Multi-Kernel Macah Support and Applications

Adam Knight

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2010

Program Authorized to Offer Degree:
Department of Electrical Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a master's thesis by

Adam Knight

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____
Scott A. Hauck

_____
Carl Ebeling

Date:_____

University of Washington

Abstract

Multi-Kernel Macah

Adam Knight

The Mosaic tool chain is a set of tools used to program applications onto coarse grained reconfigurable architecture (CGRAs) based coprocessor accelerators. The front end of this is a C-based language called Macah, which is used to program applications and benchmarks. Prior to the work described here only single sequential kernels of computation could be effectively programmed using this language. I have expanded on the libraries already present in the language by adding an additional API to introduce multi-kernel support. The improvements are shown to improve Macah by allowing multi-kernel development, as well as support syntax changes that improve the ease of development for most applications. In creation of this API I also implemented a version of a Positron Emission Tomography (PET) event detection algorithm in both single and multi-kernel variants for testing and benchmarking multi-kernel Macah.

Table of Contents

List of Figures

Index of Tables

1 Introduction

There has been a large push to develop alternate models of computing beyond the traditional sequential processor. Coprocessor architectures with coarse grained reconfigurable arrays (CGRA) and normal sequential processors working in parallel are one such type. However, programing for these hybrid devices is a difficult problem because the CGRAs used vary widely and most tools and languages are specific to a single type of coprocessor. The Macah language, along with the Mosaic tool chain, attempts to create a system that will allow programs written in a more generalized language to be mapped to a variety of CGRA accelerator architectures.

My work has been in Macah application development and in support of the Macah portion of the tool chain. In this capacity I have implemented a new API for Macah that allows programs with multiple kernels, which are blocks of sequentially written code intended to be accelerated on the CGRA. By allowing multiple kernels, developers have more tools at their disposal for describing and coding complex streaming applications. This additional API also supports a major change in Macah syntax intended to promote easier coding of all Macah applications. In the development of the API I also implemented an initial test application based on Positron Emission Tomography event detection.

2 Hybrid Micro-Parallel System Model

The Hybrid Micro-Parallel (HMP) architecture model proposed in [1] is composed of two different processing components operating in parallel. It is an extension of the normal

sequential von Neumann machine and describes a variety of computers. This abstraction from actual architectures to the model allows the programmer writing applications for these systems to be presented with the essential features and constraints without being burdened with specific details. It also allows for the development of programs that are capable of running on multiple different implementations of the HMP type model architecture. The main components of the HMP architecture are shown in Figure 1.

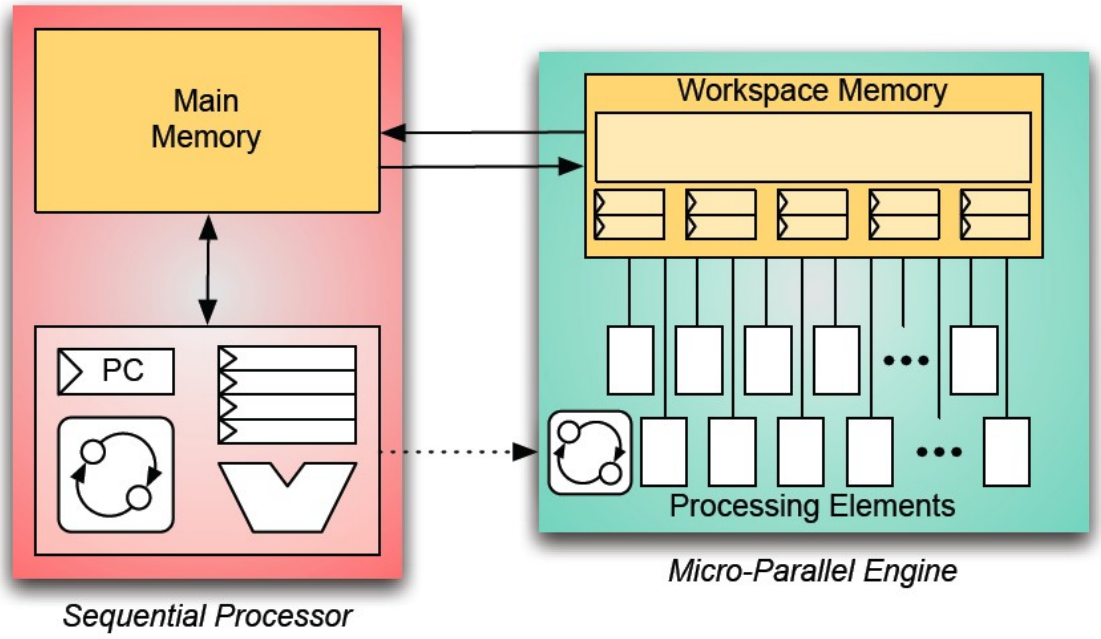

Figure 1: Hybrid Micro-Parallel model. [1]

The micro-parallel engine (MPE) consists of a workspace memory, processing elements, and control resources. The specific quantity and type of processing elements (PE) vary depending on the specific MPE implemented. The workspace memory itself is an abstraction representing the combined registers and memory structures distributed throughout the MPE,

and as with the PE the specific distribution depends on the implementation. However, this memory is always assumed to be smaller then the main memory. These two memories communicate through a bidirectional channel, which acts as the primary method of communication between the processor and MPE. Its bandwidth is constrained in the same way that the main memory to sequential processor bandwidth is constrained, and is often designed to support high bandwidth for specific memory access patterns.

The system has a single thread of control that alternates between the sequential processor and the MPE. The parts of the program with low parallelism are executed on the sequential processor, while computationally intensive parallelizable portions of the code are run on the MPE. Due to the costs associated with switching between the two modes, including initialization and configuration, only sections of code that require a significant amount of computation run on the MPE.

An example of this type of architecture is an FPGA coprocessor system where the FPGA acts as the MPE alongside a standard sequential processor. In this case the FPGA's LUTs, registers, and block RAM act as the workspace memory. Commonly used Graphics Processing Units (GPUs), seen in most consumer computers, are yet another example. The GPU is specifically designed to perform graphics operations quickly and in parallel, relieving the burden from the CPU and performing them much more quickly then the CPU would be able to alone.

3 Mosaic

The purpose of Mosaic [3] is to take programs written for these HMP type architectures and map them to specific architectures, which can then be simulated. This allows for the exploration of several different areas of interest including application development, CAD tools, and coarse grained architectures. The full tool chain is illustrated in Figure 2.

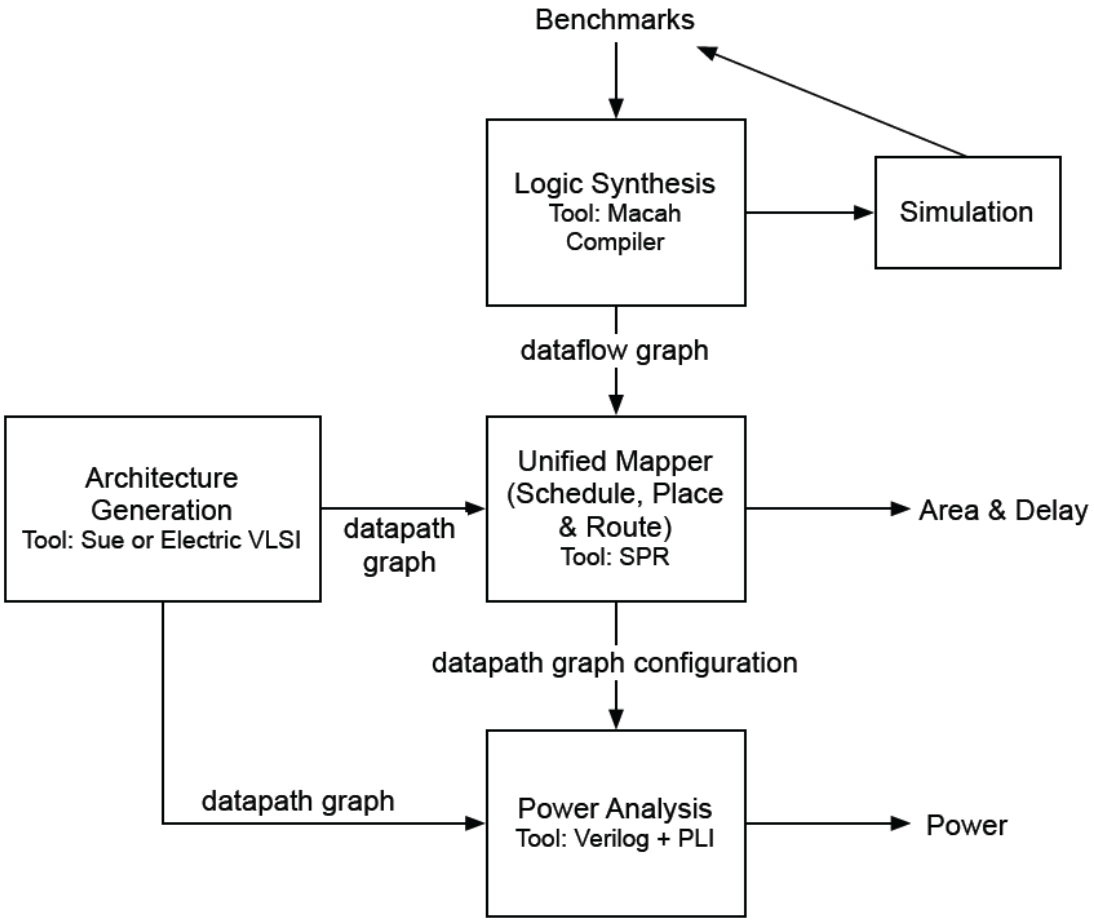

Figure 2: Mosaic Tool Chain. [3]

