11-(b) does the same for $s_1 s_2 = 2048$. As one can see, the distance between the predicted and measured values is rather small except for large values of s_1.

The major reason for the discrepancy between the model and the reality is that $C(s_1, s_2)$ has non negligible component that depends on s_1 for two reasons. The first is due to the overhead at each computation iteration related to the setting required between the outer loop and the inner loop like adjustment of the pointers for every row, pre-calculation of sums of borders etc. Secondly, the creation of list elements occupies the processor and this overhead is also added to the overall execution time.

Fig. 12 combines the measured results for different super block sizes. The measured optimum is obtained for $(4, 256)$ while our calculation yield $(56, 33)$ whose nearest feasible value is $(64, 32)$ whose measured overall performance is less than 10% above the performance for the optimum. The discrepancy can be attributed to the reasons stated above, namely the dependence of C on s_1.

VI. DISCUSSION

Adapting array-processing algorithms to multi-core architectures is an activity that will occupy a lot of programmers time in the coming future and it is highly desirable to make it as transparent as possible. Efficient use of the memory hierarchy is crucial for performance on this new class of execution platforms. In this work we have demonstrated how the problem can be approached in a systematic manner for the CELL architecture. Starting from an abstract logical description of the application and the DMA specifications, we could build a model that captures the influence of the size and shape of buffered super blocks on performance. In particular, our model captures the tension between preference of flat super blocks (due to asymmetrical transfer cost) and square super blocks (due to the nature of the application).

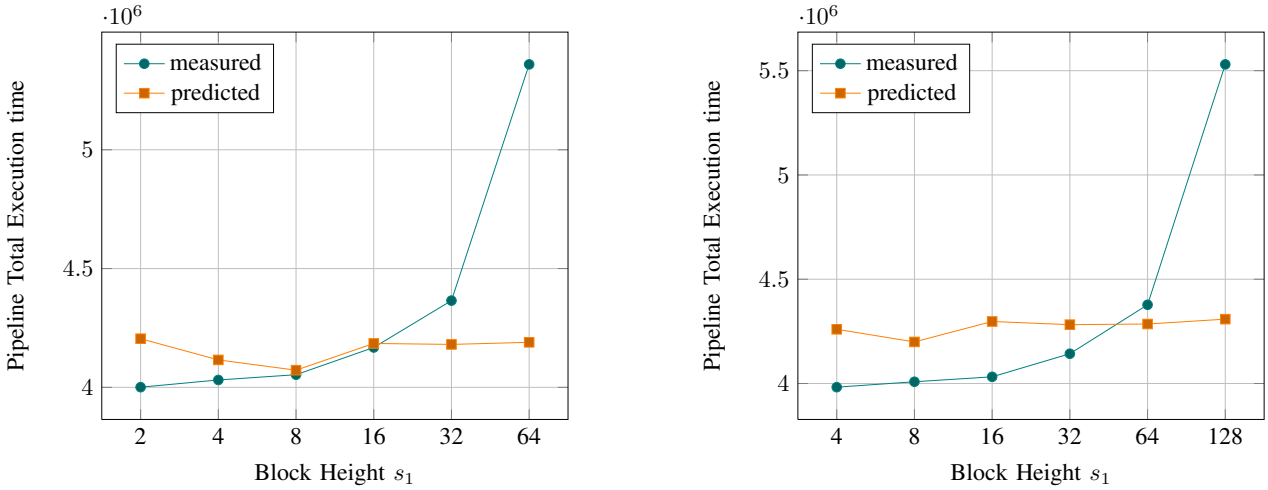


Fig. 11. Predicted and measured values for different combinations of $s_1 \times s_2$

We are of course aware of the fact that each real program and each architecture will have its own particularity, more complex and richer in parameters than the model we have built but we believe that this is a first step toward making such decisions more systematic than by pure trial and error.

Our major observation the experience is that for this type of "collaborative" applications where a large computational task is split, executed and then merged, it is preferable to have a centralized DMA mechanism that can schedule data transfers at the super block rather than the packet level. This way, useless contentions between sub tasks of the *same* application can be avoided. At least in terms of predictability, such policies, used for example in the context of hard real-time systems such as in automotive control [11] will be much simpler. In the future we intend to refine the model and make it more accurate. Then we plan to replicate this work for applications currently being developed for the P2012 architecture where the DMA is more centralized, and the local memory is *shared* between all the cores that reside in the same cluster.

REFERENCES

[1] Platform 2012: a many core programmable accelerator for ultra efficient embedded computing in nanometer technology, 2010.

[2] A. Agarwal, D. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared memory multiprocessors. *IEEE Trans. Parallel Distributed Systems*, 6:943–962, 1995.

[3] T. Altilar and Y. Paker. Minimum overhead data partitioning algorithms for parallel video processing. In *Proceedings Domain Decomposition Methods Conference*, pages 25125–8, 2001.

[4] K. Fatahalian, D. Horn, T. Knight, L. Leem, M. Houston, J. Park, M. Erez, M. Ren, A. Aiken, W. Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 83–es. ACM, 2006.

[5] M. Gschwind. The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, 2007.

[6] IBM. Cell SDK 3.1. https://www.ibm.com/developerworks/power/cell/.

[7] IBM. Cell Simulator. http://www.alphaworks.ibm.com/tech/cellsystemsim, June 2009.

[8] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49:589–604, July 2005.

[9] C. kin Lee and M. Hamdi. Parallel image processing applications on a network of workstations. *Parallel Computing*, 21(1):137 – 160, 1995.

[10] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26(3):10 –23, 2006.

[11] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[12] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879 –899, may 2008.

[13] S. Saidi, P. Tendulkar, T. Lepley, and O. Maler. Optimizing explicit data transfers for data parallel applications on the cell architecture. *ACM Trans. Archit. Code Optim.*, 8(4):37:1–37:20, Jan. 2012.

[14] J. Sancho and D. Kerbyson. Analysis of double buffering on two different multicore architectures: Quad-core Opteron and the Cell-BE. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE.

[15] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[16] S. Schneider, J. Yeom, and D. Nikolopoulos. Programming multiprocessors with explicitly managed memory hierarchies. *Computer*, 42(12):28–34, 2009.

[17] S. Yeom, B. Rose, J. Linford, A. Sandu, and D. Nikolopoulos. A Comparison of Programming Models for Multiprocessors with Explicitly Managed Memory Hierarchies. 2009.