Energy Usage in an Embedded Space Vision Application on a Tiled Architecture

Peter M. Kogge*
Univ. of Notre Dame, Notre Dame, IN, 46556, USA
Benjamin J. Bornstein†
and Tara A. Estlin ‡
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

The need for greater autonomy in platforms such as planetary rovers is driving rapidly to codes that far overwhelm the capabilities of conventional space-qualified single core processors to run them in real-time. However, a new generation of potentially space-qualified 2D "tiled" multi-core microprocessor chips is emerging with significant performance potential. Leveraging such inherently parallel hardware for space platforms requires consideration of both time *and* power limitations - the latter of which is not normally done in conventional parallel computing. This paper takes one such application, Rockster, and analyzes it for energy usage when ported to a multi-core tiled chip such as may come from the Maestro program. The results demonstrate not only the criticality of memory and interconnect in the energy of real-time parallel codes, but also the effects of possible "energy-aware" changes in partitioning and algorithm design.

I. Introduction

Multi-core processors are chips that contain several processing "cores," each capable of running a separate program thread. The commercial marketplace is seeing a wave of such designs that offer significant performance and throughput potential, but require new programming models if this is to be achieved. One variant is a "tiled" architecture, where the chip is designed as a relatively large array of simple processing core "tiles" that can communicate directly with their nearest neighbors in a 2D topology. Applications are parallelized so that all tiles can be running in parallel, each executing a potentially different piece of code on a different piece of data from a larger object. These code fragments communicate directly with other fragments in neighbor tiles, and indirectly through multiple hops with tiles anywhere in the array, and with external memory or I/O channels.

When one thinks about energy and power dissipation during a computation, the normal focus is on the power dissipated by the functional units in the tiles' cores themselves. However, as this paper will show, the energy involved with both communicating between tiles, and with accessing memory (both within a tile and from external memory) is extremely significant, perhaps to the point of being the dominant cost.

To show this we will use an application called ROCKSTER whose function is to find rocks in images that have interesting properties so that an autonomous navigation system in a rover can steer to them. This application is particularly good for the kind of study done here because it is much more than a simple single-loop kernel, with a great many interacting aspects that need to be considered in order to come up with the best low latency and low energy implementation.

The rest of this paper is organized as follows. Section II described the assumed processor structure. Section III discusses the term "energy" as used here. Section IV overviews the Rockster application. Section V discusses the effects on energy of data partitioning. Section VI then looks at one key computational kernel of Rockster, and its energy effects. Section VII summarizes the results.

*McCourtney Prof., CSE Dept., 384 Fitzpatrick Hall, Notre Dame, IN 46556, non-AIAA member.
†Senior Technical Staff, M/S 306-463, 4800 Oak Grove Drive, Pasadena, CA, 91109, non-AIAA Member.
‡Senior Technical Staff, M/S 301-260, 4800 Oak Grove Drive, Pasadena, CA, 91109, non-AIAA Member.

II. Tiled Multi-Core Processors

A tiled multi-core processing chip is one that contains multiple separate cores in a regular 2D arrangement. Perhaps the first such tiled multi-core chip, the 8-core EXECUBE,[5] was in fact designed for aerospace purposes (radar tracking) in the early 1990s. More recent research designs include the 16-core RAW,[8] the TRIPS chip,[6] and the Intel 80-core teraflop chip.[7] RAW in particular led to successful commercial products such as the 64 core Tile64®,[1] and now the 49-core OPERA MAESTRO Processor, a "radiation hard by design" version of the Tile64 being developed by the Boeing Company.[3a]

Fig. 1 diagrams the generic layout of a typical processor chip of this class of architecture. The central region of the die is a 2 dimensional array of identical "processing tiles," with the "edges" of the die holding circuits for memory controllers (to access data from external DRAM chips), and I/O interfaces (to send and receive data from sensors, other subsystems, I/O devices, etc).