Programs are written in a C derived language called Macah and its complier transforms the application code in to a control data flow graph (CDFG). The CDFG contains all of the control and data dependencies needed to execute the accelerated code. This CDFG, along

with a data path graph (DPG) generated by an architecture generator tool, is fed into a unified schedule, place, and route tool (SPR). The SPR tool generates a configuration of the DPG and provides a measurement of the application's initiation interval (II), and the dependency loop in the CDFG that is limiting the II. These two pieces of data are useful for application optimizations, but are architecture-specific. To get around the architecture specific nature of SPR's profiling results there is an additional mode useful for application programmers. SPR Analyze performs similarly to normal SPR but works without an input DPG from the architecture generator. The resulting II is a lower bound on the application's II based primarily on the dependencies between instructions and not the resource limitations of a particular architecture.

4 Macah

Macah [2] is a language based on C, but unlike that language it forces the programmer to think about some of the abstract architectural features present in the HMP model. It provides the functionality of C along with threads and extensions designed to help program HMP coprocessor accelerators.

Programming for accelerators is generally difficult because they require the use of unconventional languages or compilers which are often platform specific. They are especially weak at handling more complex input-dependent control and data access patterns. Macah and its complier make it easier to write complex platform-agnostic applications for accelerators by balancing the strengths of the programmer and the compiler.

The accelerators that the language targets take advantage of a high degree of parallelism to achieve substantial gains over traditional sequential processors. To do this the language must be able to allocate local data structures to workspace memory, use the higher latency bandwidth to main memory efficiently, and pipeline loops so that operations from different loop iterations can execute concurrently. The main additional language features in Macah are streams, kernel blocks, FOR loops, and shiftable arrays.

4.1 Streams

Streams are abstractly just first in first out (FIFO) unidirectional channels that provide communication and synchronization between threads of an application. As a rule all communication between processes within the accelerator is done through streams. There are two uses for streams in Macah programs. Memory access streams provide access to main memory for accelerated threads of an application. These streams are only attached to a single kernel and either send data from main memory through the stream, or take data off a stream and write it back to memory. The other type of stream is used within the MPE and connects two asynchronous accelerated threads and is used both for communication and synchronization between them. These threads are asynchronous in that they perform their operations independently of each other and can only affect each other with either the presence of or lack of data on a connecting stream. By definition a Macah stream must have both a sender and receiver thread defined before it is called or the system will exit with an error. This is to prevent streams not being properly formed prior to use.

The streams are created and manipulated with calls to functions in the PStreams library. To

create a stream a create function called pstream_create is used as shown below:

```
stream = pstream_create(sizeof(int));
```

The create function requires the size of the data type being sent over the stream. A single stream can only support a single data type and if two different types are needed, doubles and ints for example, then either two streams are needed or you convert the data types before and after the streams.

Streams by default use blocking sends and receives, which are sufficient for most applications, and which serve to synchronize the threads as well as provide communication. The send and receive commands are shown below:

```
Send: stream <! data;
Receive: var <? stream;
```

There also are non-blocking send and receive functions available which require an additional argument that stores the success or failure of the operation. These operations are used less frequently since in many applications there is no useful work to be performed without data. Also, the current assumption for streams in Macah is that they are large enough to accommodate the application requirements, but currently no specific capacity is specified.

4.2 Kernel Blocks

Kernel blocks are denoted by the keyword “kernel” and designate what code should run on the accelerator. Macah will attempt to generate an accelerated implementation of this code for mapping to an accelerator. Code outside of these blocks will be translated to standard C to

be run on the accompanying sequential processor.

In order to efficiently use the PEs of the accelerator the kernel code must be pipelined. This means that different parts of adjacent loop iterations need to be executing simultaneously. In order to accommodate this, order of execution rules are relaxed and stream sends and receives are allowed to execute in a different order than explicitly stated in the code. This can cause deadlocks and requires that the programmer take proper precautions to ensure that data sent out of a kernel does not require data to be received later in the same kernel. A simple example of this would where a kernel is both writing and reading to memory. In this case a deadlock would occur if the memory is expecting a number of sequential writes before a read and the pipelined kernel performs an out of sequence read and write concurrently causing the pipeline to stall. The onus is on the programmer to prevent these kinds of deadlocks.

There are some restrictions to the code used in a kernel block. First, all function calls inside a kernel must be inlineable. Recursive functions, calls through function pointers, and special system calls are not supported in kernel blocks. Second, accesses through pointers are generally not allowed. There are exceptions for locally declared arrays which are different from pointers, and for certain limited cases of pointer dereferencing where it can be inlined and simplified to remove the pointer manipulation. Last, all data structures used in a kernel must be statically allocated in the workspace memory. When a variable is accessed both inside and outside the kernel, the system can copy it into workspace memory and then back out after finishing.

4.3 FOR Loops

FOR loops are a Macah extension of traditional *for* loops. The purpose of these loops is to allow a programer to specify a *for* loop that will be completely unrolled at compile time. These unrolled loops can then be pipelined or parallelized to improve performance. Loop optimizations have a big influence on performance of an application and can have a large impact on memory and bandwidth requirements. Because of this, *FOR* loops were created in order to provide a tool to the programer for determining how the application is parallelized.

4.4 Shiftable Arrays

Shiftable arrays are arrays that support an additional shift operation to the right or left. The shift syntax is “array <<= N” for left shifts and “array >>= N” for right shifts. Execution of a shift will “move” the contents of the array N units in the correct direction. It should be noted that data shifted off the array is lost, and new data shifted on is considered garbage until replaced by the user. These structures are useful in many applications such as FIR filters where the algorithm contains a similar construct.

4.5 Architecture-Dependent Pseudoconstants

One of the goals of Macah is to be as platform agnostic as possible. Architecture-Dependent Pseudoconstants (ADPC), also known as tuning knobs, help achieve this. ADPCs are constants present in the code that can be modified by the compiler in order to adapt an application to a particular architecture. They are most typically used to determine array sizes or loop limits, and are usually chosen by the compiler from a range of values. One example of this type of constant would be in an image processing application such as convolution where a portion of the image must be stored on the workspace memory to process on while

the full image may be stored in main memory. In this case the ADPCs can be adjusted by the compiler to best adapt the size of the workspace image buffer based on the target's resources.

4.6 Using the Macah Compiler

Macah has four modes of compilation, each of which is used for different purposes. The first mode is sim which is used to compile the entire application to C code. Any non-kernel code is always converted to C with calls to the special libraries as needed for Macah-specific structures. This same technique can be used to compile the entire Macah program, including the kernel portions. Under this mode the compiler produces an executable file that can be run to simulate the application. This mode is the primary tool for application programming and basic debugging, because it allows the programmer to check the basic functionality of their design.

The next mode is fsim, which like sim converts the entire program into C code, but performs additional optimizations. For the application developer this mode is important because it generates additional profiling information about the application. After compiling and running the resulting simulation, a file is produced that identifies the number of loop iterations the kernel performed.

The last two are pre and post SPR. Pre-SPR mode produces a simulation where simplified data flow versions of the kernels are generated in Verilog. This mode serves primarily to act as a tool chain debugger, but can also be used for basic application testing. Post-SPR mode produces the DFG and necessary files to continue to the SPR phase of the tool chain where mapping and additional profiling take place.

