FPGA vs. MPPA for Positron Emission Tomography Pulse Processing

Michael Haselman[1], Nathan Johnson-Williams[1], Chad Jerde[1], Maria Kim[1], Scott Hauck[1], Thomas K. Lewellen[2], Robert Miyaoka[2]

[1] *Electrical Engineering Department, University of Washington*
Seattle, WA USA
{haselman,nathanjw,chadj3,mbkim,hauck}@ee.washington.edu

[2] *Department of Radiology, University of Washington*
Seattle, WA USA
{tkdog,rmiyaoka}@u.washington.edu

***Abstract*—As FPGAs follow Moore's Law and increase in capacity and complexity, they are becoming more complex to use and are consuming increasing amounts of power. An interesting alternative for reconfigurable computing that is lower power and may be easier to program are Massively Parallel Processor Arrays (MPPAs). In this paper we investigate the Ambric AM2045, a commercial MPPA. To understand the differences between the architecture and computational models of MPPAs and FPGAs, we have implemented two pulse-processing algorithms used in Positron Emission Tomography (PET). The algorithms for event timing and event location were developed for FPGAs and then adapted to MPPAs. In this paper, we present the two implementations and discuss the main differences. Specifically, we show how the MPPA's lack of a real-time mode, their distributed memory structure, and object based programming model posed challenges for these algorithms.**

I. INTRODUCTION

For the last two decades FPGAs have been following Moore's Law, and the latest devices contain more than 500,000 logic elements, megabits of memory, a thousand dedicated multipliers, and very sophisticated I/O. Unfortunately, these advancements have created challenges for FPGA users. These include design complexity and power consumption. While increased transistor counts have allowed FPGAs to support larger and more complex circuits, it has also increased the complexity of the designs. This has resulted in slower design, debug and verification cycles for new designs.

A promising technology that attempts to address these issues is massively parallel processor arrays (MPPAs). A commercial version of the MPPA, the Ambric AM2045 [2], has recently become available. MPPAs can reduce the configuration overhead of FPGAs because they are configured at the word level; one configuration bit can route 32 signals through a switch instead of one bit per signal. To address the design complexity problem, Ambric has developed an efficient programming model. In the Ambric model, each processor independently executes a single encapsulated Java object, and the processors communicate through a network of self-synchronizing registers.

Previous work [3,4,5,13] has shown that the Ambric MPPA is simple to program and achieves performance close to an FPGA. In this work, we investigated the architecture of the Ambric MPPA and how some of the constraints of the computation model affect an embedded process such as Positron Emission Tomography (PET) data acquisition and pulse processing.

II. AMBRIC MPPA

The Ambric AM2045 MPPA is a 2-D array of RISC processors [6,7]. The array consists of 360 32-bit processors with 360 1KB RAM banks. Each processor executes a single Java object that is strictly encapsulated. The processors are connected together with a network of self-synchronizing channels, removing the need to globally synchronize the system.

Fig. 1 shows the architecture of the compute units (CU) and RAM units (RU) of the Ambric MPPA. Each CU contains four RISC processors. Two of the processors (SR) are simple 32-bit streaming processors that are best suited for simple tasks such as joining channels or address generation. Each SR CPU has a 64 word local RAM for instructions and data storage. The other processors (SRD) are 32-bit streaming processors with DSP extensions for more complicated computations. Each SRD CPU has three ALUs, two in series and one in parallel. A 256 word RAM is available for each SRD CPU. Additionally, the SRDs are directly connected to the RAM unit (RU), which is the main on-chip memory. Each RU has four single-ported RAM banks that are 256 words each, for a total of 32k bits of memory. Two CUs and two RUs are combined together to make a bric. The brics are then tiled together to in a 5x9 array to make the MPPA.

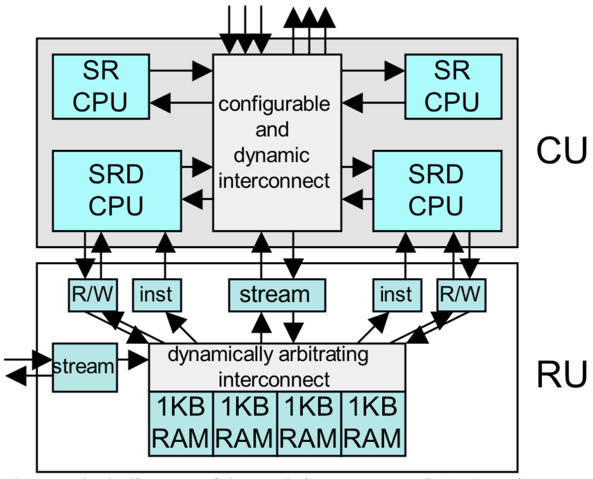

Fig. 1. Block diagram of the Ambric compute unit (CU) and RAM unit (RU).