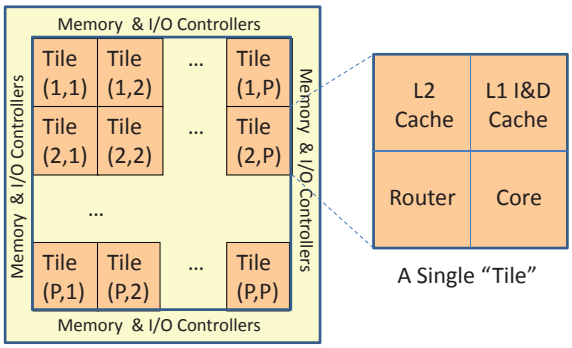


Figure 1. A Generic Tiled Processor Chip.

Each core tile has a structure as shown on the right of Fig. 1. There is a processing core capable of executing more or less conventional programs. There is also a cache hierarchy to hold programs and data needed by the core when it is active. However, unlike more conventional cores, there is no direct path to off-chip memory. Instead there is a router unit that connects this tile to tiles above and below, and to the left and right. The current tile may inject messages into the router, and receive responses in return.

The router then determines where the message should go: up/down, left/right. Depending on the chip design this routing may be based on the address of the data that is desired, or on the basis of some destination tile number. In some systems like Tilera and Maestro there may in fact be multiple such communications paths, each of which routes messages on the basis of different information. For Tilera, for example, there are five such variations, all of which may be in action at the same time.

For this paper we focus primarily on just one such path - one that routes memory access requests from a tile to one of the memory controllers on the periphery of the chip. For Tilera, there are four of these, two each on the top and bottom of Fig. 1. Memory requests from a tile (typically its L2 cache on a "miss") are injected into the local router, and then passed from tile to tile through their router to reach the appropriate memory controller. This controller then accesses the external memory, and, using information from the original message, sends the data (for a read) back to the appropriate core.

There are mechanisms that allow memory data to be held in some other tile's L2, and so in this case memory references to it are routed to that tile rather than a memory port.

III. Energy of Computation

Power is defined as the energy used per second, with units of joules for energy and watts (joules per second) for power. The normal focus of discussion, at least for spacecraft, is on power, as that determines much of a spacecraft's dynamic environmental requirements, such as size of solar panels and/or heat dissipation needs. However, energy is perhaps more fundamental in that it is *the* fungible resource - it is a measure of what you can do with a battery of a certain capacity, or what you must dissipate to do some basic computation - regardless of how long you have to do it. Knowing the energy needed for a computation provides novel

[a] http://www.zettaflops.org/spc08/Malone-Govt-OPERA-FT-Spaceborne-Computing-Workshop-rev3.pdf

insight during algorithm design, especially for tiled architectures where you may be capable of "dialing in" the number of tiles you want for a specific step. Knowing the energy per cycle also allows specification of the maximum clock rate at which the tiles can be run before exceeding some power budget.

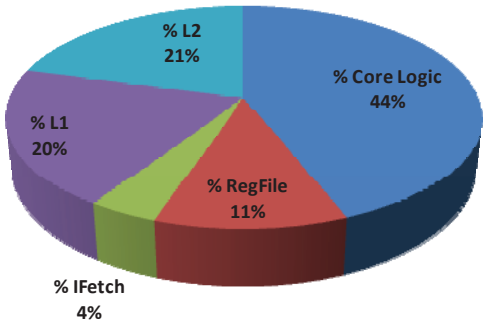
The energy/power dissipated by a chip comes in two forms: *dynamic power* that is dissipated when the chip is actively computing, and *static or leakage power* that is dissipated even when the chip is idle. This paper focuses just on dynamic energy of computation, leaving other issues for a later time. The emphasis is on breaking this dynamic energy into small pieces that can be summed as appropriate for individual steps of the overall algorithm, and studied as a function of the number of tiles to be allocated to the computation.

We use as an estimation baseline some published numbers for the Tile64®. This chip has 64 tiles, which when fabricated on a 90nm process running at 1 volt has various quoted overall chip powers running different applications of 10.8W at 750MHz,[1] or more recently of between 15 and 22 watts at 700MHz[b]. A presentation[c] in 2007 quoted power "per tile" at between 170mw (600MHz) and 300mw (1GHz).

Energy per compute cycle is power over clock rate, and dividing overall chip power by 64 gives an upper bound on per tile numbers. Such calculations over the above data provides per-tile estimates of 283-300 pJ (pJ = "picoJoule" or 10^{-12}J) per cycle from the per tile estimates, and 334-491 pJ per cycle from the more recent chip estimates.