5 Multi-Kernel Macah

The addition of multi-kernel is another step in the evolution of the Macah language that brings two main benefits to the language: official support for multiple asynchronous kernel blocks and increased ease of programing. There are a number of reasons why introducing multi-kernel applications to both Macah and the Mosaic tool chain are beneficial. In many signal processing and data streaming applications the algorithm will be composed into distinct conceptual blocks such as filters or accumulators. Multi-kernel Macah gives the programer the ability to construct these blocks as separate kernels. Maintaining the conceptual separation in the actual code makes it potentially easier for the programmer to program and relate back to the original algorithm. There are also potentially applications where the data flow is complicated enough so that a performance gain can be obtained by manually separating the code into distinct kernel blocks. Separating an algorithm into different kernel blocks also allows the tool chain to distribute the accelerator's resources in uneven ways to maximize performance. In general, addition of multi-kernel gives the programer additional options specifying the structure of an application, and it gives the tools additional information when mapping to a specific architecture.

Writing applications with multiple kernels was possible with the previous version of Macah, but it was cumbersome and required the use of Macah library functions that were no longer officially supported. All benchmarks and applications prior to the improvements discussed in this section had been single kernel. The one exception is the Positron Emission Tomography application discussed in section 7. I wrote this application originally in the old style as

preparation for designing the new API improvements that introduced official support for these applications.

Ease of programming is important for Macah because it has a major impact on how long it takes to get new programers into the system and generating useful code. The old version of Macah required a fair amount of knowledge of the Macah-specific libraries and their functions in order to effectively program an application. It also required knowledge of the pthreads library in order to handle the different threads needed for a single application. Handling threads and streams, in particular, is cumbersome and prone to causing both compile and simulation errors that can be difficult to track down. Figure 3 shows a simple example multi-kernel program written in the old style. The highlighted regions are lines of code dedicated to stream and thread manipulation. This simple two kernel design requires a large amount of code overhead just to manage the threads and streams in this simple example. Changes to the language that accompanied the inclusion of multi-kernel support help to alleviate the difficulties of writing both single kernel and multi-kernel applications. These changes remove much of the highlighted code out of the sight of the programmer, and this management is taken care of by the system. The new API also stores more information about the design which can be used for a variety of purposes including profiling.

```
void readFile (intstream strm, void *args) {
        int i;
        for(i=0;i<dataSize;i++){
                strm <! data[i];
        }
}
intstream dStrm = spawn_in_stream_clos (readFile, NULL, sizeof(unsigned int));
intstream * const pdStrm = &dStrm;
intstream kernelStrm  = pstream_create(sizeof(unsigned int));
intstream * const pkernelStrm  = &kernelStrm;

void Kernel1 (){
        pstream_change_sender(*pkernelStrm, pthread_self());
        pstream_change_recver(*pdStrm, pthread_self());

        kernel{
                for(i=0;i<dataSize;i++){
                        int temp <? *pdStrm;
                        *pkernelStrm <! temp;
                }
        }
}
void Kernel2 (){
        pstream_change_recver(*pkernelStrm, pthread_self());
                kernel{
                        for(i=0;i<dataSize;i++){
                                int temp <? *pkernelStrm;
                        }
                }
}
pthread_t thread1,thread2;
pthread_create_clos( &thread1, NULL, thesholdKernel, NULL);
pthread_create_clos( &thread2, NULL, normKernel, NULL);

pthread_join( thread1, NULL);
pthread_join( thread2, NULL);
```

Figure 3: Multi-Kernel code in old style.

Another benefit of multi-kernel applications is the ability to allocate a different amount of resources to each individual asynchronous kernel. Splitting up the program into kernels

provides additional information to the tools which can be used during floor planning to determine the amount of resources to assign to each kernel. This affects performance since the functional II of these kernels when mapped on an accelerator are based on resource allocation. By also taking into consideration other performance metrics generated by the tools such as number of iterations and approximate data production and consumption rates it is possible to effectively floor plan the kernels for balanced performance. A simple example, shown in Figure 4, is composed of two sequential kernels where the first kernel (red) consumes data at four times the rate it produces data for the second kernel (blue). Figure 5 shows the resource allocation for this example were the first kernel is assigned 80% of the available PEs and the second kernel is assigned only 20%. In this example since the first kernel runs at a faster rate and if both kernels are assigned equal resources then the second is often starved for data and idle. By assigning more PEs to the first then the second the production and consumption rates of the two kernels can be balanced.

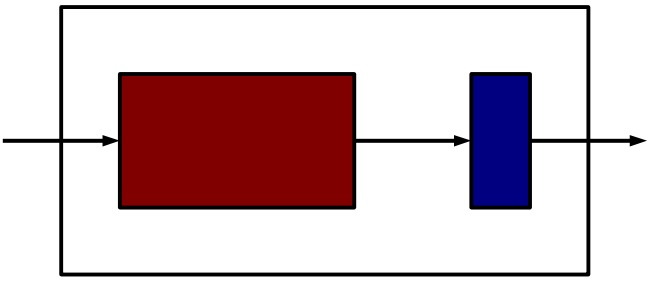

Figure 4: Floor plan example data flow structure. First kernel in red and the second in blue.

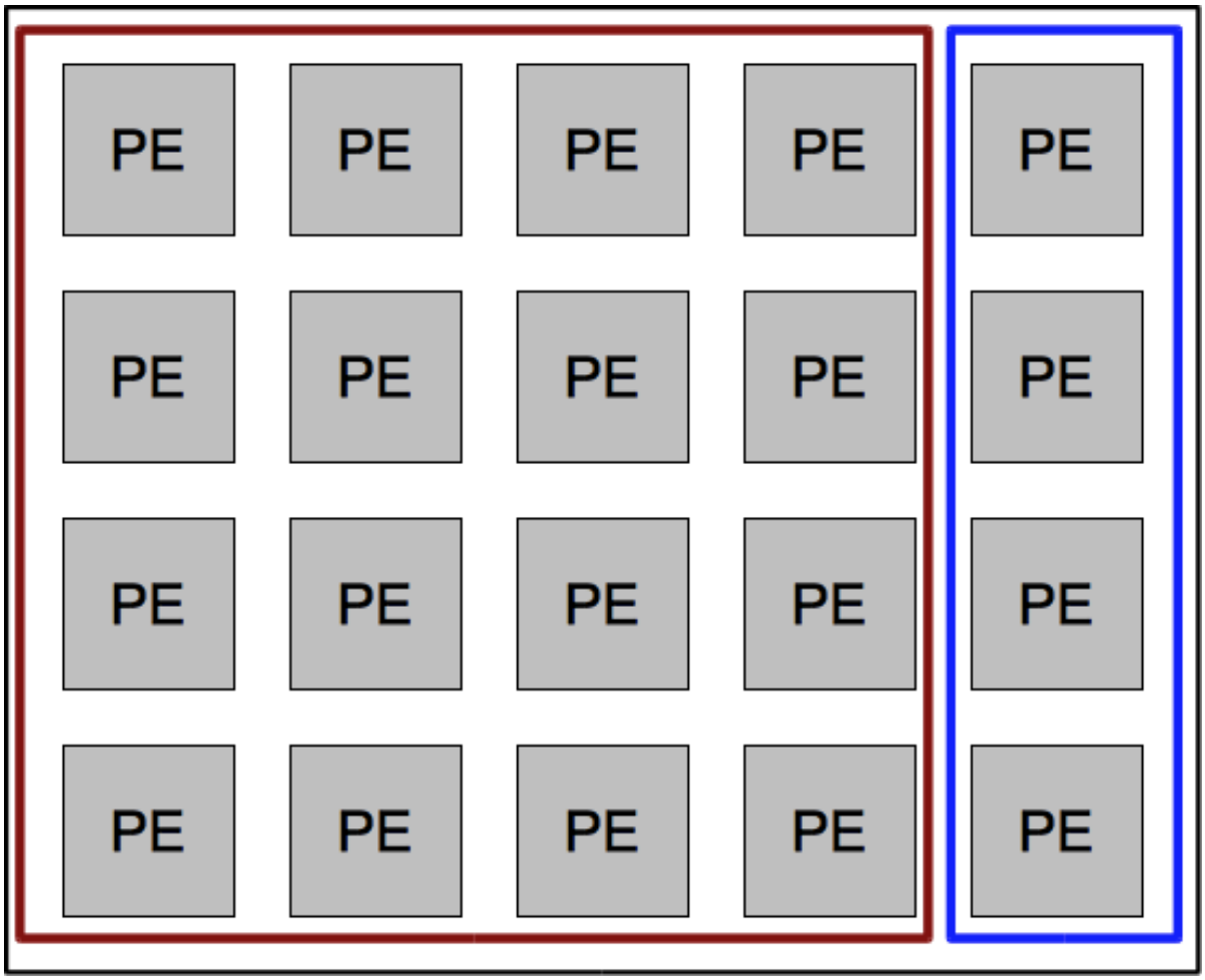

Figure 5: Possible resource allocation for a two kernel application. Kernel 1 is allocated to the elements within the red box and kernel 2 within the blue box.

