Macah: A "C-Level" Language for Programming Kernels on Coprocessor Accelerators

Benjamin Ylvisaker Allan Carroll Stephen Friedman Brian Van Essen
Carl Ebeling Dan Grossman Scott Hauck
University of Washington
{ben8, vanessen, sfriedma, allanca, ebeling, djg}@cs.washington.edu hauck@ee.washington.edu

Abstract

Coprocessor accelerator architectures like FPGAs and GPUs are increasingly used in embedded systems because of their high performance on computation-heavy inner loops of a variety of applications. However, current languages and compilers for these architectures make it challenging to efficiently implement kernels that have complex, input-dependent control flow and data access patterns. In this paper we argue that providing language support for such kernels significantly broadens the applicability of accelerator architectures. We then describe a new language–called Macah–and compiler that provide this support.

Macah is a "C-level" language, in the sense that it forces programmers to think about some of the abstract architectural characteristics that make accelerators different from conventional processors. However, the compiler still fills in several important architecture-specific details, so that programming in Macah is substantially easier than using hardware description languages or coprocessor-specific assembly languages. We have implemented a prototype Macah compiler that produces simulatable Verilog which represents the input program mapped onto a model of an accelerator. Among other applications, we have programmed a complex kernel taken from video compression in Macah and have it running in simulation.

1. Introduction

Coprocessor accelerator architectures, like field-programmable gate arrays (FPGAs) and graphics processing units (GPUs), have substantially higher execution unit density than sequential processors, but little to no support for unpredictable program behavior. The higher execution unit density translates into higher performance–with respect to time and/or energy efficiency–on applications that fit the architectural constraints. Applications from a variety of domains, including signal, image and video processing, cryptography, computational biology, and scientific simulation, have been profitably accelerated with coprocessors.

An important weakness of accelerators is that programming them requires unconventional languages and/or compilers, and the tools currently available are hard to use. In particular, these tools are weak at handling applications that have moderately complex input-dependent control flow and data access patterns. To address this weakness, we have developed a "C-level" programming language for accelerators, called Macah, and a prototype compiler. Macah makes it easier to program more complex applications on accelerators, because it carefully balances the strengths of human programmers and algorithmic compilers.

Coprocessors can only accelerate applications that are repetitive and predictable, but there are degrees of repetitiveness and predictability. We divide this spectrum of algorithms up into three ranges: *brute force*, *efficient*, and *fast*. Brute force algorithms, like dense matrix-matrix multiplication and finite impulse response (FIR) filters, are almost perfectly predictable and have simple repetitive control flow and data access patterns. Efficient algorithms, like fast Fourier transforms (FFTs) and dynamic programming algorithms such as Smith-Waterman sequence alignment, are still highly predictable, but have less repetitive control flow and/or data access patterns. Fast algorithms use input-dependent heuristics that are not completely predictable and repetitive to avoid unnecessary computation at the expense of accuracy in a way that the algorithm designers deem acceptable.

Existing programming tools work well for many brute force–and even some efficient–algorithms, but make implementing fast algorithms either harder than is necessary or impossible. Fast algorithms pose two primary challenges. By definition, the logic of such algorithms is complex, meaning that using low-abstraction languages, like hardware description languages (HDLs) or accelerator-specific assembly languages, is extremely time consuming and error-prone. Because the control flow and data access patterns are somewhat unpredictable and irregular, purely data-parallel languages simply do not apply, and systems that depend on static loop and array analyses for automatic vectorization and parallelization do not work well or at all.

Macah makes programming fast algorithms on accelerators easier by providing a "C-level" abstraction for this family of architectures. We are referring to C's role as a "portable assembly language" for conventional sequential processors. Well written C programs are mostly portable–and *performance-portable*–across essentially all conventional sequential (or "von Neumann") processors and compilers. Coprocessor accelerators do not implement this sequential model, so we need a new abstract model and corresponding "C-level" programming language to fill the equivalent space in the programming ecosystem.

The remainder of the paper is organized around a running example, block matching motion estimation, which is the most computationally significant part of video compression for modern, high compression-ratio codecs. We use this application to motivate the novel features of Macah and our prototype compiler, the description of which constitute the bulk of the paper. We have also pro-

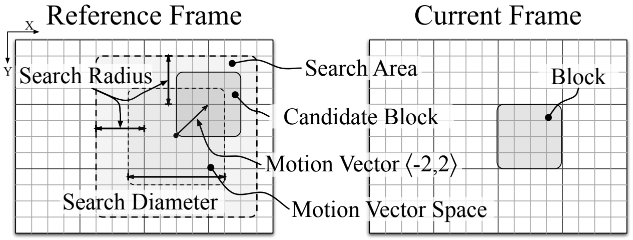


Figure 1. An illustration of some motion estimation terminology. In this picture, the frames are 16px × 12px, the blocks are 4px × 4px, and the search radius is 3px.

grammed other applications in Macah and believe it to be applicable to any application that fits the constraints of coprocessor accelerators. Our compiler currently generates code for a simulated accelerator, which is described in the final section.

2. Block-Matching Motion Estimation

Block-matching motion estimation (BMME) is the part of video compression that finds similar blocks of pixels in different frames. Video codecs that support high compression ratios, like H.264, allow blocks of pixels in one frame (the *current* frame) to be defined as similar to some other block of pixels in another frame (the *reference* frame). The difference in position of the two blocks within their respective frames is called the *motion vector* (MV). The MV plus the small pixel value differences between the two blocks can be encoded in far fewer bits than the raw pixel data. Motion estimation terminology is illustrated in figure 1.

During the compression process, the encoder must decide which block in the reference frame to use for each block in the current frame. This decision is made by the motion estimation algorithm. "Full search" (FS) is the simplest BMME algorithm. For each block in the current frame, it does a complete block comparison with every block in the reference frame that is within the *search radius* defined by the codec. This algorithm clearly finds the best MV, but at a huge computational cost. For example, one 1920×1080 frame of full search with a search radius of 15 requires almost 2 *billion* pixel comparisons.

Fortunately, BMME can be approximated very accurately with heuristics that drastically reduce the amount of computation required. The variety of BMME heuristics that have been proposed is impressive, but most use four basic ideas. 1) Motion estimation can be performed on down-sampled versions of the input frames, with detailed block comparisons only done in regions that the down-sampled comparison judged to be promising. 2) Block comparisons for a sparse subset of MVs can be tested first, with more detailed searching in the area of the best comparisons. 3) "Predictive" BMME algorithms first try MVs based on which MVs were best for adjacent blocks, which works because of the strong spatial correlation of motion in most video. 4) Finally, the search for a good MV for a particular block can be terminated early as soon as a "good enough" MV is found. When carefully combined, these heuristics can reduce the computational demands of BMME by two to three *orders of magnitude* compared to FS[1], with negligible reduction in video quality[1].