Since we are interested in the contributions from different parts of the whole chip architecture, we will simply assume as a baseline that approximately 15% of a chip's power is static, and thus a tile by itself takes about 250pJ per cycle at 1 volt. We also assume that the higher numbers reported above include the energy of exercising the memory and I/O interfaces.

To get more insight into the energy costs of different operations, we made some assumptions about the internal structure of the major cache structures, and used the CACTI tool[d] from HP Labs to estimate the energy per access of a variety of memory structures found in Tile64. From these numbers and some assumptions about the number of cache accesses per cycle and average hit rates, we then estimated how much of the 250pJ per cycle are spent in the caches, and how much in the dataflows of the core itself. Fig. presents a pie chart of this breakdown. Around 40% of the energy is spent in the caches and the rest in the data flow. We call this dataflow-only energy expenditure a unit of *compute energy* (*CE*).



250	pJ: Average tile energy/cycle
50	pJ: Average L1 access energy/cycle
52	pJ: Average L2 access energy/cycle
148	pJ: Average Compute Energy (CE)/cycle

Figure 2. Approximate Energy Breakdown within a Tile.

It is instructive to delve a bit deeper into what happens when a tile core makes a memory reference. The left side of Fig. 3 diagrams most of the possible sequences of events that might occur as a result of such an access, with the only events listed here being ones that clearly consume a non-trivial amount of energy. We note that there are many possible outcomes for the same access (there could be an L1 cache hit, an L1 miss but L2 hit, a miss out of L2, etc), with multiple side sequences that might occur (a TLB miss, the need for a cache line flush, ...). We used the estimates discussed above to tag each event in these sequences with an energy cost, and then summed to get the energy for various outcomes. We note that the transport energy for a core to access off-chip DRAM assumes accessing the closest of the 4 memory ports - accessing any of the others would be higher. The table on the right side of Fig. 3 lists some of these overall energy costs.

It is important to note that these estimates are just that, and are used here to do comparative analysis on application characteristics, not make absolute predictions of a specific chip's characteristics.

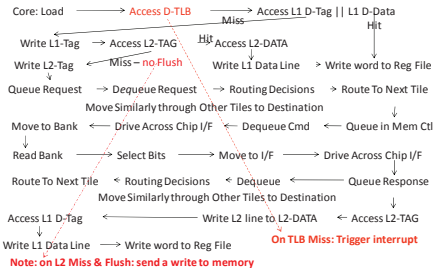
[b] http://www.tilera.com/sites/default/files/productbriefs/PB010_TILE64_Processor_A_v4.pdf

[c] http://www.hotchips.org/archives/hc19/2_Mon/HC19.03/HC19.03.04.pdf

[d] http://quid.hpl.hp.com:9081/cacti/ - we assumed low operating power logic uprated from 0.9 to 1.0V, with conservative interconnect and semi-global outside mat wiring.

The single most eye-opening number in the set is the cost of a memory access that requires going all the way out to the off-chip memory. This energy is *in excess of 225 CEs* - 225 times the energy of a cycle of computation in the core. If operands are in the core's register files, we can perform 225 cycles of computation (potentially over 600 operations in with a Tile64-like 3-way VLIW micro-architecture) before expending the energy cost of reaching out to memory just once.

The rest of this paper focuses on what such a dichotomy means to a real application.



pJ	Event Type	Eqvt CEs
89	L1 Hit; RegFile load	0.60
651	L1 miss, L2 Hit, L1/RegFile fill	4.41
33257	L1, L2 miss; DRAM access; cache fills	225.25

Figure 3. The Energy Expenditure Path of a Load.

IV. The Rockster Application

The Rockster algorithm is an application that has been developed to aid in the autonomy of rovers and other platforms that are too far from Earth for direct real-time control.[4] The purpose of Rockster is to take a 2D image of a platform's surroundings, and quickly determine where there are rocks and what are their visible attributes. This knowledge can then be used either to steer the vehicle to avoid such rocks, or perhaps to trigger a more exhaustive science study of individual specimens that exhibit some unique properties.