One key feature of the computation model of the Ambric MPPA is the channels between the processors. The processors in this architecture compute independently of any other processor, and any control or data that needs to pass between processors is sent over these self-synchronizing channels. Self-synchronizing is achieved by using Ambric registers in the channel. Ambric registers are conventional registers with two extra control signals. The two signals control when a register can send or receive data. Each register is 32-bits wide and has a valid and accept control signal. Before a processor can write to a channel, it must assert its valid signal and the receiving register must have an asserted valid bit. If the valid bit is zero, the processor must stall until the valid bit is asserted, signifying the channel is no longer full. Likewise, if a processor needs a piece of data from a channel, it must assert the accept signal and stall until the valid signal is asserted. This communication scheme allows the individual processors to have varying workloads and execution times without having to perform global synchronization.

The programming model that led to the Ambric MPPA architecture is based on a strict subset of serial Java code. The code for each processor is written as a separate Java object. Each object executes serially on a single processor, without any side effects to any other object. The objects are connected to make composite objects or applications using a proprietary language called aStruct. aStruct, statically specifies all processes and communication channels in a design.

III. Positron Emmission Tomography

PET is a medical imaging technique that uses radioactive decays to measure certain metabolic activities inside living organisms. The first step for a PET scan is the generation and administration of the radioactive tracer. A tracer consists of a radioactive isotope and a metabolically active molecule. The metabolically active molecule acts as a carrier to transport the isotope to the tissue of interest. For example, FDG, the most commonly used tracer in PET, is an analog of glucose. This is valuable because cancerous tissue metabolized more glucose than normal surrounding tissue. So, if cancerous tissues are present in a subject that receives FDG, a higher concentration of FDG will accumulate in the cancerous cells and therefore more radiological activity will occur in those cells.

The PET scanner hardware is designed to detect and localize the radioactive decays of the tracer isotopes. One important feature of PET isotopes that makes PET possible is that the final result of the decay process is the emission of two anti-parallel 511KeV photons. A 511KeV photon has a substantial amount of energy and will pass through many materials, including body tissue. While this is beneficial for observing the photon outside the body, it makes it difficult to detect the photon. Photon detection is the task of the scintillator and photodetector detector set. To detect both photons from an event, the scanner is built as a ring of detectors that surrounds the subject. As a photon exits the body, it first interacts with the scintillator. A scintillator is a crystal that absorbs high-energy photons and emits light in the visible spectrum. The photodetector is coupled to the scintillator to convert the visible light into an electrical pulse.

The pulses from the photodetectors are fed into the front-end electronics for data acquisition and pulse processing. The pulses have to be processed to extract start time, location and pulse energy. For the scanner that we are currently building [8], this is performed in an FPGA [10]. In this paper, we will discuss the algorithms to extract the start time of the pulse as well as one of the methods used to determine the location of the scanner.

The start time of the pulse is important for determining coincidence pairs. Coincidence pairs refer to the two photons that arise from a single decay event. Many of the photons from a radioactive event don't reach the scanner because they are either absorbed or scattered by body tissue, they don't hit the scanner, or the scintillator crystal does not detect them. The scanner works by detecting both photons of an event and essentially draws a line that represents the path of the photons. If only one of the two emitted photons hits the scanner, there is no way to determine where the event occurred. To determine if two photons are from the same event, they have to occur within a certain time of each other. Considering the photons travel at essentially the speed of light, the timing portion requires a very precise time stamp be placed on each event. The better the precision of the time stamp, the lower the probability that two separate random events will be paired together. The fewer randoms that are paired together, the better the final image will be.

In addition to determine coincidence, the location of the photon interaction with the scanner is needed to create the final image. Image reconstruction essentially draws lines between the two detectors that detected photons from a single event. Where more lines intersect more activity is present. Precise location of the photon interaction in the scanner will result in more accurate lines, which will produce higher quality images.

IV. Event Timing

As a part of our project for developing a PET scanner [8], we have developed an all-digital timing algorithm implemented in an FPGA [9,10]. This timing technique will replace several custom analog circuits with an ADC and FPGA, while achieving similar timing resolution.

The pulses from the photodetector are first sent to a low-pass filter and then sampled with a 70MHz ADC. The pulse input into the FPGA is shown in Fig. 2. As can be seen, with a sampling period of 14.3ns, the first sample of the pulse will not necessarily be near the start of the pulse. The desired timing resolution is around 2ns, so some form of interpolation is required to calculate the start time.

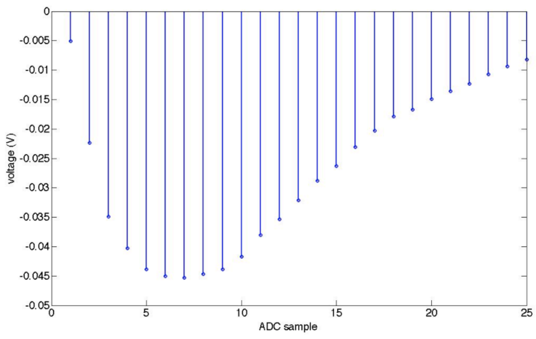

Fig. 2. Event pulse from ADC that is input to the FPGA.