Heuristic approaches to BMME, a high-level sketch of which is shown in figure 2, are extremely fast, but also relatively complex. As a result, many researchers continue to use FS as a benchmark to demonstrate the power of coprocessor accelerators. But there is no reason to run FS on an accelerator when smarter algorithms can compute (almost) the same result at least as quickly on a conventional processor. Similar patterns exist in other application domains. For example, the BLAST tool uses a heuristic approach to compute the same biological sequence alignments as the Smith-Waterman algorithm in a fraction of the time, with only a small loss of accuracy. Just like the motion estimation example, BLAST is less predictable and more irregular that Smith-Waterman. Programming tools for coprocessor accelerators must be able to handle these fast algorithms for the architectures to be relevant to the given application.

```
for (i=0; i<ImgH/BlkH; i++) {
  for (j=0; j<ImgW/BlkW; j++) {
    dist[SrchDia][SrchDia];
    for (y=0; y<SrchDia; y++) {
      for (x=0; x<SrchDia; x++) {
        dist[y][x] = NOT_COMPUTED;
      }
    }
    searching = 1;
    while (searching) {
      chooseMV(dist, bestMVs, &mv);
      d = compareBlks(ref,cur,i,j,mv);
      dist[mv.i][mv.j] = d;
      searching = stillSearch(dist);
    }
    bestMVs[i][j] = mv;
  }
}
```

Figure 2. Sketch of heuristic motion estimation in C. The differences between heuristics are in how `chooseMV` picks the next motion vector and when `stillSearch` cuts off the search.

2.1 Accelerating Motion Estimation

The particular BMME implementation that we chose to accelerate is called the enhanced hexagonal search (EHS)[1]; other heuristic searches would have worked as well. EHS is a three phase algorithm. In the "predictive" phase, block comparisons are done for the $\langle 0, 0 \rangle$ MV and a small number of other MVs that were found to work well in adjacent blocks. The best of these MVs is taken as the initial center of the "coarse search" phase. In the coarse phase, block comparisons are done for the six MVs arranged in a hexagon around the current center MV. If any of them is better than the center, the best is taken as the new center and the process repeats. When the center is better than all of the points of the hexagon around it, the algorithm moves on to the "fine search" phase. In the fine phase, a few more blocks inside the perimeter of the final hexagon are compared. The best MV from the fine search phase is taken as the best MV for the block.

Three things are important about heuristic algorithms for motion estimation: 1) they still do a large amount of computation in the block comparisons 2) their control flow and data access patterns are highly dependent on the input data, and 3) intermediate results produced by the algorithm are used relatively quickly to make decisions about what to compute next. There is additional complexity in real video compression systems that we do not discuss in this paper, including variable block sizes, sub-pixel MVs, and multiple reference frames. This added complexity only increases the importance of support for sophisticated algorithms in accelerator programming systems.

Before implementing EHS in Macah, we will analyze its potential for acceleration and come up with a high-level strategy for accelerating it. This analysis is done relative to an abstract model of the behavior and performance characteristics of accelerators, like

[1] The speedup factor depends strongly on the search radius.

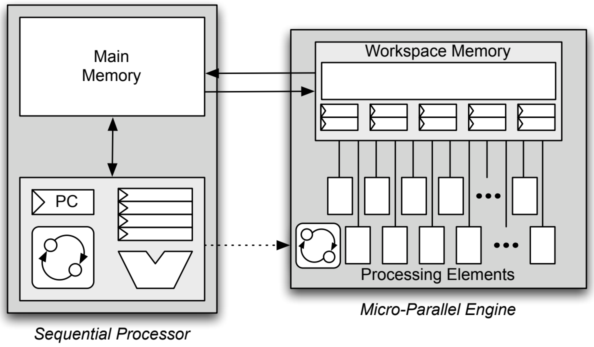


Figure 3. An abstract model of a hybrid processor-coprocessor system. Macah programmers need to think about writing accelerable code at the level of detail shown in this model.

that described in [2] and illustrated in figure 3. In order to write good Macah code, the programmer must do this kind of analysis, and therefore must have a high-level understanding of the structure and behavior of coprocessor accelerators. Though Macah looks like C, a well written version of EHS for a sequential processor will not go through the Macah compiler and produce efficient code. Macah is "C-level" for accelerators; it is not C.

Some of the most important constraints in the accelerator are the limited local memory and limited external communication bandwidth. The processing elements draw most of their input data from this local memory, because the bandwidth of the connection to the larger main memory is substantially lower than the computational throughput of the accelerator. Applications must have a sufficiently high computation to main memory bandwidth ratio in order to accelerate well. This constraint is overlooked surprisingly often.

EHS performs approximately 10 block comparisons per block on average. Each block comparison requires $16 \times 16 = 256$ pixel comparisons. Each pixel comparison requires approximately 4 operations (2 reads, one absolute difference, and one accumulation). So EHS requires about $10 \times 256 \times 4 = 10240$ operations per block.

The main memory bandwidth requirements depend on what is stored in workspace memory. At the very least we will need to transfer a block's worth of pixels for the current frame and the reference frame ($2 \times 16 \times 16 = 512$ pixels). Depending on how large and flexible the workspace memory is, we may have to transfer pixels from the reference frame multiple times. Optimistically assuming that each pixel is transfered only once, and assuming two bytes per pixel, that makes the main memory bandwidth requirement 1024 bytes per block. The computation to main memory bandwidth ratio comes to approximately 10 operations per byte. This number is reasonable, but leaves very little room for wasting memory bandwidth.

The next important feature of EHS that we consider is its complex control. Accelerator architectures have very poor support for unpredictable control flow, such as the logic to determine which block comparison to perform next. We believe that the best way to implement algorithms of this complexity is to partition them into a control part that executes on a conventional sequential processor and a kernel part that performs the repetitive, predictable piece of the computation. The control part sends commands consisting of block locations and motion vectors to the kernel part. The kernel part then does block comparisons and sends back computed block differences.

For such an implementation to work well, the accelerator must be integrated with a sequential processor. This could be die-level integration, as in FPGAs with embedded processors or board-level, as in products from XtremeData, Inc. and DRC Computer Corp.

Next we must consider what data can be buffered in the workspace memory. A single frame of 1920×1080 video is almost 4MB of data (assuming 16 bits per pixel). Real accelerators have workspace memory capacity in the range of low hundreds of KBs to very low MBs, so realistically we will only be able to store a few blocks worth of data in the workspace memory at a given time. This will affect how we do buffering in the Macah code.