The input to Rockster is an NxN pixel image (basedlined here at 1K by 1K of 8 bit pixels) such as Fig. 4. An intermediate output from the algorithm is a "rock mask," a binary image with values "rock" or "not rock." The final output of the part of the algorithm we study here is a "rock description," namely the "edge list" of pixel positions in the image that represent rocks.

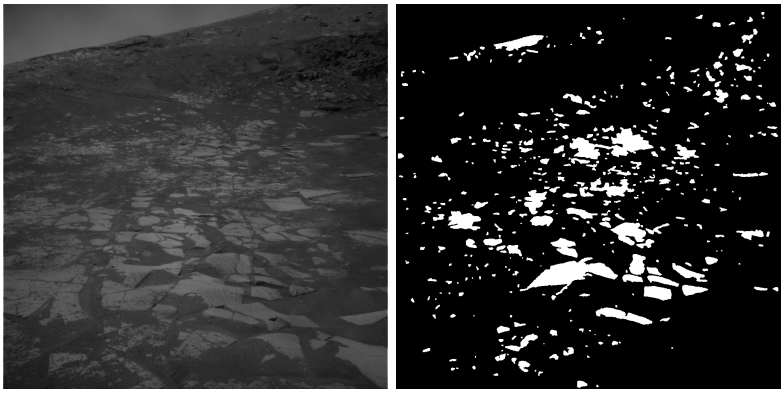


Figure 4. A typical image with a corresponding rock mask.

At the time we performed the investigation discussed here (spring 2010), the individual steps of Rockster were, at a high level, as follows:

1. Assuming a sensor has streamed an image into memory, move the image from memory to the tiles. Note that this requires splitting the image into pieces for the individual cores.
2. Apply a 5x5 Gaussian smoothing algorithm to the image, followed by a 3x3 Canny edge sharpener. This enhances the boundaries between different objects in the scene to make finding them easier.
3. Perform a "horizon determination" (sky fill) to find those regions of the image where rocks will not be found, regardless of contrast (such as in the sky).

4. Look for sharp differences in contrast between neighboring pixels, and use these boundaries to perform an "edge following."

5. Perform a background fill to generate the binary rock mask, and write the mask back to memory for other applications to use for rock analysis.

6. Perform a connected component analysis to trace the edges of rocks in this mask, and report back an "edge list" of pixels describing each rock.

Since 2010, work has advanced towards implementing Rockster on a multi-core processor.[2]

V. Energy in Access and Placement

In this section we look at the energy consequences of partitioning images into "sub-images," and moving them initially into the tiles from memory, and then running algorithms on the tiles that require communication between tiles holding neighboring sub-images.

A. Initial Partitioning

As above, we assume an NxN image is to be broken into sub-images, with each sub-image moved to a separate tile out of an array of PxP tiles devoted to this algorithm. Since nothing is known a priori about the image, the partitions are assumed of equal size, and small enough to remain resident in a tile's L2 once loaded (this is to avoid continued exposure to the 200X cost of referencing memory discussed earlier). As a reference, for a 1 megapixel image and an array of 7x7 tiles as found in Maestro this is equivalent to about 21KB of data per tile - enough to fit into almost any L2.

We also assume that each tile knows in advance which sub-image of the image is its, and is in charge of requesting the data before computation begins. If we assume 64 byte cache lines in the L2 of a tile and one byte per pixel, each core then makes $2^{20-6}/P^2$ line requests from memory, which results in one read request wending its way from the tile to the memory controller, and a line of data back again.

For a tile at position [i;j] in the array relative to the memory controller, this requires the request and data passing through at least $|i + j - 1|$ different tiles in the process. If we account for this energy, and sum up over all tiles of a 7x7 array as will be found in Maestro, we find that the total energy just to perform this initial load of all sub-images is equivalent to **3.2+0.15*P Million CEs!** - the energy cost of just loading the data is equivalent to *millions of cycles* of computation.

We note that paradoxically if we were to move to a more advanced chip with say 10x10 tiles (so that we could pick up some potentially additional performance), the energy costs for just loading the image would go *up* by over 25%!