5.1 Multi-Kernel API

The multi-kernel API introduces a new hierarchy to Macah applications. The structure is shown in Figure 6. The configuration stores lists of the tasks and streams associated with this configuration. Tasks represent the blocks of code in the algorithm and pstreams are the communication channels between tasks. I wrote the multi-kernel API and then Ben Ylvisaker [1][2] implemented complier changes updating the Macah syntax, introducing syntax changes and additional keywords to mask the API function calls.

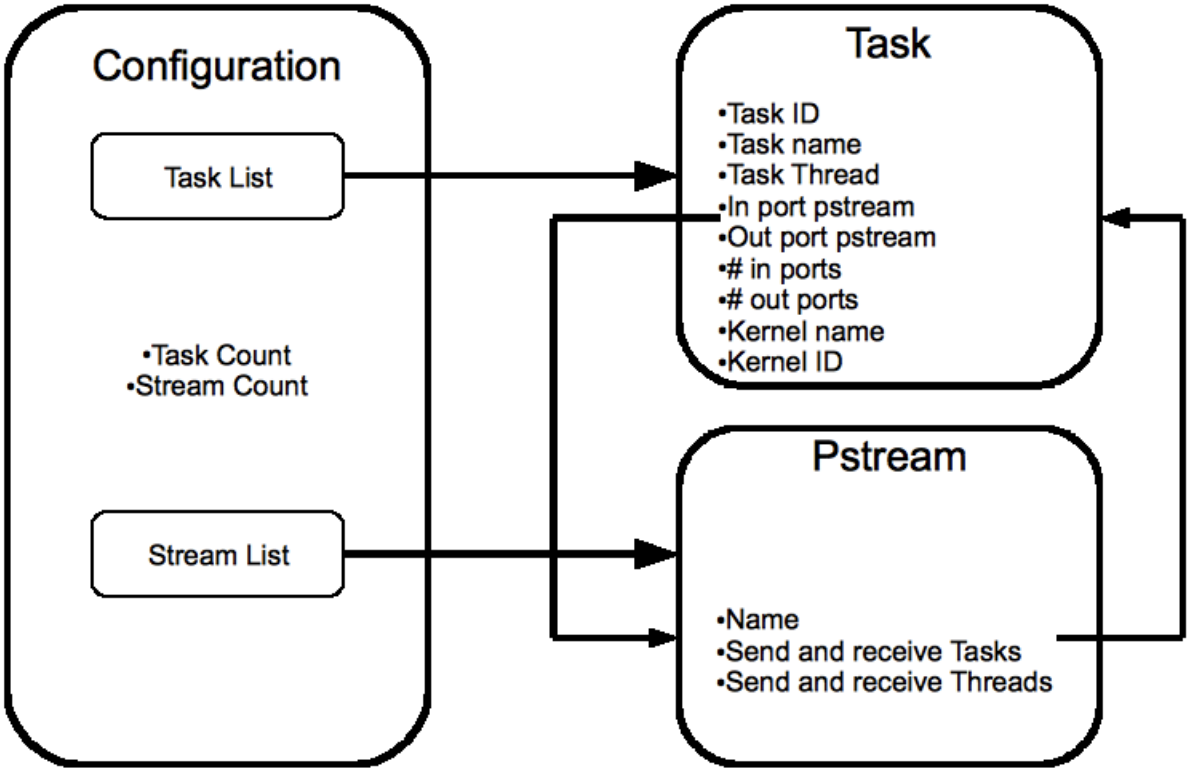

Figure 6: Macah data structure hierarchy.

5.1.1 Configuration Block

The configuration is the highest level of a Macah program. Its struct stores the complete global lists containing pointers to all tasks and streams in play in the configuration. A pointer to this configuration is global and therefore accessible by any function. By definition there can be only one configuration per Macah program. This makes logical sense because the configuration represents the complete structure of the program; everything about the implementation is contained within.

5.1.2 Tasks

Tasks are everything in a Macah program; they contain all of the code executing on both the

sequential processor and the accelerator. For this reason tasks may or may not have kernels within them. Also, by definition a task can only have one kernel inside it because each task is abstractly its own thread. This is exactly how Macah programs that are compiled to C and simulated, each task is given its own thread and then run at once.

The task struct contains the parameters listed below:

- ID
- Name
- Pointer to function
- Task's thread
- List of pointers to input and output streams
- Number of input and output streams in list

The ID and name fields are used for visualizations of the configuration's structure and for the Verilog module names used later in the Mosaic tool chain. The pointer to function is a pointer to a data structure generated for the code inside the task by the compiler. There is also a pointer to this task's assigned thread. Last there are lists of pointers to the input and output streams associated with this task as the number of streams for each.

5.1.3 Pstreams

The pstreams data structure is part of the pstreams library that existed prior to my work on the new API. In order to incorporate the new features of the API several new fields were added to the Pstreams struct. Pointers to the sender and receiver tasks were added, as well as a name field for the stream.

5.1.4 Data Structure Redundancy

From Figure 6 it is clear that there exists a fair amount of redundancy in the data structures for the API. There are two reasons for this. The first reason for the redundancy is that some of it is necessary for legacy reasons. Many library functions and benchmarks developed throughout the life of Macah and Mosaic would fail without the redundancy. One major goal was to avoid the disruption of other researchers using the Macah libraries and benchmarks, and to avoid the need to test and correct older code. Also, until the programs written in the new style can progress through the entire tool chain, the legacy code must be maintained to ensure that tool chain development can continue with functional benchmarks. The second reason is that it allows the functions in the API to quickly transverse the data structures regardless of their starting position. For example if a piece of code possesses a handle on a pstream and needs to get to the stream's task to check that the threads stored in the pstream struct are correct. In this situation without a pointer to the task within the stream it would be necessary to search all tasks' stream lists, starting from the configuration task list, in order to match the stream to a task. This is extremely cumbersome and slow, and since the memory cost of the pointers are insignificant there is no reason not to include the redundant information.

5.1.5 API Functions

The main API functions are designed to generate the complete configuration, including the data structures discussed in previous sections, and then run the configuration. Most of the non-utility functions are not designed for direct use by programmers. The compiler automatically calls the functions as necessary. Use of the functions directly without being

very careful would likely cause conflicts with the compiler. The basic steps are as follows:

1. Create and initialize the configuration
2. Create tasks
3. Add all tasks and streams to configuration
4. Connect tasks and streams correctly
5. Run configuration

The creation and initialization of the configuration simply creates a configuration struct with default values and sets the global pointer to this struct so that all other functions can use it. The presence of this pointer also determines if the system is in multi-kernel or regular mode. “Regular” mode is the previous version of Macah. This pointer check allows the complier to avoid using new functions when running legacy code.

The task creation function takes two arguments. The important one is a pointer to a data structure present in the libraries called a closure that contains all of the information about the function representing the code within a task. This closure is used by other Macah library functions to run the code. The other parameter is a name field used for a variety of bookkeeping purposes.

Both tasks and streams have their own functions for adding them to the global lists in the configuration. The configuration's two main functions serve to connect streams to tasks, one for in ports and one for out ports. These functions use the task ID and stream pointer to properly add them to the correct task list and configure the other necessary parameters.

Running the configuration first requires iterating through all tasks and creating threads and then calling the library function “pthread_create_clos” to start the task. This function is used directly in the previous versions of Macah, but is now hidden within the new API. It takes both a thread and the task's function pointer, which points to a closure, to start the task. After all tasks have begun, the configuration stream list is processed in order to add the appropriate threads to the pstreams struct's sender and receiver parameters. We needed to wait until the thread had been created in the previous step before we could finish registering the streams. Also by waiting, the streams can be registered together with a single function, without relying on unsupported functions that use a discontinued set of mutex locks to register one thread at a time. Last we wait for all the threads to complete before exiting.

A summary of the important functions for the multi-kernel API can be found in Table 1.

Function	Purpose
init_configuration	Initializes the configuration struct. Sets global pointer.
create_task2	Creates a task with default parameters with a pointer to the function it executes.
add_task_to_config	Adds a task to the configuration task list. Creates list if empty, and expands if full.
add_stream_to_configuration	Adds a stream to the configuration stream list. Creates list if empty, and expands if full.
attach_stream_to_task_in_port	Adds a stream to the task in port stream list. Creates list if empty, and expands if full.
attach_stream_to_task_out_port	Adds a stream to the task out port stream list. Creates list if empty, and expands if full.
run_macah_configuration	Runs a Macah configuration generated by the other functions. Creates and starts threads and makes sure that streams are properly registered to the correct threads. Waits for thread completion before exiting. Also performs any required post processing or data gathering after completion.
print_macah_config_dot	Utility: Produces a graph visualization (.dot format) of the configuration with labels to show key features(tasks, kernels, names, etc.).