Accelerators work by executing many simple operations concurrently on their simple processing elements (PEs), so we have to think about which operations can execute in parallel. The inner loops that perform a single block comparison are a simple reduction, so they will parallelize nicely. The only complication is the order in which pixels from the reference frame are accessed depends on the MV currently being tested. This fact will make the buffer for the reference frame slightly more complicated than the buffer for the current frame.

Finally, there is inevitably some latency involved in sending a MV from the sequential processor to the coprocessor and getting a result back. Therefore, we want to have multiple MVs "in flight" at any time to keep the whole pipeline full. However, at certain points in the motion estimation algorithm, there may only be one or two new MVs to perform block comparisons for before those results are needed to decide what to do next. To keep the pipeline full, we need to work on multiple blocks from the current frame simultaneously.

This requirement forces us to change the algorithm, because the predictive part of the sequential version needs to know what the best MVs are for its neighboring blocks. This change illustrates one of the most important weaknesses of a pure optimizing compiler approach to programming accelerators. Even if we assume that a compiler's loop and array analyses are smart enough to optimize irregular, input dependent code well, we generally expect compilers to not change the meaning of a program. We believe that this kind of super-aggressive optimization is best done in a separate code restructuring tool.

The accelerated implementation of EHS that we have sketched here is inarguably more complicated than the sequential version. The Macah code, parts of which are presented in the next section, is longer and more complicated than the C version as well. However, it is not clear how to program this kind of fast motion estimation algorithm in purely data parallel languages. The aggressive loop and array optimizations used by "C to gates" compilers do not work with data-dependent control flow and data access patterns. Our only remaining option for programming FPGAs, at least, is HDLs which force the programmer to think at an even more detailed level about the hardware in the coprocessor. To our knowledge, there are no published implementations of the fastest motion estimation heuristics for any coprocessor accelerator.

3. Macah

Macah is C plus threads and extensions designed for programming coprocessor accelerators. The reason to use an accelerator is to take advantage of the performance and energy benefits of executing kernels in a highly parallel fashion. The key issues for achieving this parallelism are being able to allocate local data structures to the workspace memory in a distributed way, using the limited bandwidth to main memory judiciously and pipelining the loops so that operations from different loop iterations can execute simultaneously. Macah's features are designed to make it easy to write kernels that achieve high performance, as long as the programmer understands accelerators well enough to do the kind of analysis demonstrated in the previous section. However, Programmers should not have to think about the details of a particular architecture.

```
 1 cmd_t stream cStrm = pstream_create(cmd_t);
 2 rslt_t stream rStrm = pstream_create(rslt_t);
 3 args = { cStrm, rStrm, curFrame, refFrame };
 4 pthread_create(&accelThread, NULL, accel, &args);
 5
 6 int dist[^NumBlks^][SrchDia][SrchDia];
 7 for (i=0; i<ImgH/BlkH; i++) {
 8   for (b=0; b<NumBlks; b++) {
 9     for (y=0; y<SrchDia; y++) {
10       for (x=0; x<SrchDia; x++) {
11         dist[b][y][x] = NOT_COMPUTED;
12       }
13     }
14   }
15   for (j=0; j<ImgW/BlkW - NumBlocks; j++) {
16     dist <<= 1;
17     for (y=0; y<SrchDia; y++) {
18       for (x=0; x<SrchDia; x++) {
19         dist[NumBlks - 1][y][x] = NOT_COMPUTED;
20       }
21     }
22     searching = 1;
23     while (searching) {
24       blockNum = chooseMV(dist, bestMVs, &mv);
25       cmd.code = COMPARE_BLOCKS;
26       cmd.i = mv.i;
27       cmd.j = mv.j;
28       cmd.b = blockNum;
29       cmdStrm <! cmd;
30       dist[blockNum][mv.i][mv.j] = IN_PROG;
31       searching = stillSearch(dist);
32     }
33     bestMVs[i][j] = mv;
34   }
35 }
```

Figure 4. High level sketch of fast motion estimation, updated to interface with a kernel thread that performs the block comparisons.

```
 1 void chooseMV(int dist[NumBlocks][SrchDia][SrchDia],
 2               mv_t bestMVs[ImgH/BlkH][ImgW/BlkW],
 3               mv_t *mv) {
 4   ...
 5   // complex logic to choose b, x and y
 6   ...
 7   if (dist[b][y][x] == IN_PROG)
 8     dist[b][y][x] <? rsltStrm;
 9   ...
10 }
```

Figure 5. `chooseMV` implements the heuristics of a particular motion estimation algorithm. Because the interface between the sequential logic and the kernel is asynchronous, this code might find a MV in the distortion table that has been sent to the kernel, but for which a result has not yet returned. In that case, the code does a blocking receive on the result stream.

Macah's features are streams, kernel blocks, FOR loops, shiftable arrays, architecture-dependent pseudo-constants, and dynamic scheduling annotations. We use motion estimation to motivate the language features, though we did not design the language specifically for this application, and have programmed several other applications in it. Four snippets of a Macah version of motion estimation are contained in figures 4, 5, 6 and 7.

The code in figure 4 is the main part that runs on the sequential processor; it is very similar to the C version sketched in figure 2. There is some extra startup code for creating the thread that the kernel will run in and streams for communicating commands and results. There are three substantial changes in the motion estimation code itself. Where the C version calls a function to do a block comparison for a MV, the Macah version sends a command to the kernel and marks that MV in the distortion table as currently being worked on. In the function for choosing the next MV to try, illustrated in figure 5, if the heuristic needs to know the distortion for a particular MV, and finds that entry marked with a `IN_PROG`, it blocks until the coprocessor sends back a result. Finally, there is an additional loop to let the sequential side send MVs from several blocks at the same time.

```
void refReader (px stream s,
                px refFrame[ImgH][ImgW]) {
  int ib, jb, i, j;
  for (ib=0; ib<ImgH; ib+=ImgH) {
    for (i=ib-SrchRad; i<ib+BlkH+SrchRad; i++) {
      for (j=-SrchRad; j<BlkW+SrchRad; j++) {
        s <! refFrame[i][j];
    } }
    for (jb=0; jb<ImgH; jb+=ImgH) {
      for (i=ib-SrchRad; i<ib+BlkH+SrchRad; i++) {
        for (j=jb+SrchRad;
             j<jb+SrchRad+BlkW; j++) {
          s <! refFrame[i][j];
} } } } }
```

Figure 6. Memory accessor function. This function is run in a separate thread. It feeds the kernel through stream s.