B. Partitioning and Mapping

There are at least two ways that our images can be partitioned onto the array of tiles. First, the obvious approach is to partition the NxN image into (N/P)x(N/P) square blocks, one per tile. However, a second approach is to "slice" the image into rectangular slices of (N/P^2xN) pixels, again one per tile. Each slice stretches from one end of the image to the other. Both of these are pictured in Fig. 5.

In either case, the way pixels are stored in memory and in the tile's L2 makes a difference. For this paper we assume that neighboring pixels from a row in the image are in sequential locations in memory, and thus tend to pack tightly into cache lines. This also means, however, that pixels that are "neighbors" in the image's column direction probably fall in different cache lines.

C. "Ghost Cells"

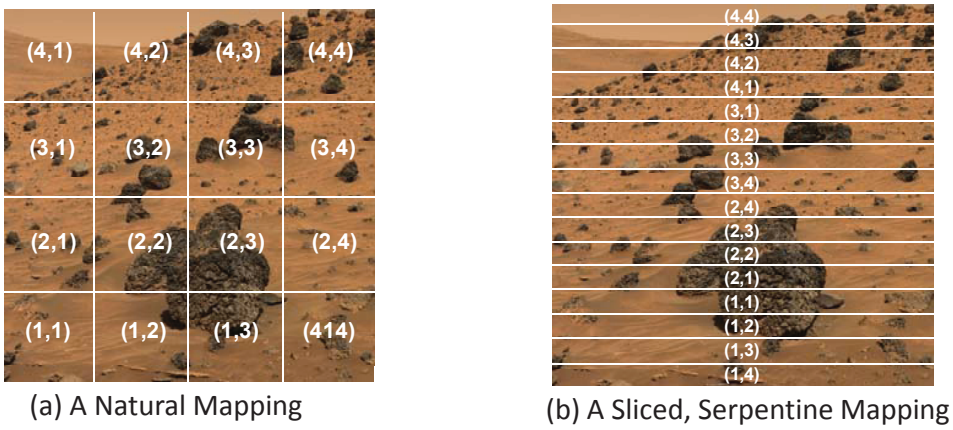
Given that we assume we are loading equal-sized sub-images in each tile, a related question is what is the shape of these sub-images, and which sub-image goes in which tile. If computations within each sub-image were totally independent of each other, this would not matter, but in sophisticated enough algorithms (or where the sub-images begin to get smaller than a single rock), it does matter.

Within the parallel computing community this situation is frequent, and the typical solution is to ensure before any major computational step that each tile exchanges data values at the boundaries of its sub-image

with its neighbors, so that the algorithm will have immediate access to these values once inside the inner loop. These boundary values are often called *ghost cells*, and their exchange a critical part of the algorithm. Typically, one tile may need ghost cell values from its neighbors, but does not need to modify them (it modifies only its own sub-image).

For example, the 5x5 Gaussian smoother discussed earlier as part of Rockster would need 2 rows of pixels from the tiles holding neighboring sub-images before it could update the outermost pixels of its own sub-image. The other steps of Rockster could probably get away with only a 1 deep edging of ghost cells for their processing, but this 1 deep pattern is different data at each step of the overall algorithm. Thus the ghost exchange is liable to be a frequent part of an overall computation.

Exchanging these ghost cells could be done by memory accesses from each tile that get resolved into references to neighboring tiles' L2, or (as is possible on Tile64) by a direct exchange with a neighboring tile. Not counting the cost of sending the request, the energy to move a cache line of data from one tile to another is the cost of reading the remote L2, transporting the line over however many tiles are required, and then writing the data into the local L2 in a place where the algorithm can access it naturally. With our previous estimates this energy is about equal to 26+10T CEs per line moved, where T is the number of tiles traversed.



(a) A Natural Mapping

(b) A Sliced, Serpentine Mapping

Figure 5. Two different methods for partitioning sub-images.

To minimize this energy we want to minimize both T and minimize the number of lines of ghost data to be moved. To minimize T, we want tiles to be neighbors; this is easy for the square pixel sub-array as pictured on the left of 5; the pixel array starting at (iP, jP) is placed in tile (i, j). However, lets look at the number of cache lines from a square that have to be moved. With pixels assumed stored in row order, the top and the bottom ghost cells are easy; the row pixels are tightly packed in cache lines. The left and right sides, however, are different. Each pixel is in a different cache line from the square where it originates. To move it into a cache line in the destination still requires a cache line read at the source (even though its only a single pixel), AND THEN a cache line read at the destination so that the single pixel can be inserted in its correct spot before it is written back. On a per pixel basis, the energy costs for a column ghost pixel is thus several times more expensive than a row pixel. In fact, there are only 2*N/(P*64) lines for the top and bottom, but 2*N/P lines for the sides, for a total of 2*(N/P)*(65/64).