Table 1: Summary of important API functions.

5.1.6 Multi-Kernel Syntax and Application

An example of the new syntax introduced to make use of the API is shown in Figure 7. The “configure_tasks {” keyword denotes the beginning of the configuration block. Task declaration begins with the “task” keyword followed by the task name, and a list of all input and output streams. Streams are specified as inputs or outputs by the “in_port” and “out_port” keywords.

In order to see the benefit of the new multi-kernel Macah on ease of programming compare the code in Figure 7 with that in Figure 3. Both sets of code perform the same simple function and both are highlighted to show code devoted to thread and stream manipulation. The code in Figure 7 is significantly shorter, as well as being simpler and easier to understand. The programmer also has to do far less pointer manipulation to achieve a basic functional program. As an added benefit it is also easier to visualize the structure of the application directly from the code than it was previously. These benefits should make the time required to get a new user up to speed and producing functional code far shorter than the previous version of the Macah language.

```
configure_tasks {
        intstream dStrm = pstream_create2(sizeof(int));
        intstream kernelStrm = pstream_create2(sizeof(int));
        intstream outStrm = pstream_create2(sizeof(int));
        task reader(out_port dstrm) {
                int i;
                for(i=0;i<dataSize;i++){
                        dstrm <! data[i];
                }
        }

        task t1 (in_port dstrm, out_port kernelStrm){
                kernel{
                        for(i=0;i<dataSize;i++){
                                int temp <? dStrm;
                                kernelStrm <! temp;
                        }
                }
        }
        task t2 (in_port kernelStrm, out_port outStream){
                kernel{
                        for(i=0;i<dataSize;i++){
                                int temp <? kernelStrm;
                                outStream <! temp;
                        }
                }
        }
}
```

Figure 7: Multi-Kernel code in new style. The same function as Fig. 3.

6 Application Performance Metrics

In order to judge whether or not a specific application is performing well or poorly a metric is needed. Ideally we would like something that is platform neutral. To accomplish this we have come up with a metric for Macah applications based on the II provided by SPR analyze and the number of iterations provided by the compiler's fsim. The result of multiplying these two parameters together is an approximation of the number of cycles it takes a program to

complete. In a single kernel application it can then be divided by the number of data points being processed to get a general sense of the amount of processing per data item. In multi-kernel programs it is more difficult since the kernels have different II and number of iterations. A rough approximation for the system performance can be found in some of these cases by using the largest (II*iterations) value. Even here, if data starvation is causing stalls in the system it may not accurately represent the performance. The metric does have its problems but by comparing the number of data items into and out of a kernel to the product a reasonable estimate of performance can be determined.

7 Positron Emission Tomography Event Detection Application

7.1 Background

Positron Emission Tomography (PET) [3][4][5] is a nuclear medical imaging technique that produces a three dimensional image of a patient's biological processes. An example image is shown in Figure 8. A short-lived radioactive tracer isotope attached to a biologically active molecule is introduced into the body. This allows a radiologist to track molecules as they are transported throughout the body and absorbed in various tissues. In this sense it is very different from other medical imaging techniques like computed tomography (CT) or magnetic resonance imaging (MRI) scans which focus on generating structural images of the body rather than functional.

Figure 8: Whole Body PET scan. [6]

The isotope itself is not easily traceable, but as it undergoes positron emission decay it releases a positron and a neutrino. After traveling a short distance the positron will collide with an electron, annihilating both particles and producing a pair of high energy gamma photons, Figure 9. The majority of the photons created during the annihilation event are produced such that they move at almost 180 degrees from each other, traveling in roughly a straight line. This phenomenon is what allows us to trace the pairs of photons back to their source.

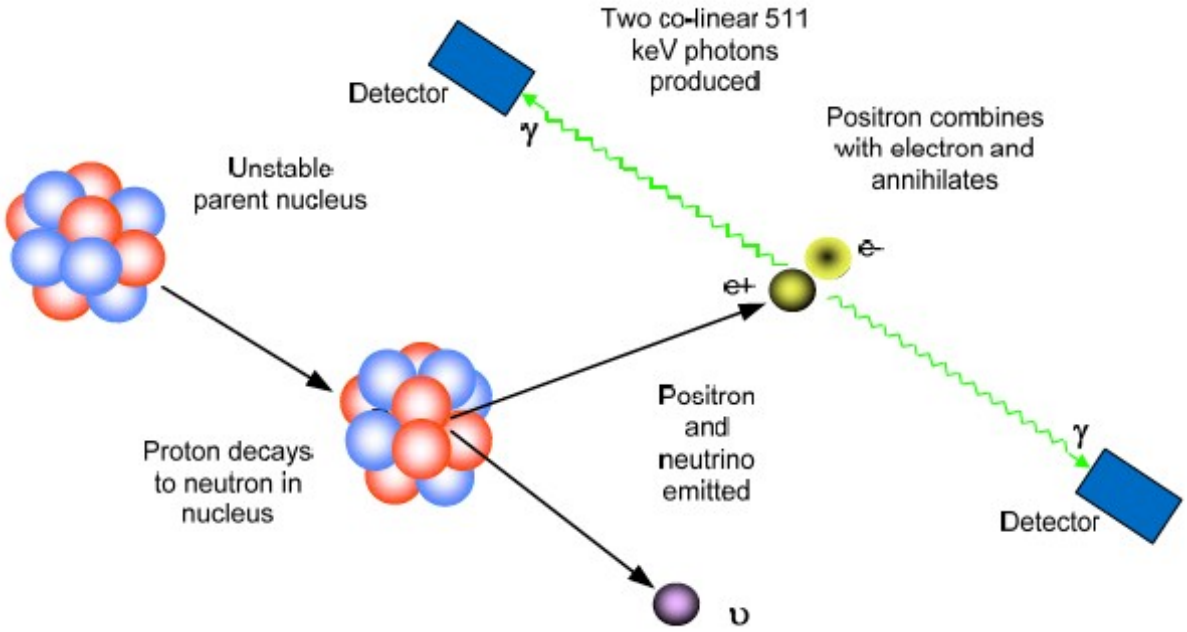

Figure 9: Positron emission decay [6]

A circular array of sensors is used to detect the photons. Normal photodetectors are not able to detect this frequency of light so these sensors contain scintillator crystals which absorb the high energy photons and reemit a burst of light in the visible range. These bursts of light can then be detected by either photomultiplier tubes or silicon photodiodes like those in Figure 10.

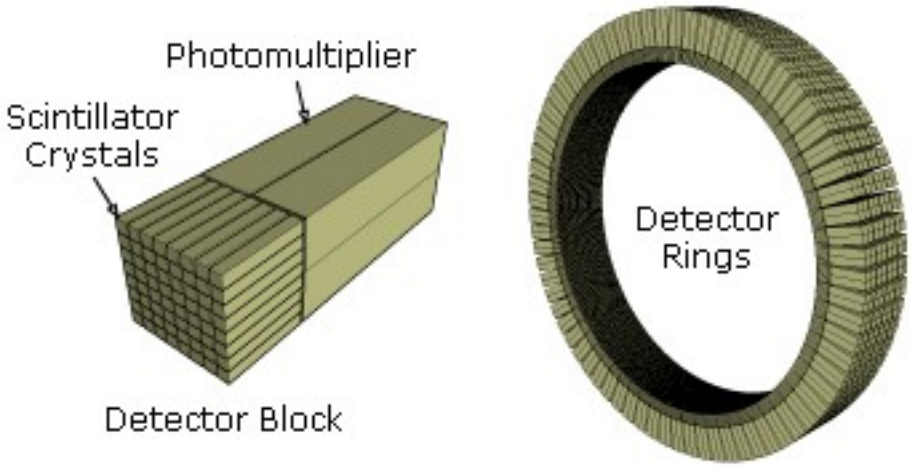

Figure 10: PET Detector and ring configuration.[6]

7.2 Event Detection

Two pieces of information are needed from the sensors to approximate the location of the annihilation event: the location a particular photon hit the sensor and the time it hit the sensor. The algorithm presented here is to determine the time a particle hit the sensor. The input data for the algorithm consists of a series of voltage pulses sampled at 125 MHz each, with a pulse duration of 24 samples, and a max peak amplitude of 1.9 V. Prior to the event detection the data has been run through a lowpass filter with a cutoff of 16.7 MHz. The duration of the pulses is consistent between pulses, but the amplitude varies. Example pulse data after filtering before input into application are shown in Figure 11.