Our timing algorithm is based on using a reference pulse that has the same shape as the event pulses to interpolate the start time. The general idea is to fit the reference pulse to the sampled pulse and then utilize the reference pulse to interpolate the start time. The first step is to detect a pulse from the free-running ADC. This is accomplished by using two triggers. The first trigger is very close to the baseline noise to get the first sample of the pulse closest to the origin. A second trigger is needed to differentiate whether noise or a true pulse tripped the first trigger. The second trigger needs to be high enough that noise does not trip it, but low enough that almost all possible voltages of the second sample of a good pulse trip it. Once a pulse is detected, the amplitude of the event pulse is normalized so that it matches that of the reference pulse. This is accomplished by normalizing the summation of the pulse samples because the pulses all have the same shape. As a pulse is detected all of the 25 samples that make up the pulse are summed and the ratio of the summation of the reference pulse to the data pulse is calculated. The first sample of the pulse is then multiplied by this ratio to normalize the first sample. This normalized first point is then sent to a lookup table to lookup the start time of the pulse based on the sample voltage. To perform this lookup, a reference pulse is precalculated that has the same shape as the event pulses. To determine the start time, the table is reference by voltages, and the output is the time it takes for the reference pulse to reach this voltage. This is referred to as the fine grain portion of the time stamp. Notice that this time stamp is simply breaking up the ADC sampling period into many higher resolution periods. In this implementation, the 14.3ns ADC period is broken into 30ps steps. In order to get the overall time, the fine-grain time stamp is combined with the coarse-grain time, which is a free funning counter at the rate of the ADC. The coarse-grain component of the time stamp is important because time stamps for coincidental pairs are calculated by separate sets of electronics (each detector in the scanner has a separate set of data acquisition electronics) on opposite sides of the scanner. Coincidental events are paired in a post-processing step in the host computer.

A. *FPGA Implementation*

The design for the FPGA was done in Verilog and was pretty straightforward. The system runs at 70MHz to match the rate of the ADC. The bit width is 12-bits to match the precision of the ADC. The divide to calculate the normalization ratio is performed with a lookup table. The lookup table is 83k bits (not all areas are possible). The reference pulse is also stored in memory for the time lookup, and is 26k bits. Finally, the coarse-grain time stamp is simply a 12-bit counter that increments on the system clock. It does not matter that this counter will roll over, because data for each event is sent to the host computer in chronological order. When events from two detectors are compared to find coincidental pairs, the search window is small enough that events from different rollovers won't be paired up.

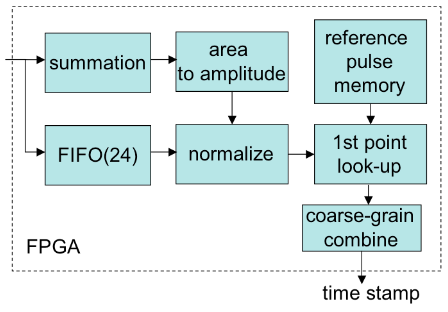

Fig. 3. Block diagram of the FPGA implementation of PET timing.

The design shown in Fig. 3 was place and routed using QuartusII 8.0 for implementation on a StratixII S60 development board. One timing core uses 250ALMs (out of 71,760), 114 kbits of memory (out of 9.4 Mbits), 1 PLL (out of 8), and 36 I/Os (out of 743). The timing core runs up to 150MHz. In the scanner, there will need to be more than one timing circuit since there will be a large number of channels supported by each FPGA.

B. *Ambric Implementation*

The timing algorithm was also implemented in the Ambric AM2045 as an application experiment for the MPPA. The first step in any Ambric implementation is determining how to partition the application into separate objects. Fig. 4 shows how the application was initially partitioned. The first block in the system is the sum block (shown in Fig. 5). This object keeps a running sum of the last 24 samples of the pulse. It achieves this by adding the latest sample to the running total, while subtracting the 25th sample. The current sample is also sent through a 24 stage FIFO so that when it is compared to the triggers, the sum of the pulse samples is already available.