3.1 Streams

Streams are first-in, first-out data channels that two threads can use to communicate and synchronize. There are two different styles of streams in most Macah programs. *Memory accessor streams* have the kernel at one end and a function that only either reads data out of main memory and sends it down the stream, or receives data from the stream and writes it into memory at the other end. This kind of stream is used in the motion estimation code to read the frame data and load it into the local buffers. An example of a memory accessor function is shown in figure 6. The other kind of stream has active compute threads on both sides, like the command stream and result stream in the motion estimation code. Stream sends and receives also interact with kernel blocks in a way that eases automatic pipelining; this is discussed further below.

Streams are created with calls to built-in library functions. The `stream_create` function, shown in figure 4 on line 1 (fig. 4:1), builds a stream capable of carrying data elements of the given type. `pstream_free` deallocates the resources associated with a stream. The `mem_*_spawn` and `mem_*_join` functions (fig. 7:7-8) are for creating and destroying memory accessor streams. The spawn functions create a stream, spawn a new thread, and start running the given function in the new thread with the given arguments. The join functions wait for the thread on the other end of the stream to finish.

The basic operations supported by streams are send and receive, written `expS <! exp` (fig 4:29) and `exp <? expS` (fig 5:8), respectively. The default send (receive) operator blocks if the stream is full (empty). There are also non-blocking versions written `expW :: expS <! exp` and `expW :: exp <? expS` (fig 7:12), respectively. After one of the non-blocking stream operations executes, expW (W stands for "worked") is set to 1 or 0, depending on whether the operation actually succeeded. Non-blocking receives are used in the kernel to get commands, and are discussed further below.

Macah streams are unlike streams in languages like StreamIt[3], and StreamC[4]. In these languages, kernels are defined to consume and produce a particular number of stream elements per *firing*.

In other words, they have no send and receive operators that can execute conditionally. This more restrictive use of streams gives the compiler more opportunity to statically analyze the interactions of a group of kernels, but makes some programming styles difficult or impossible to use. For example, it is not clear how to program the motion estimation kernel that conditionally receives data into its buffers when it gets the command to move to the next block.

3.2 Kernel Blocks

Kernel blocks serve to mark what code should be run on the accelerator, and ease the challenge of pipelining loops by relaxing the order of evaluation rules. Kernel blocks are written like standard C blocks, preceded by the new keyword `kernel` (fig 7:9). The Macah compiler attempts to generate an accelerated implementation for the code in kernel blocks. If such an implementation is not possible for whatever reason, the compiler will report either an error or a warning, along with diagnostic information to help the programmer understand why the block cannot be mapped to the accelerator. Code outside of kernel blocks is translated to standard C, as discussed below. The motion estimation kernel is shown in figure 7.

In order to find enough parallel operations to keep the PEs busy, almost all kernels need to be pipelined. This means that different parts of adjacent loop iterations are executed concurrently. The order of execution rules inside kernel blocks have been subtly relaxed to accommodate this pipelining. Consider the simple example illustrated in figure 8. There is a loop with three operations: a receive, some computation, and a send. In the sequential implementation the first receive happens before the first send, which happens before the second receive, and so on. In the pipelined trace, however, the second receive happens before the first send. If the rest of the program that this code interacts with is expecting to receive a value on s2 before sending another value on s1, the pipelined implementation will cause the program to deadlock.

The motion estimation kernel has exactly this structure. The kernel receives a command on the command stream, computes a block difference, and sends back a result. Both streams are connected to the sequential thread that receives results, chooses what MV to try next and sends back commands. This circular structure in the stream communication structure has the potential to create deadlock, which is why the receive on the command stream in the kernel is non-blocking. If the latency of the pipelined kernel and the communication between the processor and coprocessor is long enough that the sequential thread cannot keep the kernel filled with commands, the non-blocking receive will fail, and "bubbles" will automatically be introduced into the pipeline.

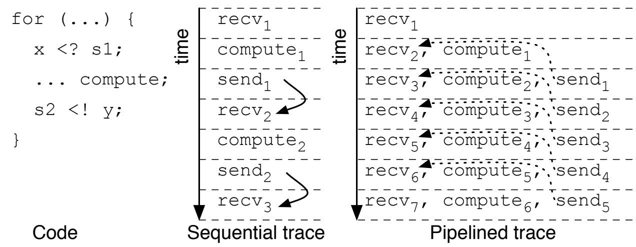
The semantics of Macah explicitly allow stream sends and receives in kernel blocks to happen "late" and "early", respectively, from the perspective of an outside observer. This relaxation permits the compiler to perform loop pipelining without analyzing the other code that interacts with the kernel through streams. This definition puts the onus on the programmer to ensure that their kernels do not send data out that causes other data to be received later in the same kernel, unless the proper precautions are taken. Tools for analyzing whole Macah programs for safety of stream communication patterns could clearly offer helpful error checking. In the spirit of Macah's "C-levelness", the default is to trust the programmer on this point.

```
 1 #define RefBuffH (BlkH+2*SrchRad)
 2 #define RefBuffW (NumBlks*BlkW+2*SrchRad)
 3 #define RowsPer (Ceil(RefBufH / BlkH))
 4
 5 px curBuff[^NumBlks^][BlkH][BlkW];
 6 px refBuff[BlkH][RowsPer][^RefBuffW^];
 7 refStrm = mem_reader_spawn(refReader, refFrame);
 8 curStrm = mem_reader_spawn(curReader, curFrame);
 9 kernel {
10   done = 0;
11   do {
12     recvCmd :: cmd <? cmdStrm;
13     if (recvCmd) {
14       switch (cmd.code) {
15         case CMD_COMPARE_BLOCKS:
16           FOR (i = 0; i < BlkH; i++) {
17             dists[i] = 0;
18             refI = (i + cmd.i) / BlkH;
19             for (j = 0; j < BlkW; j++) {
20               refJ = j + cmd.j + BlkW * cmd.b;
21               refPx = refBuf[i][refI][refJ];
22               curPx = curBuf[cmd.b][i][j]
23               dists[i] += ABS(curPx - refPx);
24           } }
25           dist = 0;
26           FOR (i = 0; i < BlkH; i++)
27             dist += dists[i];
28           rsltStrm <! dist;
29           break;
30         case CMD_NEXT_BLOCK:
31           curBuff <<= 1;
32           FOR (i = 0; i < BlkH; i++) {
33             for (j = 0; j < BlkW; j++)
34               curBuff[NumBlks-1][i][j] <? curStrm;
35             for (i2 = 0; i2 < RowsPer; i2++) {
36               if (i2 * BlkH + i < RefBuffH) {
37                 refBuff[i][i2] <<= BlkW;
38                 for (j = 2*SrchRad;
39                      j < RefBuffW; j++)
40                   refBuff[i][i2][j] <? refStrm;
41           } } }
42           break;
43         case CMD_NEXT_ROW:
44           FOR (i = 0; i < BlkH; i++) {
45             for (b = 0; b < NumBlks; b++) {
46               for (j = 0; j < BlkW; j++)
47                 curBuff[b][i][j] <? curStrm;
48             for (i2 = 0; i2 < RowsPer; i2++)
49               if (i2 * BlkH + i < RefBuffH)
50                 for (j = 0; j < RefBuffW; j++)
51                   refBuff[i][i2][j] <? refStrm;
52           }
53           break;
54         case CMD_DONE:
55           done = 1;
56           break;
57     } }
58   } while (!done);
59 }
```

Figure 7. Block comparison kernel in Macah. This code resides in the "accel" function referred to on line 4 of figure 4.