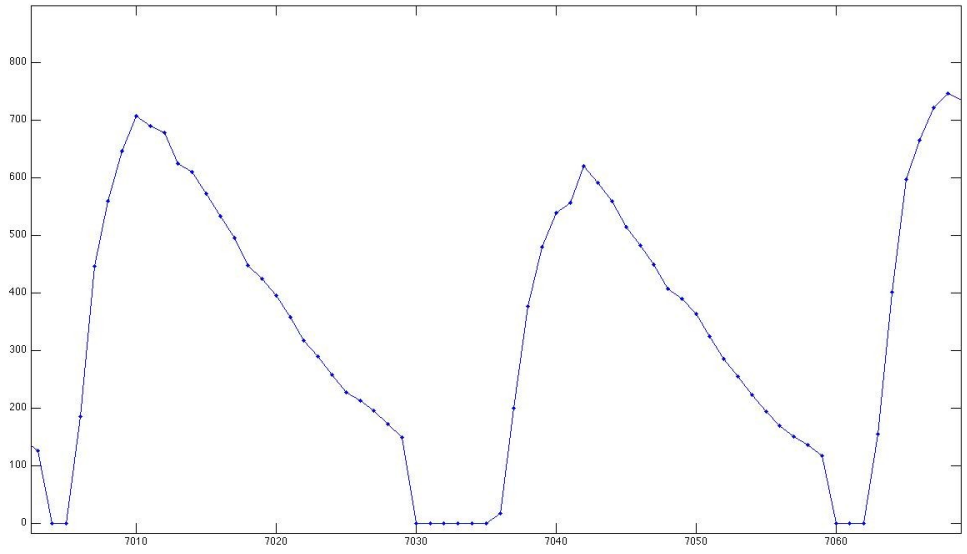

Figure 11: Example of PET data pulses after filtering.

Determining the coarse grain time for an event is simply a matter of determining the first data point considered to be part of the pulse. However this level of resolution is insufficient for this type of application, which requires fine grained timing resolution to properly match photon pairs and reconstruct the image. In order to determine the fine grained start time the pulse must be extrapolated beyond the set of sampled data. This can be achieved because of the characterization of the pulses performed by Mike Haselman[5]. This characterization showed that the pulses were found to have the same shape and duration but different amplitudes. An ideal pulse was also generated based on the data to serve as a baseline which can be used to normalize an individual pulse and extrapolate the start of that pulse. Figure 12 illustrates the normalization of an unknown pulse (blue) to a reference pulse (red). Since the extrapolation is based on the first point it is only necessary to normalize that point shown as the black point in the figure. The point must be multiplied by a normalization factor derived by dividing the normalized pulse area by the area of the pulse under consideration. Once the

corresponding point in the reference is determined the difference between it and the reference pulse start can be used to find the fine grained start of the unknown pulse.

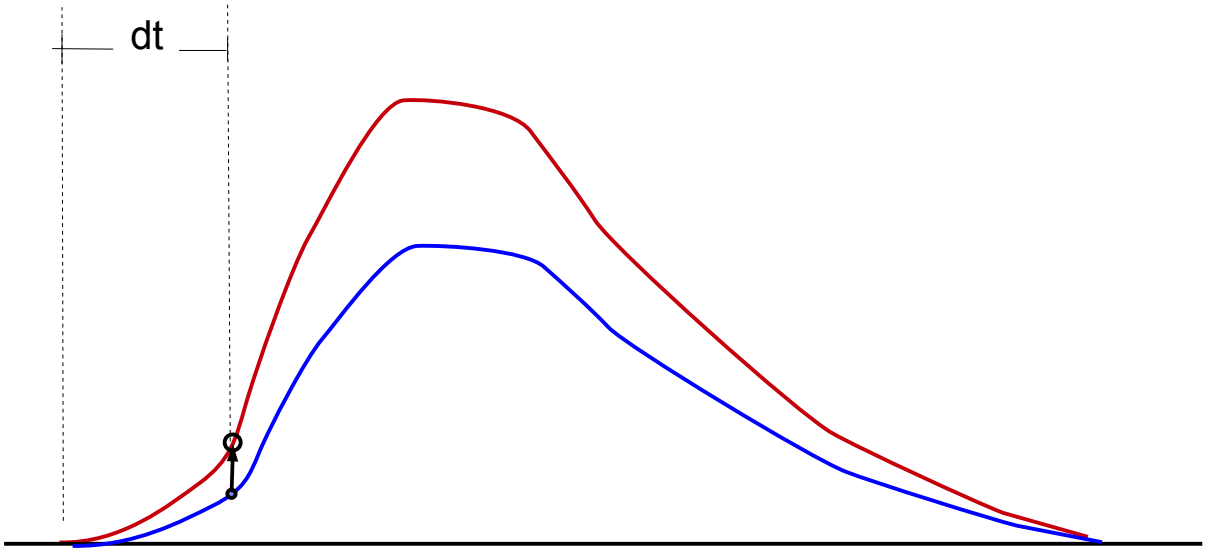

Figure 12: Pulse normalization. Reference pulse in red and pulse under consideration in blue. The black point is the first point of the pulse under consideration pulse.

The existence of an event is determined using two thresholds. These thresholds can be varied and have effects on accuracy and error rate. An event is considered to exist when a data point surpasses the first threshold and its following point surpasses the second. Once an event is detected a running sum of the 24 data points is computed to determine the area, and then the pulse's normalization factor is calculated. The first point of the pulse is multiplied by the factor, and using a lookup table derived from the reference pulse the time difference between the first point and the true start is determined. The fine grain time is then determined from the course grained time and this time difference. The pseudo code can be seen in Figure 13.

```
int n = 0;
int count = 0;
While(data remains){
      if(!eventFound){
            if(data[n-1] > Threshold_1 && data[n] > Threshold 2){
                  eventFound = True;
                  sum=data[n-1]+data[n];
                  count=2;
            }
      }else{
            sum+=data[n];
            count++;
            if(count = pulse width){
                  ratio = ref_area/sum;
                  normalized_point=first_data_point*ratio;
            }
      }
      n++;
}
```

Figure 13: Pseudo code for PET event detection algorithm.

7.3 Implementations

The PET event detection application has been implemented in both single kernel and multi-kernel Macah. The basic algorithm remains the same for both versions, but it is split into two asynchronous kernels in the multi-kernel version.

7.4 Single Kernel Implementation

The single kernel implementation of the application was created after the multi-kernel version in order act as a method of comparison and to have another benchmark for the original Mosaic tool chain. It was written in the original Macah style rather than using the

new syntax and API features. The basic structure is shown in Figure 14.

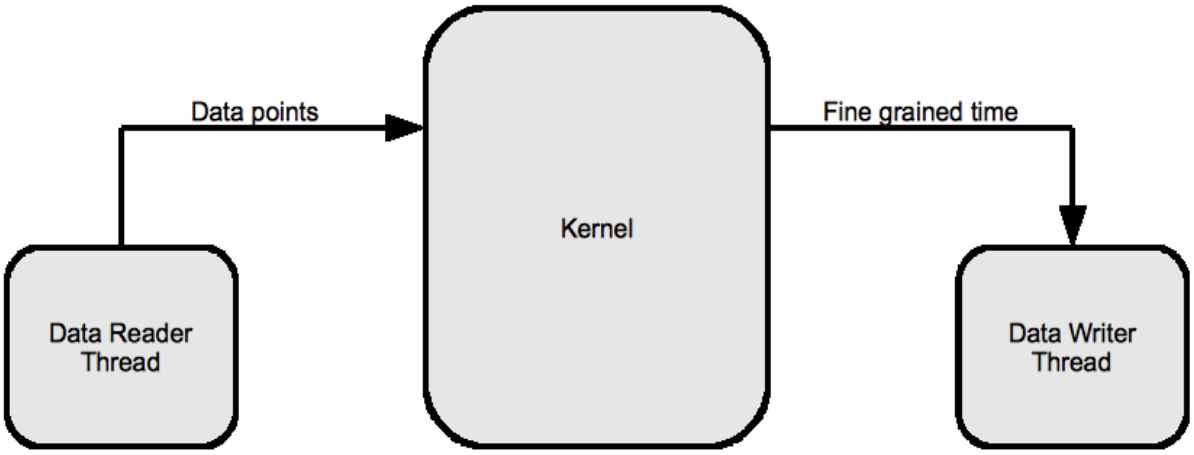

Figure 14: Single kernel PET data flow.

In addition, this implementation has the additional ability, compared to the multi-kernel version, to run independent data streams in parallel. This allows it to run data streams from different sources, which is extremely useful in applications like PET where there are a large number of data streams coming from independent sensors that need to be processed. This was achieved by using a Macah *FOR* loop to add an extra dimension to the two for loops already present in the algorithm. The abstract diagram of the system is shown in Figure 15. The number of data streams is defined as an ADPC and can be used to scale it to different architectures.