Now consider the slice approach on the right of 5. There are longer top and bottom ghost areas (N pixels), but no sides, for a total of 2*N/64. This is a factor of 65/P less, or close to a factor of 10 for a P of 7.

This reduction in exchange energy is achievable ONLY if the tiles that hold logically neighboring data are in fact physical neighbors in the tile array. To do this requires a "serpentine" mapping of slices to tiles, as shown by the indecies of Fig. 5.

Fig. 6 diagrams the energy of ghost exchange for both layouts. As can be seen, there is an explosive increase as we move to future generations of processors with more and more tiles.

	P				
	2	4	8	16	32
Square	149,760	299,520	599,040	1,198,080	2,396,160
Slice	78,336	92,160	147,456	368,640	1,253,376

Figure 6. Ghost data exchange as a function of tile array size (in units of CE).

VI. Energy in Computation

The effects of the memory hierarchy on energy are present even when no data enters or leaves an individual tile. To see this, we consider here the smoothing and edge sharpening phases of Rockster, and assume at the start that all data needed for this tile's sub-image, including any ghost data from neighbors, is present in at least the tile's L2. We also organize the computation to try to maximize hits in L1. The computation proceeds across the columns (to match the assumed storage of pixels in memory) one row at a time, starting with the upper left pixel, and involves pixels of three types:

- "raw" pixels that have not been smoothed or sharpened,
- "smoothed" pixels that have been processed by a 5x5 filter centered around the each pixel, using the raw pixels.
- "sharpened" pixels that have been processed by a further 3x3 computation centered around each pixel, using all smooth pixels.