3.3 Shiftable Arrays

Shiftable arrays are simply arrays that support a shift operation in addition to the standard array indexing operation. The result of shifting an array left (right) by N is that each element of the array is moved to the left (right) by N places, assuming that array indices increase to the right. After a left (right) shift of N, the right-most (left-most) N places in the array are uninitialized. Shiftable arrays, in addition to being convenient for many application domains, help describe the kinds of regular PE to PE communication patterns that accelerators support well. The reference frame buffer in the motion estimation kernel is defined as a shiftable array because there is significant overlap between the search areas for adjacent blocks. When the kernel receives a command to move from one block to



```
for (...) {
  x <? s1;
  ... compute;
  s2 <! y;
}
```

time
recv$_1$
compute$_1$
send$_1$
recv$_2$
compute$_2$
send$_2$
recv$_3$
time
recv$_1$
recv$_2$, compute$_1$
recv$_3$, compute$_2$, send$_1$
recv$_4$, compute$_3$, send$_2$
recv$_5$, compute$_4$, send$_3$
recv$_6$, compute$_5$, send$_4$
recv$_7$, compute$_6$, send$_5$
Code
Sequential trace
Pipelined trace

Figure 8. Simple pipelining example

the next, it shifts the reference frame buffer by the block width and fills in the empty piece.

Shiftable arrays can be simulated with normal arrays and extra index arithmetic. However, shiftable arrays can be used to more directly describe the spatial relationships that exist in an algorithm, and potentially lead to a more efficient implementation.

3.4 FOR Loops

FOR loops are simply for loops that the user has declared should be unrolled completely at compile time. In the motion estimation kernel, the loops over the height of the block are all FOR loops (fig 7:16,26,32,44). Of course, the accumulations needed to produce a single distortion value for a block create a moderately long dependence chain, which is exactly why we rely on pipelining to overlap the computations of adjacent iterations.

Loop transformations like unrolling, interchange and blocking are well understood and can be performed by compilers. Advanced parallelizing and vectorizing compilers [5] use sophisticated linear algebra-based loop and array analyses to decide how to apply loop optimizations. However the extent to which they are applied can have a significant impact on important issues like workspace memory and main memory bandwidth requirements, and it is far from trivial to automatically decide what combination of transformations will produce good results. The authors of [6], who include pioneers of parallelizing and vectorizing compilers, state "The quality of compiler-optimized code for high-performance applications is far behind what optimization and domain experts can achieve by hand…the performance gap has been widening over time". We therefore consider it important to give the programmer the tools needed to express which operations should be carried out in parallel. In the future, it may be worthwhile to use these kinds of optimizations on the most inner loops of applications like motion estimation.

3.5 Architecture-Dependent Pseudoconstants

Macah is intended to be as portable as possible, but deciding how to structure a kernel to best exploit the local memory, external bandwidth and parallel computation resources of a particular accelerator often requires non-trivial application-level trade-offs. Architecture-dependent pseudoconstants (ADPCs) give programmers a tool to write code that can be automatically adapted to different architectures. They are typically used to control code features like the size of buffer arrays and the extent of loops. They are declared by the programmer via a call to the built-in function ADPC which takes two integers, a min and a max. The compiler then chooses a value in this range that produces efficient code. In the motion estimation code, the ADPC `NumBlks` (fig 4:6 and fig 7:5) controls the number of blocks from the current frame that are are buffered in workspace memory at same time. The size of the current frame buffer and reference frame buffer both depend directly on `NumBlks`. By using an ADPC instead of a fixed constant, the compiler can adapt the program to accelerators with significantly different amounts of workspace memory. This explicit technique gives a level of portability and automation that exceeds the *ad hoc* technique of manually tuning C #define values for each architecture.

Currently all Macah ADPCs are integers with min and max constraints, but generalizing the concept to other types and kinds of constraints may prove valuable.

3.6 Dynamic Scheduling Annotations

Dynamic scheduling annotations are not used in the code presented in this paper, but they are important in some applications, like molecular dynamics (MD) simulations. MD simulation is an N-body problem that simulates the movement of a collection of particles over time. The kernel of a MD simulation is the all-pairs interaction between the particles (*i.e.* $O(N^2)$ complexity). The computational complexity of this kernel is typically reduced by ignoring the interaction of particle pairs beyond some cut-off distance. The kernel of MD considers many pairs of particles, but only computes the forces between them if they are within the cut-off distance.

The standard strategy for implementing conditional control flow on most accelerators is predication, or if-conversion. That is, the conditionally guarded expressions and statements are unconditionally evaluated, and then their results are either used or discarded. The inefficiency of this strategy increases with the frequency with which results are discarded, and the quantity of resources used to do the discarded work. Even though the simplicity of accelerator controllers makes actually dynamically choosing to do some work or not relatively expensive, it is sometimes necessary to achieve reasonable efficiency.

In the MD simulation, the distance cut-off test is an example of conditional control flow that is better to dynamically evaluate, because for a given atom, the percentage of atoms in its neighboring regions that actually fall within its distance cut-off is relatively low (∼15%), and because the amount of computation needed to compute the force for a single atom pair is non-trivial.

Macah allows programmers to label conditional statements with the fraction of the executions that the programmer expects the expression to be true. The compiler is then able to use this information to decide if those conditional statements should be implemented with predication or some form of dynamic evaluation. This information could be gathered with profiling, too, but since the allocation of accelerator resources can have such a large impact on performance, it is important to give the programmer a way control it. One mechanism that facilitates dynamic evaluation are the non-blocking stream operators discussed in section 3.1.

3.7 Parallel Extensions

Task, *thread* or *process* parallelism is also important in embedded systems. The kind of parallelism exploited by Macah and accelerators can be integrated with these other forms of parallelism to program multiprocessor machines with multiple coprocessors. There are some interesting challenges raised by such an integration–for example, how multiple sequential threads can share a single coprocessor–but we have not yet investigated these issues. We believe, however, that coprocessor accelerators and chip multiprocessors (CMPs) should be seen as complementary, not competitive, architectural families. We are already seeing announcements of products from processor industry leaders that integrate multiple sequential cores and GPUs. We expect this kind of integration to continue.