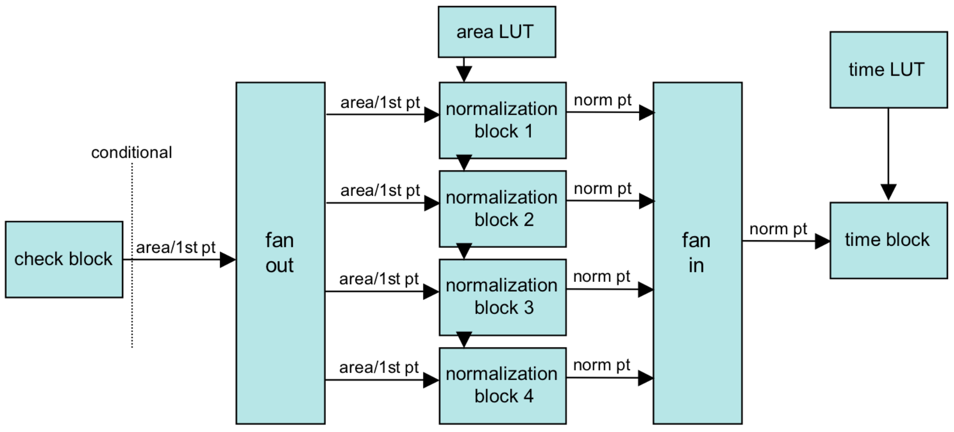

Fig. 4. First partition of the timing application into separate object

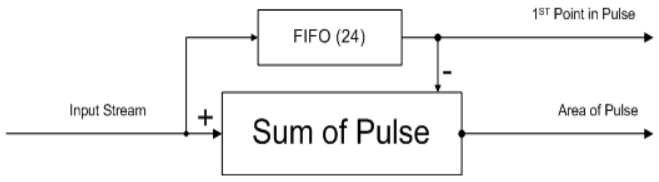

Fig. 5. Block diagram of the sum block.

The check block is where the incoming samples are checked against the two triggers to see if an event pulse has arrived. Notice that there is only one output from this block. This was the first optimization. An Ambric processor takes a cycle to write out each output so if the first point and area of the pulse were sent separately, it would take the check stage two cycles to send it's outputs. Instead, these two channels could be combined to one 32-bit word because they were both less than 16-bits.

Notice that the normalization block, where the pulse area ratio is looked up, had to be manually separated into four different objects and therefore four different processors. This is because the lookup table memory requirement is four times larger than the memory contained in a single RU. The lookup table is required because the Ambric ALU does not support divides. To accommodate this, a fan out and fan in block had to be added. The input to the fan out block occurs only when an event is detected in the check block. This is a scenario where the Ambric channels greatly simplify the control, as the check block processor will only assert the valid signal when an event is triggered, so the fan out processor simply stalls until this occurs.

The fan out block distributes the combined first point and pulse sum to the normalization blocks. Before the data is sent to the normalization block, the lookup address (pulse sum) is adjusted by an offset for each normalization block. Only one of the blocks will receive a valid address, while all others will receive an address out of the memory bounds, and the processors are programmed to output a -1 when this occurs. The normalization block that receives a valid address will normalize the first point and send it to the fan in block. This organization keeps all normalization blocks in lockstep, and avoids event inversion.

Finally, the time block performs the time lookup. Unlike the normalization block, the memory requirements for the lookup table in this block can fit in one RU.

Table I
COMPUTATION REQUIREMENTS OF EACH STAGE BEFORE OPTIMIZATION.

Block	Cycles per input transaction	Input transactions per output transaction
Sum	8	1
Check	10	25 or more
Fan Out	4	1
Normalization	11	1
Fan In	4	1
Time	5	1

Table I shows the computation requirements for each processor for this initial implementation. An input or output transaction is the transfer of a value on all inputs or outputs respectively. While this is a functioning implementation, the large number of cycles per input limited the achievable throughput. Therefore, optimizations were performed to reduce the overall cycles per input. This is akin to pipelining in an FPGA. Notice that the check block requires 25 inputs before it can output a new value because a pulse is 24 samples long. This means that everything downstream from the check block has ample time to compute so there is no need to optimize any of them. Therefore, the optimization effort was focused on the sum and check blocks.

The first optimization was to break up the check block. This object had two if statements to check the two threshold values. Since the if statements were the most costly operation, because of branching, we realized that it would be necessary to break up the if statements into separate blocks.

Fig 6 shows how the single check object has been separated into multiple objects: CheckVfpt checks the first sample threshold, CheckVet checks the second sample threshold, and And determines if both of triggers were crossed. At this point a peculiar thing was noticed; CheckVet took eight cycles while CheckVfpt only took seven cycles. The assembly code revealed the issue. The threshold value for CheckVet

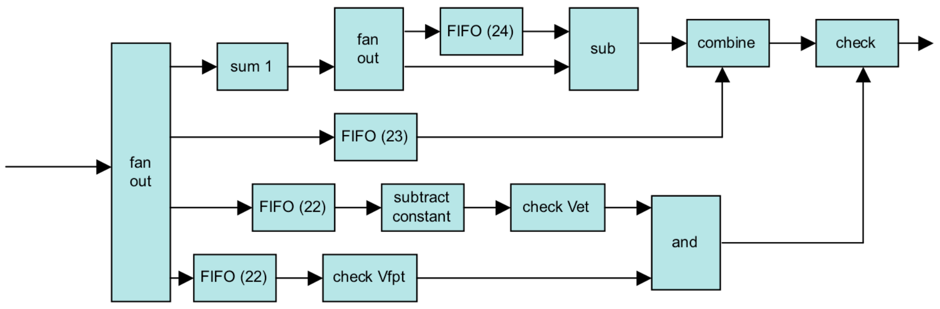

Fig. 6. Block diagram of final optimized version of the sum and check blocks.

required more than 16 bits, so an extra assembly operation was required. For this reason another block before the CheckVet was used to apply the value. This reduced its cycle time 6 cycles. This caused us to realize that writing input stream values to a local variable and then evaluating the variable caused an extra cycle, as opposed to just computing on the stream. Based on this, we were able to reduce all the Check Blocks to 6 cycles. Fig 6 shows the final optimized partitioning of the sum and check objects. The optimized results are shown in Table II.