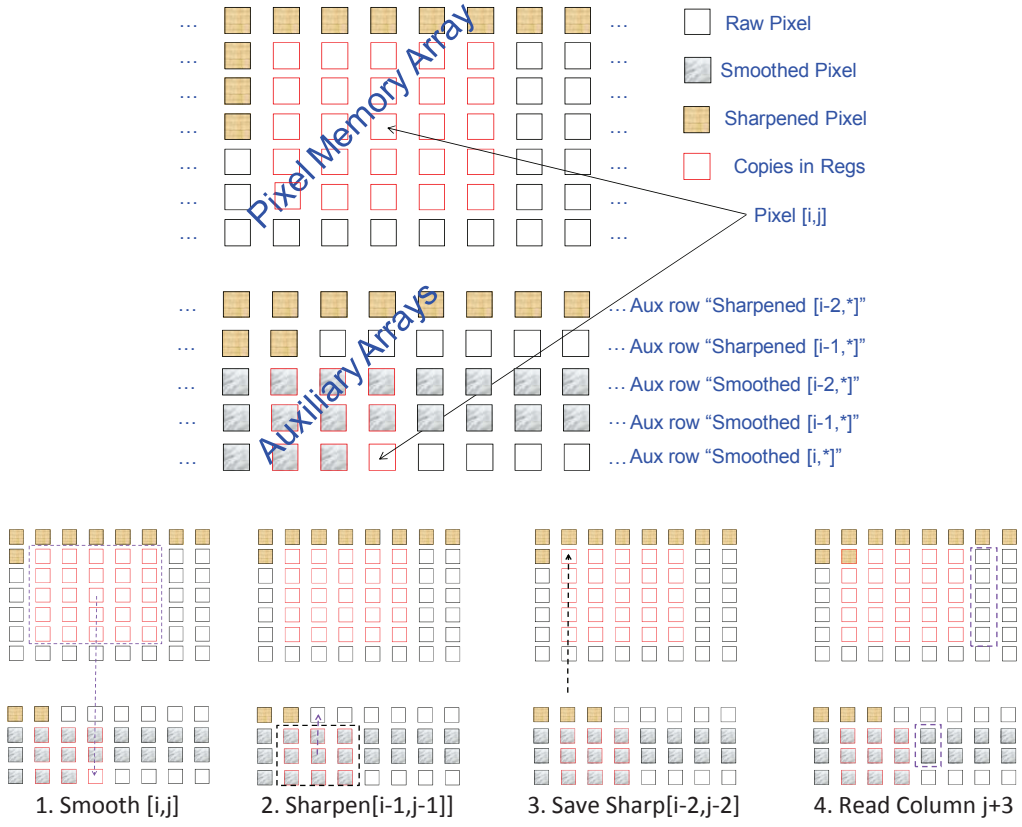


Figure 7. Data movement in the filtering process.

A. Data Arrangement

We assume several arrays in L2, as pictured in the top of Fig. 7:

- An Nx(N/P+4) pixel memory array holding the pixel data both as input and as processed. The "+4" in the row dimension represents two rows of ghost data from each of the neighboring slices. The middle N/P rows hold the actual pixel array owned by this tile. We assume at any point in time we are processing point [i,j], and move along the i'th row one j position at a time from left to right. We also assume that there are 16 bits devoted to each pixel. 8 bits are used at the beginning to hold the raw image value, and 16 bits at the end hold sharpened magnitude and angle.
- An auxiliary N element array holding the "sharpened" values from row i-2.
- An auxiliary N element array holding the "sharpened" values from row i-1.
- An auxiliary N element array holding the "smoothed" values from row i-2.
- An auxiliary N element array holding the "smoothed" values from row i-1.
- An auxiliary N element array holding the "smoothed" values from row i.

B. A Processing Step

The bottom of Fig. 7 tracks the data movements for the processing of the smoothed [i,j] pixel, and the sharpened [i-1,j-1] pixel. The sequence assumes the data outlined in red is, at the beginning, in 5x5 + 3x3 = 34 core registers, and proceeds through the following steps:

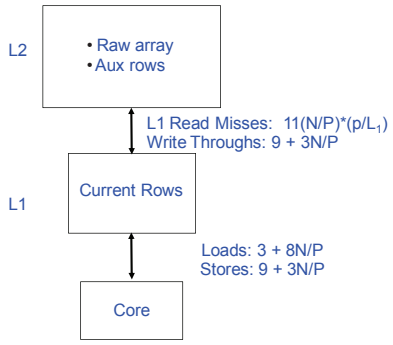
1. The 5x5 raw pixels are used to create a new smoothed pixel [i,j], which is stored in one of the 3x3 registers holding a chunk of the smoothed auxiliary rows.
2. We now have in registers a full 3x3 smoothed array that allows sharpening of its center [i-1,j-1] pixel. This value is saved in both a register and the sharpened i-1 row.
3. The [i-1,j-2] element from the auxiliary sharpened row i-2 can now be stored back in its place in the pixel array, overwriting the raw value that is no longer needed for smoothing.
4. The next 5 element column of raw pixels ([i-2, j+2] through [i+2, j+2]) is read from memory into registers, overwriting the [*,j-2] column values that are no longer needed. Likewise, [i-1, j+1] and [i, j+1] are read into registers from the auxiliary smoothed rows.

At the end, we are now ready to work on smoothing [i,j+1], and sharpening [i-1, j]. We note that smoothed values never go back into the pixel array. This saves data movements. Also, this approach is very efficient on cache lines in L1; 10 lines holds the working chunks of of all these arrays, with hits except when the new column is read in.

C. Starting a New Row

Once all the N values in a row have been computed, the algorithm starts on the next row.

- The auxiliary sharpened [i-1,*] array becomes the sharpened [i-2,*] array.
- Likewise the two smoothed [i-1,*] and [i,*] arrays "move up."
- A 5x5 block of raw pixels are read into registers to initialize the computations.
- Whatever special computations that are needed to compute the first 8 smoothed and 1 sharpened values are done.



Tiles	4	16	64	256	1024
Filter	P				
Energy	2	4	8	16	32
Subimage	3.9E+07	3.9E+07	3.9E+07	4.0E+07	4.0E+07
Sliced	3.2E+07	3.2E+07	3.2E+07	3.2E+07	3.2E+07

Figure 8. The Energy of Cache Traffic in Filtering (in CEs).