4. Compiling Macah

A Macah compiler targeting a processor/accelerator system is currently in an early prototype state. The kernel bodies are compiled into configurations for a simulated accelerator and the non-kernel code is simply translated to C and compiled with a normal C com-

```
while (e) {
  S;
}
```

(a) Before

```
if (e) {
  do {
    S;
  } while (e);
}
```

(b) After

Figure 9. Loop inversion allows the body of the loop to be executed without additional conditional tests, if the outer **if** can be optimized away.

```
if (e) {
  a = x*2;
  b = y/3;
  c <? s;
}
else {
  a = z+r;
  while (1) {
    ...
  }
}
```

(a) Before

```
eThen = e;
eElse = !eThen;
aT = x*2;
bT = y/3;
if (eThen)
  c <? s;
aE = z+r;
if (eElse)
  while (1) {
    ...
  }
a = eThen ? aT : aE;
b = eThen ? bT : b;
```

(b) After

Figure 10. If conversion replaces conditional blocks with unconditional statements, selection expressions and individual predicated statements. This is almost always preferable for accelerators, which have poor support for unpredictable control flow.

piler. Kernels are translated into Verilog code that can be simulated both before and after the backend part of the compiler runs.

Our compiler is based on the CIL C parsing and translation infrastructure [7]. We have modified the parser and internal data structures to accommodate Macah's new features. Most of the analyses and transformations described in this section are well known. What we provide here are explanations of why and how they are applied differently in the context of coprocessor accelerators. In CIL, all loops are represented as infinite loops with explicit breaks and continues. This works well for us, because we need the loop optimizations we use to apply to all kinds of loops.

4.1 Kernel Partitioning

For each kernel block, the necessary control transfers between the sequential processor and the accelerator are automatically generated by the compiler, as are any data transfers that are necessary for data structures that are accessed both inside and outside of a kernel. This piece of compiler support is conceptually simple, but quite valuable, because systems that require kernel code and non-kernel code be written in different languages create a large amount of manual interfacing work for the programmer.

4.2 Function Inlining

Function inlining is a well known optimization, which replaces calls to a function with a copy of the body of the function. Complete function inlining is required for accelerators, because they do not support function calls.

4.3 FOR Loop Unrolling

A FOR loop is replaced by multiple copies of its body, with constants filled in for the loop induction variable. It is considered an error, if the initial value, termination condition or induction variable increment cannot be computed at compile time.

```
while (1) {
  S1;
  if (e1) {
    while (1) {
      S2;
      if (e2)
        break;
    }
  }
  S3;
}
```

(a) Before

```
before = 1;
inner = 1;
after = 0;
while (1) {
  if (before) {
    S1;
    before = 0;
    if (!e1) {
      inner = 0;
      after = 1;
    }
  }
  if (inner) {
    S2;
    if (e2)
      after = 1;
  }
  if (after) {
    S3;
    before = 1;
    inner = 1;
    after = 0;
  }
}
```

(b) After

Figure 11. Loop flattening example for the special case where there is only one inner loop.

4.4 Array Scalarization

Array scalarization breaks arrays up into smaller pieces that can be accessed independently, when it is legal to do so. The motion estimation code is carefully structured so that after FOR loop unrolling, both the current frame buffer and the reference frame buffer are accessed only by constants in their first dimension. It is then clear without any sophisticated array analyses that each sub-array can be allocated to a different physical memory and accessed in parallel. The `dists` array will be similarly scalarized.

Shiftable arrays that are not scalarized are implemented as normal arrays with additional offset and size variables. Indexing is performed relative to the offset, modulo the size, and shifting is implemented as offset arithmetic. If an architecture has built-in support for this kind of indexing, we take advantage of that.

4.5 Loop Inversion

Loop inversion is a simple loop transformation illustrated in figure 9. In the common case that the outer conditional test in figure 9(b) can be optimized away entirely, the body of the loop after the transformation is not guarded by any conditions. Because accelerators have little to no support for executing large pieces of code conditionally, it is useful to reduce the "conditionality" of statements.

4.6 If-Conversion

If-conversion is illustrated in figure 10. After if-conversion, both sides of conditional branches are executed unconditionally. Variables that are modified on either side have to be renamed, with the final result selected after both sides have executed. Statements with side-effects, like the stream receive and the loop in the example have to be individually predicated. The current compiler completely converts all if-then-else and switch-case statements, though as mentioned previously this is sometimes an inefficient strategy.

4.7 Loop Flattening

In order for kernels to perform well, the loops must be pipelined. The actual pipelining process is described below. Pipelining algorithms, like software pipelining [8] and iterative modulo scheduling

```
while (1) {
  S1;
  while (1) {
    S2;
    if (e1)
      break;
  }
  S3;
  while (1) {
    S4;
    if (e2)
      break;
  }
  S5;
}
```

(a) Before

```
count = 1;
while (1) {
  if (count == 1) {
    S1;
    count++;
  }
  if (count == 2) {
    S2;
    if (e1)
      count++;
  }
  if (count == 3) {
    S3;
    count++;
  }
  if (count == 4) {
    S4;
    if (e2)
      count++;
  }
  if (count == 5) {
    S5;
    count = 1;
  }
}
```

(b) After

Figure 12. Loop flattening example for multiple inner loops. These loops are not predicated only to keep the example manageable. Our loop flattening algorithm can handle multiple predicated inner loops.

[9] can only handle a single loop. But Macah programs can have multiple nested and sequenced loops. We apply a transformation that in slightly different forms has been called flattening [10], coalescing [11] and collapsing [12]. The basic idea is that the bodies of inner loops are placed directly into the outer loop, the outer loop statements are appropriately guarded and the loop induction variable calculations are patched up. In other work it is generally taken for granted that the loops involved have to be reasonably analyzable to avoid a large number of added conditional tests. However, we must flatten all loops in order to enable pipelining, so we generalized flattening to work with all kinds of loops.

We found it to be important to treat loops with a single inner loop as a special case. In this case we can avoid wrapping the body of the inner loop in additional conditional statements, which can have a big impact on the amount of control and selection logic generated. Flattening with a single inner loop is illustrated in figure 11; with multiple inner loops in figure 12. These examples show only a single level of flattening; the basic algorithm can be applied recursively to flatten more deeply nested loops.