Table II
Computation requirements of the optimized sum and check blocks.

Block	Cycles per input transaction	Input transactions per output transaction
Dup4	1	1
Sum	4	1
FIFO	1	1
Dup2	1	1
Sub	5	1
Load Const	4	1
CheckVfpt	6	1
CheckVet	6	1
And	4	1
Combine	6	1
Check	6	25 or more

C. Discussion

A few issues arose in the implementation of the PET timing algorithm on the Ambric AM2045 MPPA. The first issue was the inability to create a real-time counter to make the coarse-grain time stamp. This was impossible to do because of the possibility of stalls in the Ambric processor. Stalls can arise for multiple reasons, but the most likely is the processor's need to stall if the output register is not accepting data. While this is an important feature for Ambric's computation model, there may be an easy solution for this. The solution would be extending the processor to perform non-blocking writes. In this scenario, if the check block detected a pulse and tried to output to a channel that was not accepting data, the pulse would have to be discarded to avoid stalls. In many computations this would not be acceptable, but in PET, since final image is made up of the statistics of hundreds of thousands of events, this would not be an issue. This is in fact what occurs in the FPGA implementation; the system is designed to handle the average count rate, but if a short period of high activity occurs, some of the data is not processed.

Another issue is the difficulty creating a memory that is larger than one RU. For example, in the normalization block the lookup table required the memory of four RAM units. In the FPGA tools, this is a simple step: an array of the appropriate size is declared and the tools stitch together the needed memories automatically. This was not the case with the Ambric programming tools. When it was discovered that the normalization lookup table exceeded the capacity of one RAM unit, the table needed be partitioned by hand. With the memories distributed over four RAM units, the addressing had to be managed to ensure that the correct memory was read. In addition to the added complexity, the partitioning of the memory also used three extra processors. Replacing a few scattered brics in the array with larger memories would help with this problem. Alternatively, providing a library of large memory components that automatically compose together smaller memories would be a major productivity improvement.

Finally, as was discussed in the previous section, after the initial implementation the bottlenecks of the system were easy to determine. These bottlenecks prevented the system from achieving the required throughput. To speedup these sections of the code the check and sum objects needed to be manually partitioned into multiple steps in order to pipeline the algorithm. This meant that more processors were used to do the same amount of work, reducing the code density per processor. In an FPGA, while it may be harder to pinpoint the bottleneck, the speedup may be accomplished by retiming with registers. Retiming however can also create synchronization complexities for FPGA designs. Having a mechanism to automatically spread computationally intensive objects across multiple processors and aggregating several simple objects onto one CPU, would be key for making best use of the MPPA fabric.

To create a fair throughput comparison, the design was also compiled to a Stratix device (both Altera Stratix and Ambric MPPA are on a 130nm process). Without any optimizations, the FPGA can run up to 150MHz. While the FPGA only needs to run at 70MHz for the current scanner to match the speed of the ADC, the possible higher clock frequency allows for faster ADCs to be used in the future. The Ambric MPPA on the other hand can only achieve a throughput of 50MHz (300MHz clock divided by 6 cycles for CheckVfpt).

$$\ln(P(\text{event}, X, Y)) = -\sum_{row=1}^{8}\sum_{col=1}^{8}\left[\frac{\left[\text{event}_{row,col} - (\mu_{X,Y})_{row,col}\right]^2}{2\cdot\left[(\sigma_{X,Y})_{row,col}\right]^2} + \ln\left[(\sigma_{X,Y})_{row,col}\right]\right] \tag{1}$$

V. Event Localization

Our lab has been investigating the use a continuous slab crystal for a scintillator crystal [11,12]. The monolithic slab crystal has the benefit of cheaper production, but determining where the photon enters the crystal is more complicated. As can be seen in Fig. 7 the light spreads out substantially from the point of interaction, and many of the sensors on the photodetector receive light.

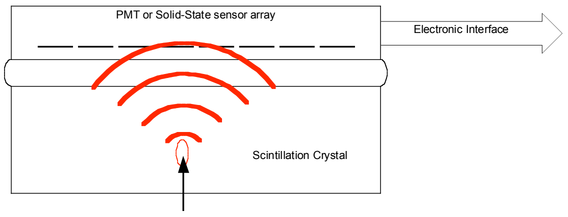

Fig. 7. Dispersed output from a continuous scintillator crystal.