D. Counting the Data Movements

The main loop of the algorithm involves on a per pixel basis: one write in each of steps 1 and 2, a read and write in Step 3, and a set of 7 reads in Step 4. This gives a total of 8 reads and 3 writes per pixel. If each pixel takes p bytes and the size of an L1 cache line is L_1, then these accesses are all hits for L_1/p steps in a row, before there are misses to bring in new lines from L2. Also, when flushes are needed from the L1 to the L2, the whole line contains real data. Finally, there are N/P rows processed in the tile.

Fig. 8 summarizes all these transfers and gives the equivalent energy in ECs, summed over all tiles in a PxP array. Dividing by N^2 (1024^2 - the number of pixels) gives the average energy spent in just data movements to local L1 and L2 caches. This adds up to around 30-40 ECs per pixel - that is, it would take 30-40 cycles of computation in the core for each pixel to match in energy the energy of data movement.

VII. Conclusion

The computing world as a whole has moved inexorably to multi-core processor chips as the primary way to continue Moore's Law through future technology generations. Space computing is following, albeit slower because of the harsh and unique environmental conditions that must be faced.

At the same time there are a growing number of applications such as Rockster that will become essential to providing the kinds of autonomy needed to carry out future missions, and these applications sorely need the kinds of enhanced processing possible with such new chip architectures. Space, however, is even more concerned about power and energy usage than commercial; the difference is not just increased "talk time" but ability to perform a mission.

Thus, it will be essential to begin to design algorithms for such processors that from the beginning are energy-aware in both design and implementation. This paper has taken several aspects of Rockster (initial image load, ghost data exchange, and filtering), and analyzed where the energy goes. Although the individual numbers were based on assumptions and projections, the ratios are wide enough to be indicative that the general message of the paper is true.

The key message is that computation by itself is now cheap energy-wise. Overall algorithm energy usage for tiled architectures is liable to be dominated by memory and interconnect, and that increasing the number of tiles (to improve time performance) very often will have a negative effect on overall energy. It takes more energy to distribute an image, and much more energy if individual tiles have to exchange data at each computational step. We have also shown that different mappings of data to tiles can influence these energy costs, often significantly, and that non-obvious arrangements can be significantly better.

Further, we have also shown indications that even when a tile's activity is all local (i.e. the sub-image is absolutely local to the tile's caches), and we are very careful on maximizing cache hits, that the pro-rated energy to access each pixel is equivalent to dozens of compute cycles.

Looking forward, it is clear that maximizing the usefulness of tiled processors for space applications will take more than just an understanding of how to efficiently parallelize an application. Issues related to data placement, layout, and exchange will in fact dominate the energy, and thus power, picture. Thus, it is crucial that our software engineering practices grow to consider both. Energy must take an equal role to

performance, and can lead, if done right, to really effective power-efficient algorithms.

Appendix

Acknowledgments

This work was performed by the University of Notre Dame and by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

References

[1]S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88 –598, 2008.

[2]B. Bornstein, T. Estlin, and P. Springer. Using a multi-core processor for rover autonomous science. In *Proceedings of the 2011 IEEE Aerospace Conference*, 2011.

[3]M. Cabanas-Holmen, E.H. Cannon, A. Kleinosowski, J. Ballast, J. Killens, and J. Socha. Clock and reset transients in a 90 nm rhbd single-core tilera processor. *Nuclear Science, IEEE Transactions on*, 56(6):3505 –3510, 2009.

[4]T. Estlin, B. Bornstein, D. Gaines, D. Thompson, R. Castano, R. C. Anderson, C. de Granville, M. Burl, M. Judd, , and S. Chien. Aegis automated targeting for the mer opportunity rover. In *Proceedings of the 10th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS 2010)*, 2010.

[5]P. Kogge. The EXECUBE approach to massively parallel processing. In *Int. Conf. on Parallel Processing*, August 1994.

[6]K. Sankaralingam, R. Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, D. Burger, S.W. Keckler, and C.R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 422 – 433, 2003.

[7]S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28tflops network-on-chip in 65nm cmos. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 98 –589, 2007.

[8]E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86 –93, September 1997.