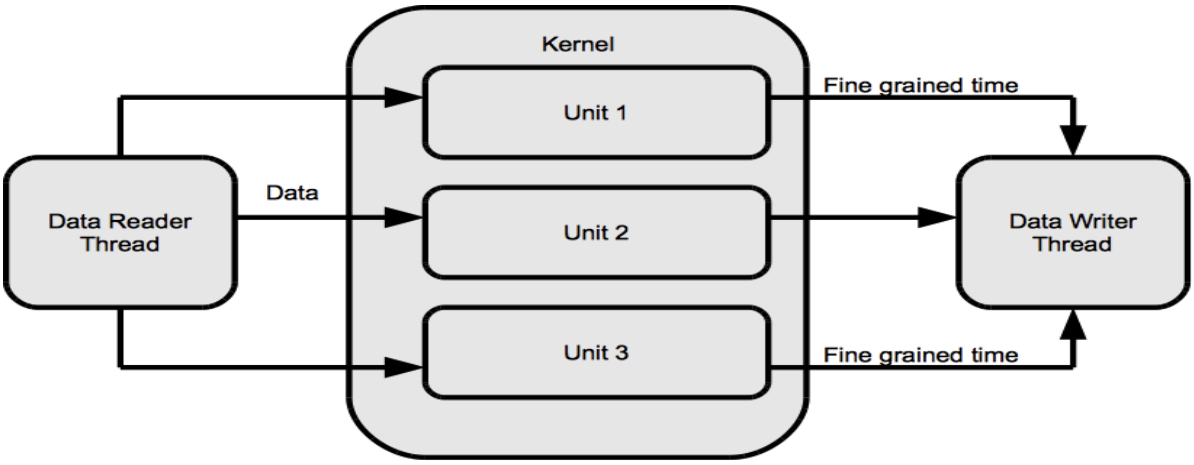

Figure 15: Abstract diagram of single kernel with multiple data streams.

7.5 Multi-Kernel Implementation

The multi-kernel PET event detection application was originally written in the old Macah style. The primary purpose was to help explore how the new style should be structured and what functions the new API would perform. This version was made multi-kernel by using different threads to represent the asynchronous kernels which were then connected using streams. Through this the multi-kernel application could be simulated for functionality, but could not move farther into the tool chain. Several functions used to set up the threads and streams required use of unsupported functions leftover from an older version of the pstreams library. The dependence on these functions was removed in the API improvements that made multi-kernel officially supported. After the API was completed the application was converted into the new style to test the functionality of the modifications and to provide an early benchmark.

The multi-kernel implementation of this application is made up of two kernel tasks and two data flow/memory access tasks. The idea behind splitting the algorithm was to have the first

kernel perform the thresholding and pulse summation and the second perform the math required to determine the fine grain start time of the pulse. The two memory access tasks act as the data source and sink for the kernel tasks and deal with main memory access. The thresholding task performs operations on every data point in the set, but the operations are simple conditionals and additions so it was believed that it would be fast. The math-heavy second kernel task would need to act irregularly as events were detected and it was believed that it would be slower since it had to perform more complex operations such as multiplies and divisions. By moving the math to a separate asynchronous task the thresholding task can continue to stream the data in and look for events without stopping to perform the calculations. The math task has more than enough time to perform the calculations before the next event is ready to be processed. The structure is shown in Figure 16.

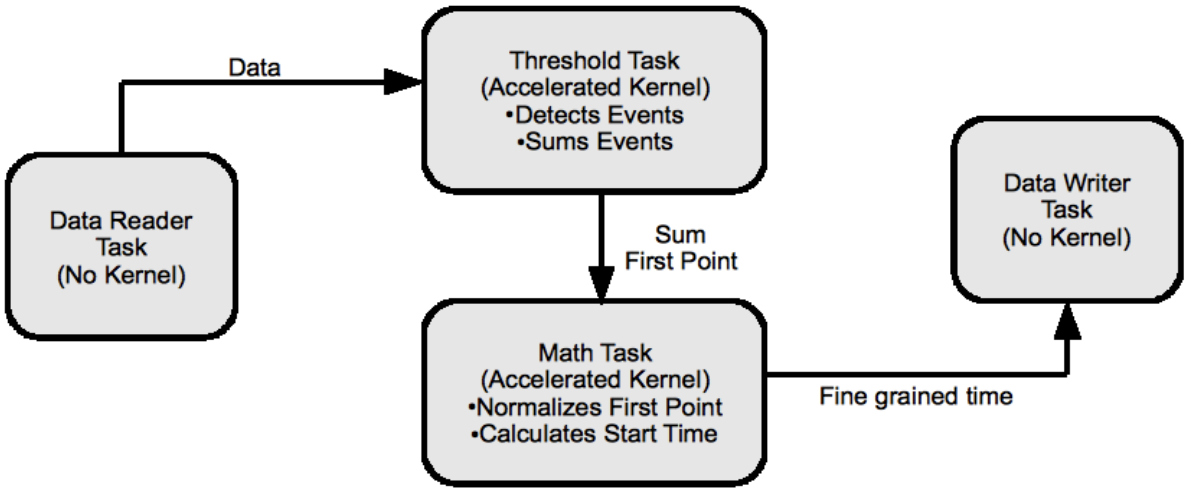

Figure 16: Multi-kernel structure.

7.6 Optimizations

During the development of both versions the divisions were removed from the code and

replaced with look up tables (LUTs). This was necessary since many accelerators do not possess ALUs capable of division. In effect this removed many of the slow elements of the calculation, improving the speed of the system at the cost of additional memory requirements.

After the initial multi-kernel version of the application was finished, measurements of the initiation interval (II) and number of iterations were made. The measured II of the application was larger than expected. In order to reduce it, the algorithm was restructured, moving and removing lines of code to reduce dependencies.

SPR identified the section with the largest II loop of the code to be in the thresholding section. In order to reduce the II this loop was analyzed by first producing a DFG of the critical section by hand. After analyzing the DFG and the code several modifications were made to the algorithm. Figure 17 shows the code before optimization and Figure 18 shows it afterward. The original had four independent conditional sections which were converted into an *if else* set with nested if statements inside. This stops the system from having to check the four conditionals each time. By storing the previous value of the data stream the thresholding can be combined into one *if* statement. Also in the code in Figure 17 the variable count is used often in both the conditionals and their bodies. This introduces a dependency chain for this variable which makes it difficult for the compiler to parallelize the code.

```
while(data remains){
        temp = next data from stream
        if(temp <= THRESHOLD2 &&  count == HOLDTIME-1){
                sum=0;
                firstPoint=NULL;
                eventTotal--;
                count=0;
        }
        if(temp > THRESHOLD1 && count == 0){
                count=HOLDTIME;
                firstPoint=temp;
                timestamp=i;
        }
        if(count > 0){
                count--;
                sum+=temp;
        }
        if(count==1){
                Send results to next kernel
                Reset variables
        }
}
```

Figure 17: Critical section of code for reduction of II. Prior to optimization.

```
while(data remains){
        temp = next data from stream
        if(foundEvent){
                if(preVal > THRESHOLD1 && temp > THRESHOLD2){
                                count=2;
                                sum=preVal+temp;
                                foundEvent=1;
                                firstPoint=preVal;
                                timestamp=i-1;
                }
        }else{
                sum=sum+temp;
                if(count==HOLDTIME-2){
                        Send results to next kernel
                        Reset variables
                }else{
                        count++;
                }
        }
        preVal=temp;
}
```

Figure 18: Critical section of code for reduction of II. After optimization.

The single kernel version also required additional optimizations not strictly required by the algorithm in order to be correctly mapped to the current Mosaic architectures. They were introduced to overcome the size limits of the memories used in the generated architectures. In the PET event detection application the LUT that replaced the area division was originally 16393 entries with each of them taking one word of space in memory. This means that this one LUT requires more than 16k words of memory while the architecture memories were 1024 words per memory unit. In order to accommodate this several modifications were made to the program. The first was to reduce the table size from 16k to 10k entries. This was done by trimming entries below and above specified limits. The limits were determined by finding the maximum and minimum area values in the data sets available. The next step was to

implement half words: splitting the table in half and storing two 16 bit entries instead of one 32-bit, cutting the table down to 5k entries. Finally, those 5k entries were broken up into 5 distinct memory blocks, each of which could be mapped to a distinct memory in hardware.

7.7 Performance Results

The performance results for several variations of the PET application for an input data set of 20k data points with roughly 550 pulses is shown in Table 2. The II optimizations discussed in section 7.7 reduce the II of the single kernel version and the multi-kernels thresholding kernel from 11 to 4 with no change to the number of iterations. These optimizations have reduced the II significantly without any drawbacks, which is a strong win for the application, giving it a work per incoming data point of roughly four. Since resource II is not factored into an architecture neutral approach such as this the II remains the same when the LUT memory modifications are made to the single kernel. It should be noted the IIs presented here are minimums and a particular architecture's constraints will determine actual II.