4.8 Loop Fusion

Loop fusion [13], illustrated in figure 13 involves putting the bodies of multiple sequenced loops together into a single loop. In the sequential processor context, it is generally taken for granted that the loop bounds and increments have to match exactly for fusion to be profitable. In the accelerator context, fusing loops allows those loops to execute in parallel, so it can be profitable even if the control of the fused loops do not match exactly. Like loop flattening, we have generalized fusion to handle arbitrary loops. In the case where the control of fused loops is the same, redundancy elimination will avoid computing it more than once at runtiume. Unlike flattening, fusion cannot be applied in all cases. If there are dependences between the two loops, it may not be legal to fuse them. If fusion is legal, our compiler applies it before flattening.

```
while (1) {
  S1;
  while (1) {
    S2;
    if (e1)
      break;
  }
  S3;
  while (1) {
    S4;
    if (e2)
      break;
  }
  S5;
}
```

(a) Before

```
while (1) {
  S1;
  brk1 = 0;
  brk2 = 0;
  while (1) {
    if (!brk1) {
      S2;
      if (e1)
        brk1 = 1;
    }
    if (!brk2) {
      S4;
      if (e2)
        brk2 = 1;
    }
    if (brk1 && brk2)
      break;
  }
  S3;
  S5;
}
```

(b) After

Figure 13. Loop fusion merges two (or more) sequenced loops into one. It can only be applied if there are no blocking dependences.

4.9 Memory Accessor Streams

Memory accessor streams can be compiled into commands or "programs" for special memory interface units like direct memory access (DMA) controllers and streaming engines. This compilation process is not trivial, but because Macah programmers segregate the memory access code into memory accessor functions, it is at least clear what should be compiled this way.

4.10 Scheduling, Placement and Routing

The back-end part of the Macah compiler is a blend of software pipelining, a normal compiler backend (instruction scheduling, register allocation) and normal CAD backend (placement, routing). Accelerator architectures are highly "clustered", in the sense of clustered VLIW processors [14], so where operations are placed spatially, and the spatial routes data take from one operation to another, are important.

In the back-end, architectures are represented as collections of interconnected ALUs, registers, local memories and stream ports. Our first prototype architecture model has a relatively simple grid-style interconnection network with nearest-neighbor and longer distance connections. We expect that a wide range of accelerators can be faithfully modeled in this framework.

The back-end uses mostly conventional simulated annealing-based placement and negotiated-congestion-based routing [15, 16]. However, the placement is in both space and time–the back-end determines both which processing element an operation is executed on and during which slot in the schedule an operation is executed. Similarly, the router routes from one operation to another in different slots in the schedule. In contrast to conventional CAD flows, where registers are treated as placeable objects, in our back-end registers are treated as a routing resource, where routing through them goes from one step in the schedule to the next, as well as spatially routing from the input to the output.

The other important capability provided by the back-end is that it does automatic time-multiplexing of the kernel. Time-multiplexing is necessary if the number of resources needed by a kernel is larger than what is available in a particular architecture. It is also important if some inter-iteration dependence in the code forces the initiation interval–the rate at which loop iterations can be started–to be greater than one. The back-end handles time-multiplexing by creating multiple copies of the graph that represents the architecture, and connecting the inputs of the registers in one copy to the outputs of the same registers in the next copy. Inter-iteration dependences can dramatically limit the amount of parallelism that accelerators can exploit. The back-end reports diagnostics when such limitations occur, to help programmers understand how to improve the performance of their programs.

The back-end process outlined here is similar to that used for a variety of aggressively clustered VLIW processor and reconfigurable hardware projects, for example [17]. More details on the back-end will be published separately.

4.11 ADPC Assignment

The compiler must search for appropriate values for the ADPCs. Because the values of ADPCs can affect the sizes of data structures and the shapes of control and dataflow graphs in complex ways, ADPC choices have far-reaching impact on the rest of the compilation flow. We are going to use Macah to investigate strategies for choosing ADPC values, but we expect that: 1) building accurate analytical models of how ADPC values relate to other program characteristics will be intractable far all but the simplest programs, and 2) it is likely that programmer-supplied auxiliary code for guiding the ADPC search will be useful. Currently, the compiler simply assigns a default value to ADPCs. The functionality provided by ADPCs can be simulated with a hand-built scripting framework over the core Macah program for initial small-scale investigations. Eventually, incorporating ADPCs more deeply into the compilation process allows search algorithms to make use of internal compiler information and terminate compilation early if bad ADPC values are chosen, which is not possible with an external constant search process.

4.12 Compiling to C

The non-kernel code is translated to C with calls into special libraries for streams and shiftable arrays. This translation to C can be used on a whole Macah program, including the kernels, for early stage debugging and execution on systems without an accelerator. This translation clearly demonstrates that Macah is not less portable than C. This translation is “lossy” in the sense that it eliminates the features that make Macah efficiently compilable to coprocessors.

5. Simulation

Macah programs can currently be simulated at two levels: before and after the back-end part of the compiler runs. The front-end generates Verilog code for each of the kernels that essentially represents the body of one loop, after all the transformations have been applied. This Verilog code is structured in a simple dataflow style, where each operator fires when its inputs are ready. There is not significant parallelism to exploit at this point, because the loop controller module does not allow overlapped execution of adjacent loop iterations.

A complete program is run by compiling and executing the C code generated for the non-kernel Macah code. When the execution hits a kernel, the generated C code calls a simulation management library that forks a separate process to run the kernel in a Verilog simulator. The two processes communicate via UNIX named pipes. When the kernel simulation evaluates a kernel send or receive, it sends a message to the C side which interprets the message and executes the appropriate send or receive action.

Kernels can also be simulated after the back-end runs. The Verilog generated by the back-end is no longer in a dataflow style. It is a configuration for a simulated statically scheduled accelerator. Now, all of the operations have been given slots in a schedule and registers and routing paths have been allocated to get data from one operation to another. The whole program simulation works just as with the unscheduled kernels. The only difference from the sequential side’s perspective is that the order in which send and receive requests come from the Verilog simulator may be different.

The optimizations in the compiler are not yet sufficiently robust to generate efficient implementations. In particular, if-conversion and loop flattening generate a large amount of predicate and selection logic that can be optimized away, but is not yet. As a result, the initiation interval for the motion estimation kernel is unreasonably large. The immaturity of the back-end tool currently limits the proper simulation and scheduling, placement and routing to example applications smaller than BMME. However this is a reflection of the state of the back-end tools and not the Macah front-end. Proper simulation of the post scheduling, placement and routing for larger applications, such as BMME, is imminent, and we expect that a modest amount of compiler hacking effort will dramatically improve the efficiency.

In addition to BMME, we have worked on the simulation of both brute force and efficient implementations for other applications such as blocked dense matrix-matrix multiplication, and several permutations of FIR filters.