The photodetector in this system has an 8x8 array of sensors. For any event in the slab crystal, a distribution of light will result, as shown in Fig. 8. Notice that all of the 64 sensors receive some light, and there is an obvious peak, but achieving resolution beyond the 8x8 will require additional processing. In this study a resolution of 127x127 is achieved.

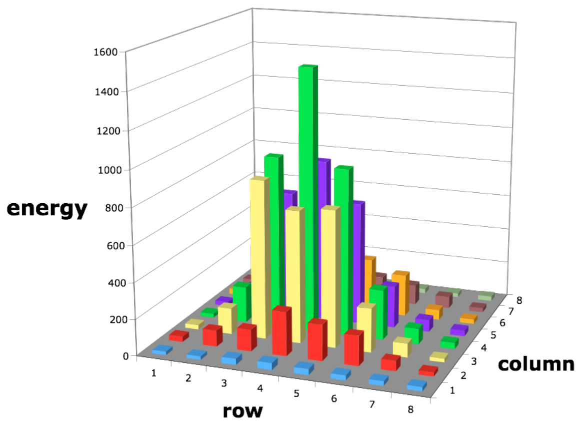

Fig. 8. A representative event, showing the response of an 8x8 PMT array.

The method we have developed for increasing the resolution of the photon sensors is known as Statistics Based Positioning (SBP) [11]. SBP is able to improve the overall detection characteristics of a continuous crystal detector. In an SBP system, each sensor array requires an initial characterization step. This involves positioning a source in the field of view at a known X-Y location, collecting data from each of the 64 electronic sensors in the array, and saving the light response characteristics for that X-Y location in a table for future reference. The two statistical characteristics that are stored are the mean (μ) and variance (σ^2) of the light distribution function of each sensor for a given X-Y location. In other words, the same location of the crystal is hit with photons many times and the mean and variance of the energy deposited on each of the 64 sensors for all photon interactions is calculated and stored. This is done for 127x127 locations. The result of this characterization is a 127x127 table that has 64 means and variances (one for each sensor in the 8x8 array) in each location. In execution mode, data collected from an unknown location is compared with the previously collected data table using (eqn. 1), and the coordinates of the characterization data that has the maximum value for (eqn. 1) are the position of the unknown source. Using this method, position resolution much finer than the spacing of the individual detectors within the 8x8 arrays is achieved.

To calculate (eqn. 1) in hardware, the following modifications were made for the equation inside the summation

$$R = \frac{(E-\mu)^2}{2\cdot\sigma^2} + \ln(\sigma) = \left[(E-\mu)\cdot\frac{1}{\sqrt{2\cdot\sigma}}\right]^2 + \ln(\sigma)$$

let : $\quad A = \mu \qquad B = \dfrac{1}{\sqrt{2\cdot\sigma}} \qquad C = \ln(\sigma)$

$$R = [(E - A)\cdot B]^2 + C$$

In this refactoring, the values for A, B and C are precalculated and stored in memory.

The search of all 127x127 positions is too much processing to complete in the time required for the desired count rate of the scanner. Fortunately, the solution space is convex and thus lends itself to a smarter search. The algorithm that we implemented is a modified hierarchal search [12].

The hierarchal search starts by sampling 9 points at multiple locations (see stage 1 in Fig 9), uniformly scattered across the whole solution space. The location with the highest value (upper right location of stage 1 in Fig 9) is assumed to be closer to the final solution than any of the other eight points with lower values. The highest valued location is then used as the center of a new search space (stage 2 in Fig 9) that is ¼ the size of the previous space. When the size of the solution space is reduced to one, the solution has been reached.

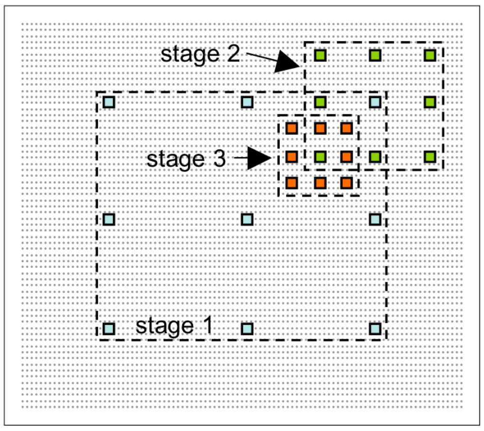

Fig. 9. Three iterations of a 3x3 search on the solution space. Note that this example only shows the first three of seven stages for illustration clarity.

The regularity of this search allows each level of the search to consist of an almost identical process. For the 3x3 search shown in Fig. 9, nine sets of characterization data table values are needed for the first stage, one for each position to be tested. The next iteration will consist of potential sample positions at one-half the spacing, with sets of points centered on each of the previous stage's points. This results in each stage's data table having approximately four times the number of testable locations as the previous stage, but still only a fraction of the overall table, until the last stage is reached. The last stage is the only stage that needs access to the entire 127x127 table.