Version of Application (MK = Multi-Kernel)	**II** For multi-kernel: Threshold kernel/Math kernel	**Iterations**
MK - ED2 (No optimizations)	11/3 cycles	20002/577
MK - ED3 (II optimized)	4/2 cycles	20002/577
SK - ED_Single2 (No optimizations)	11	20002
SK – ED_Single3 (II optimized)	4	20002
SK – ED_Single6 (II optimized + LUT Memory Modifications)	4	20002

Table 2: II and number of iterations for various versions of PET application.

The single-kernel resource vs. II tradeoff can be seen in the graph in Figure 19. The resources

represented in the graph are clusters in a generated mosaic architecture where each cluster contains an ALU and memory among other things. As one would expect when the resources are constricted the II suffers, and, as resources are added, it improves with diminishing returns. It reaches the minimum II listed in Table 2 after being assigned 6 clusters.

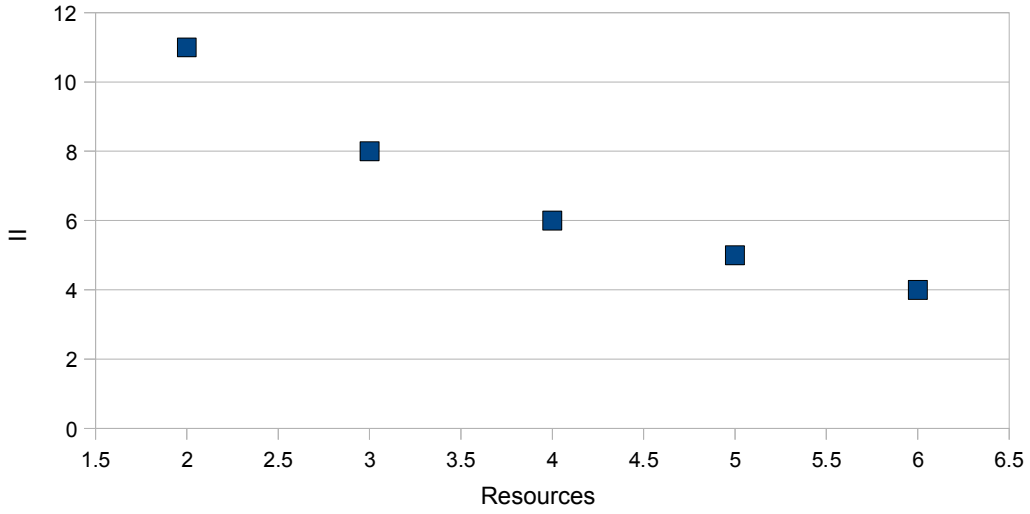

Figure 19: Resources vs. II for Single-kernel version of PET Application.

From the results in Table 2 the optimized multi-kernel version's threshold kernel runs on average approximately 69 times slower then the math kernel at the minimum II. In order to attempt to balance the loads the threshold kernel needs to receive considerably more resources during floor planning. Figure 20 shows the II vs. resources for the threshold kernel for four placements. Similarly to Figure 19 as resources are added the II improves up until it reaches the minimum II of 4. The math kernel was held constant with only 1 resource and an II of seven. The threshold kernel has hit its minimum II of four with five clusters and additional resources would provide no II benefit making the total resource count six. When

taken into account the average number of iterations and consumption rates for the application even at minimum II the threshold can't produce fast enough to keep the math kernel busy. Comparing the single kernel and multi-kernel floor planes shows that the total number of resources used for both is six and provide identical performance. For this application there was no performance benefit to the multi-kernel approach.

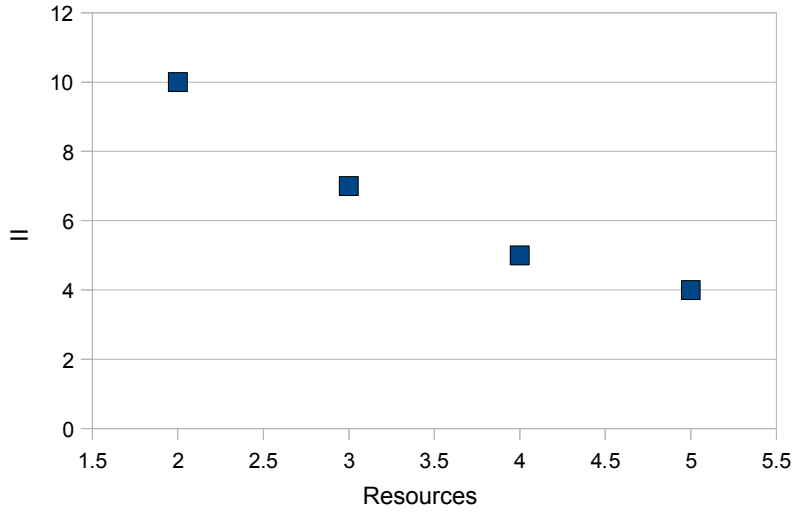

Figure 20: Resources vs. II for threshold kernel.

8 Conclusions

In this work the benefits of the new features of the multi-kernel API were examined. Also a sample application created in both the new and old versions of Macah, and in Single and multi-kernel variations is discussed.

The PET application helps to demonstrate the benefits of coding in the new Macah style, as well as the potential advantages of multi-kernel applications. The PET event detection application is not however the quintessential multi-kernel Macah application that clearly

demonstrates the usefulness of this addition to the tool chain. The performance improvement gained by moving the division in the PET application into LUTs removed much of the reason for making this application multi-kernel in the first place, as it greatly increased the speed of the kernel. Also the simple linear nature is not a complicated enough communication pattern to make the manual separation of the algorithm strictly necessary. As was shown in section 7.7 the single and multi-kernel versions had the same performance and resource consumption. The math kernel was simply too fast when compared to the thresholding kernel even with the minimum resource usage. In a case where the application is running multiple data sets in parallel there may be a gain if the multi-kernel version shares a single math kernel between multiple thresholding kernels when compared to a parallel single-kernel version.

The conversion of the LUTs also brought up an issue with memory on these types of co-processor accelerators, namely the need for larger blocks of memory for storing precomputed mathematical functions that are ether too slow or impossible to compute on the simple accelerator processing elements. It is fairly common to use such LUTs to perform divisions or trigonometric functions in many accelerated applications. If the largest memories on these distributed systems are too small for the LUT then it is required to break it up either in the tools or by hand. At the same time these larger LUTs blocks have a limited number of access ports and communication channels to support access for multiple asynchronous data streams which would make up for the size problems. Making effective use of small distributed memories under such circumstances is a problem for these types of systems.

The metric used for performance analysis, while adequate for the PET application presented here, does not translate well to more complex applications. We lack an effective architecture neutral method for determining both the system level performance of the algorithm but also the individual kernel performance.

8.1 Future Work

There are many places for improvement involving the multi-kernel Macah, including better integration of the tools in order to provide a simpler interface and means of extracting performance and profiling information for the application developer. There is still a great deal of difficulty in extracting useful performance information about an application quickly and easily. Improvement of the scripts controlling these actions would allow a faster iterative process when developing and optimizing applications. We also need to develop a larger set of multi-kernel applications and benchmarks that more clearly demonstrate the befits of the system.

9 References

[1] B. Ylvisaker, B. Van Essen, and C. Ebeling, “A Type Architecture for Hybrid Micro-Parallel Computers,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, April 2006.

[2] B. Ylvisaker, A. Carroll, S. Friedman, B. Van Essen, C. Ebeling, D. Grossman, S. Hauck, "Macah: A "C-Level" Language for Programming Kernels on Coprocessor Accelerators", *Technical Report*, 2008.

[3] Allan Carroll, Stephen Friedman, Brian Van Essen, Aaron Wood, Benjamin Ylvisaker, Carl Ebeling,Scott Hauck, "Designing a Coarse-grained Reconfigurable Architecture for Power Efficiency", *Department of Energy NA-22 University Information Technical*

Interchange Review Meeting, 2007.

[3] N. Johnson–Williams, *Design of a Real Time FPGAbased Three Dimensional Positioning Alogorithm for Positron Emission Tomography*, M.S. thesis, University of Washington, Washington, USA, December 2009.

[4] D. Dewitt, *An FPGA Implementation of Statistical Based Positioning for Positron Emission Tomography*, M. Eng. thesis, University of Washington, Washington, USA, Jan 2008.

[5] M. Haselman, R. Miyaoka, T. K. Lewellen, S. Hauck, "FPGA-Based Data Acquisition System for a Positron Emission Tomography (PET) Scanner", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2008.

[6] http://en.wikipedia.org/wiki/File:PET-detectorsystem_2.png