5.1 Architectural Models

We model accelerator architectures as collections of ALUs, registers, local memories and stream ports. Our first prototype architecture model has a relatively simple grid-style interconnection network with nearest-neighbor and some longer distance connections. The architectures we currently model have FPGA-like properties, but the compilation flow is quite flexible, and we expect that a wide range of accelerators can be faithfully modeled in this framework.

6. Related Languages

Several C-like languages for accelerator architectures have been designed and implemented. The genealogically related languages NAPA C[18], Streams-C[19] and especially Impulse C[20] are closely related to Macah. They share several concepts, like streams and pipelining accelerated blocks. There are three notable differences between the languages. The first is that Impulse C simply ignores the interaction between stream sends and receives and loop pipelining. Pipelining loops with stream sends and receives can lead to deadlock or incorrect results, if there is some external feedback. The authors of [20] observe this fact in §4.10 with little further comment. Though we have not fully formalized Macah yet, we intend to use operational semantic to prove theorems about exactly what kinds of programs have this kind of non-deterministic behavior. The informal definition of Impulse C does not permit this kind of reasoning.

The second difference is that Impulse C does not have ADPCs. The lack of ADPCs makes porting kernel code from one architecture to another more difficult.

Finally, threads in Macah are all hybrid, in the sense that any thread can enter a kernel block. In Impulse C, each process must be declared as either “software” or “hardware”. Macah’s hybrid threads can be simulated in Impulse C by a collection of processes that explicitly transfer control amongst each other with some signaling protocol. However, in Macah the tighter integration of sequential and kernel code makes programming more complex kernels easier.

Many languages designed for accelerators feature some notion of implicit data parallelism, often associated with array processing primitives. This group includes Accelerator[21], KernelC/StreamC[4], Cg[22] and StreamIt[3]. Few real applications are perfectly data parallel, but all of these languages offer features like parallel reductions and scans that allow values to be combined and distributed. ZPL[23] is a very flexible array-based language that demonstrates that the implicitly data-parallel style is quite powerful. In particular, by giving the programmer less control over the sequence of operations, implicitly data parallel languages give the compiler significant freedom to explore performance tradeoffs[24].

Even with some added features for structured sequencing, it is not clear how to program some algorithms efficiently in an implicitly data parallel style. The hexagonal search BMME algorithm is such an example. The authors of [21] were forced to use an "efficient" (not "fast") algorithm for motion estimation, because it is not clear how to program the fast one in an implicitly data parallel style.

7. Conclusion

We have designed a C-like programming language called Macah that reflects the key features of hybrid processor/accelerator systems. Macah lets programmers write efficient yet portable kernels for complex heuristic algorithms. Key language features such as streams with relaxed ordering dependencies and architecture-dependent pseudoconstants provide a flexible model for programmers without requiring heroic compiler analysis. We have used Macah to write fast, nontrivial, and portable kernels that would be difficult if not impossible to write in hardware description languages, data-parallel languages, or earlier work with more restrictive stream operations.

References

[1] C. Zhu, X. Lin, L. Chau, and L.-M. Po, "Enhanced Hexagonal Search for Fast Block Motion Estimation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 14, no. 10, pp. 1210–1214, October 2004.

[2] 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.

[3] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Computational Complexity*, 2002, pp. 179–196.

[4] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, "Programmable Stream Processors," *IEEE Computer*, vol. 36, no. 8, pp. 54–62, 2003.

[5] R. Allen and S. Johnson, "Compiling C for vectorization, parallelization, and inline expansion," in *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. New York, NY, USA: ACM Press, 1988, pp. 241–249.

[6] A. Cohen, S. Donadio, M.-J. Garzaran, C. Herrmann, O. Kiselyov, and D. Padua, "In search of a program generator to implement generic transformations for high-performance computing," *Science of Computer Programming*, vol. 62, no. 1, pp. 25–46, September 2006.

[7] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *CC '02: Proceedings of the 11th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 2002, pp. 213–228.

[8] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. New York, NY, USA: ACM Press, 1988, pp. 318–328.

[9] B. Rau, "Iterative Modulo Scheduling," HP Labs, Tech. Rep. Technical Report HPL-94-115, 1994.

[10] A. M. Ghuloum and A. L. Fisher, "Flattening and parallelizing irregular, recurrent loop nests," *SIGPLAN Not.*, vol. 30, no. 8, pp. 58–67, 1995.

[11] C. D. Polychronopoulos, "Loop Coalescing: A Compiler Transformation for Parallel Machines," in *Proc. International Conf. on Parallel Processing*, August 1987, pp. 235–242.

[12] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The structure of an advanced vectorizer for pipelined processors," in *Proc. IEEE Computer Society Fourth International Computer Software and Applications Conf.*, October 1980.

[13] K. Kennedy and K. S. McKinley, "Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution," in *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. London, UK: Springer-Verlag, 1994, pp. 301–320.

[14] P. Faraboschi, G. Desoli, and J. A. Fisher, "Clustered Instruction-Level Parallel Processors," Hewlett Packard Laboratories Cambridge, Tech. Rep. HPL-98-204, December 1998.

[15] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *ACM International Symposium on Field-Programmable Gate Arrays*. ACM Press, 1995, pp. 111–117, monterey, California, United States.

[16] S. Li and C. Ebeling, "QuickRoute: a fast routing algorithm for pipelined architectures," in *IEEE International Conference on Field-Programmable Technology*, Queensland, Australia, 2004, pp. 73–80.

[17] M. Kudlur, K. Fan, and S. Mahlke, "Streamroller: Automatic Synthesis of Prescribed Throughput Accelerator Pipelines," in *International Conference on Hardware/Software Codesign and System Synthesis*, October 2006.

[18] M. Gokhale and J. Stone, "NAPA C: Compiling for Hybrid RISC/FPGA Architecture," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.

[19] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 49–56.

[20] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*. Prentice Hall PTR, April 2005.

[21] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: simplified programming of graphics-processing units for general-purpose uses via data-parallelism," Microsoft Corporation, Tech. Rep. MSR-TR-2004-184, December 2005.

[22] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: a system for programming graphics hardware in a C-like language," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 896–907, 2003.

[23] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, L. Snyder, W. D. Weathersby, and C. Lin, "The Case for High-Level Parallel Programming in ZPL," *IEEE Comput. Sci. Eng.*, vol. 5, no. 3, pp. 76–86, 1998.

[24] S. Agrawal, W. Thies, and S. Amarasinghe, "Optimizing stream programs using linear state space analysis," in *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*. New York, NY, USA: ACM Press, 2005, pp. 126–136.