Structuring the search in this way allows points tested at each iteration to be independent of calculations currently underway in the other iterations. This allows pipelining of the calculations with different events underway at each of the stages simultaneously. A second advantage of this method is that the storage requirement for each stage's data table is reduced for each iteration prior to the last stage.

A. FPGA Implementation

This algorithm was also implemented on the StratixII DSP board [12]. Because the memory available in the prototype is not sufficient for a seven-stage system, only five stages could be created and tested. Each stage requires 192 clock cycles. This represents a solution for each of 9 X-Y locations at each stage, plus 3 cycles for the final calculation of the coordinates of the minimum value and transferring the results to the next stage. Although each event will take 192 clocks at each of seven stages, for a total of 1344 clock cycles, as each stage completes its calculation it is immediately available for the next sequential event. This results in 7 events being solved simultaneously, with each event at a different stage of its solution. The implementation utilizes only 3.6% of the StratixII S180 and 75.9% of the memory (first five stages).

B. Ambric Implementation

Again, the localization algorithm was implemented in the Ambric MPPA as an application experiment. Obviously the memory requirements of this application will be the largest implementation problem. This algorithm requires far more memory than is available on the Ambric AM2045, so for a full implementation an external memory would be required. Just as we did with the FPGA implementation, we investigated the application with the maximum number of stages that could fit on the device.

Since the memory requirements of this algorithm were so extensive, this was the focus of our investigation. While it is possible for each processor to read and write to any RU in the MPPA, it was determined that the processors would not have the required bandwidth to make this work. So like the timing algorithm, the memories have to be manually partitioned over many RUs. Two alternative methods for partitioning the table over many processors were investigated. The first method, called "By Region," was to break up the data by regions; each RU would store the response for one of the 127x127 array locations. Stage 1 requires the storage of nine locations, and therefore uses nine RUs and nine processors to compute the correlation. Stage 2 needs to store 49 locations. As can be seen in Table III the number of processors needed by stage 3 is 283, and the fourth stage alone would need 961 processors, far exceeding the number of processors on the Ambric AM2045.

Table III:
Resource requirements per stage for "By Region".

Stage	Row/Column Length	Full Table Size	Number of Processors	Memory per Processor (words)
1	3	1728	9	192
2	7	9408	49	192
3	15	43200	225	192
4	31	184512	961	192
5	63	762048	3969	192
6	127	3096768	16129	192

Notice that each RU for each processor in this scheme only requires 192 16-bit words (384 bytes). Each RU has 4KB of memory, so they are only being partially filled. Given this under-utilization of memory a second method was investigated in order to utilize more of the memory. The second method of data division is called "By Sensor". This method splits the data from the 64 sensors for each of the 127x127 locations needed for a given stage. Because there are 64 sensors, each RU could have some even portion (1/2, 1/4, 1/8, 1/16, 1/32) of the total table needed for a given stage. For example, in stage 2, 49 of the 16129 (127x127) locations are needed. Given the RUs storage capacity, it turns out that each RU can store data for 8 of the 64 sensors for all 49 needed locations in each RU. This means that 8 processor/RUs are needed to compute stage 2. Each processor computes the partial correlation for all locations based on the portion of the 8x8 sensor it has. Essentially, each processor computes only a subset of the sums in (1). The partial correlation for a given location is then sent to another processor, along with the other partial correlations, to tabulate the final sum as shown in Fig 10. Table IV shows the maximum number of sensors per processor. The actual implementation is the largest even portion that will fit in a given RU.

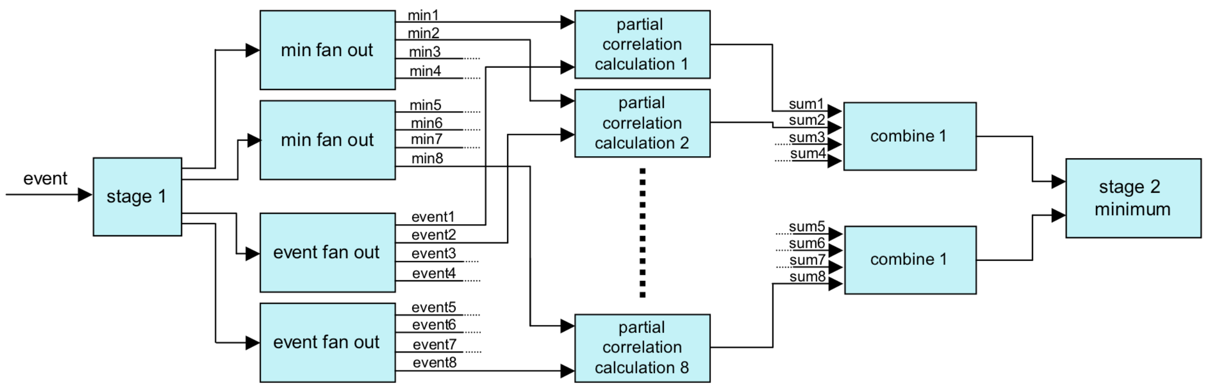

Fig. 10. Block diagram of the Ambric implementation of event "By Sensor" event localization.

Table IV
Maximum resource requirements for the "By Sensor" memory partition scheme.

Stage	Row/Column Length	Full Table Size	Number of Processors required	Number of Sensors/ Processor
1	3	1728	1	74
2	7	9408	5	13
3	15	43200	22	2
4	31	184512	93	0.69
5	63	762048	382	0.17
6	127	3096768	1549	0.04

Notice that even though this memory partition scheme reduces the number of processors needed and allows an extra stage to be stored on-chip, only four of the six stages can fit on the device. For comparison, an equivalent number of stages can also fit on a Stratix S30 (also 130nm process chip).

C. Discussion

Just like the memory partition in the timing algorithm, this application required a substantial amount of work to get a larger memory to work well on the Ambric MPPA. For both FPGAs and MPPAs, the last couple of stages need to be stored off chip, but the memory setup for on-chip memory was much easier for the FPGA. Also, since so many processors on the MPPA are needed to distribute the memory, there would be only a couple of processors available to perform the correlation calculations for the last stage. On the other hand, the FPGA has plenty of unused logic available.

VI. CONCLUSION

Ambric was designed to be a massively parallel computing platform like FPGAs. Unlike FPGAs though, the Ambric MPPA was designed to achieve the parallelism with lower power and more efficient design cycles. Previous academic research indicates that these two goals were accomplished. However, we have shown in this paper that some of the choices made to reach those goals have a large implication on certain applications. Both the Ambric MPPA and FPGAs are best suited for streaming applications, and the two applications that we covered here are certainly streaming applications. These applications however also have requirements that made them difficult to implement on the Ambric MPPA. Specifically, we showed how the inability to create a real-time clock and the lack of larger memories made these applications less suited to this MPPA. We also demonstrated how achieving speedup for timing critical objects in an MPPA requires manual partitioning. We discussed some possible software/compiler tricks that could solve these problems. We also discussed how partitioning to increased throughput also leads to low code density. This phenomenon however seems to be inherent in MPPAs and is one of the large differences between FPGAs and MPPAs.

ACKNOWLEDGMENT

This work is supported in part by Altera, Ambric, NIBIB R21/R33-EB0001563, NIH grant EB002117, NSF, and Zecotek.

REFERENCES

[1] ITRS: 2007 Edition - Design. http://www.itrs.net/Links/2007ITRS/2007 Chapters 2007 Design.pdf

[2] www.ambric.com

[3] B. Hutchings, B. Nelson, S. West, R. Curtis, "Optical Flow on the Massively Parallel Processor Array (MPPA)," *International Symp. on Field-Programmable Custom Computing Machines (FCCM),* 2009.

[4] P. Top, M Gokhale, "Application Experiments: MPPA and FPGA," *International Symp. on Field-Programmable Custom Computing Machines (FCCM),* 2009.

[5] C. Hu, et al., "Design of a 64-channel Digital High Frequency Linear Array Ultrasound Imaging Beamformer on a Massively Parallel Processor Array," *IEEE International Ultrasonic Symp. Proc.,* 2008, pp. 1266-1269.

[6] M. Butts, A.M. Jones, P. Wasson, "A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing," *International Symp. on Field-Programmable Custom Computing Machines (FCCM),* 2007, pp. 55-64.

[7] M. Butts, "Synchronization Through Communication in a Massively Prallel Processor Array," *IEEE Micro,* pp. 32-40.

[8] T.K. Lewellen *et al*., "Design of a Second Generation FireWire Based Data Acquisition System for Small Animal PET Scanners," *IEEE Nuclear Science Symp. Conf. Record*, 2008, pp. 5023-5028.

[9] M.D. Haselman, S. Hauck, T.K. Lewellen, and R.S. Miyaoka, "Simulation of Algorithms for Pulse Timing in FPGAs," *IEEE Nuclear Science Symp. Conf. Record*, 2007, pp. 3161-3165.

[10] M. Haselman, et al., "FPGA-Based Front-End Electronics for Positron Emission Tomography," *ACM/SIGDA International Symp. on Field-Programmable Gate Arrays*, 2009, pp. 93-102.

[11] J. Joung et al., "cMiCE:a high resolution animal PET using continuous LSO with a statistics based positioning scheme," *Nuclear Science Symposium Conference Record, 2001 IEEE*, 2001, vol.2, pp. 1137-1143.

[12] D. DeWitt, "An FPGA Implementation of Statistical Based Positioning for Positron Emission Tomography," *Masters of Science in Electrical Engineering thesis*, University of Washington, 2008.

[13] D.B. Thomas, L. Howes, W. Luk, "A comparison of CPU's, GPU's, FPGA's, and massively parallel processor arrays for random number generation," *ACM/SIGDA International Symp. on Field-Programmable Gate Arrays*, 2009, pp. 63-72.