1.0 Overview

This document explains writing Imagine applications to a reader familiar with single processor programming techniques and generalized vector processors. A brief overview of the Imagine system from the applications programming perspective is presented, along with the Imagine programming model and tools. A simple example on vector addition follows, accompanied by explanation. An on-line version of this example can be found in im_apps/vect_add. The steps required to integrate an application into the Imagine simulator are then described. Finally, additional details of programming, either not encountered in the example or not explained in detail, are covered with the help of another example: fft. An on-line version of this example can be found in im_apps/fft_new. Please note that this documentation is at a preliminary stage and hence will updated on a regular basis. For the same reason, many sections may look incomplete. Corrections, comments, and suggestions on this document are always welcome and should be sent to abhishek@cva.stanford.edu.

1.1 Imagine

Imagine is a programmable single-chip processor that supports the stream-programming model. Imagine provides a three tiered storage bandwidth hierarchy consisting of a streaming memory system, a large (128KB) stream register file, and direct forwarding of results among arithmetic units. Eight clusters of parallel arithmetic units process data retrieved from the shared stream register file. Each cluster includes three adders, two multipliers, one divide/square unit, one 256-entry local scratch-pad register file and one inter-cluster communication unit (see figure 1 in *The Imagine instruction Set Architecture*). The stream register file acts as a buffer between the arithmetic clusters and Imagine's main memory.

Imagine is a coprocessor that is programmed at two levels: kernel and application. A kernel is a small program that runs on the arithmetic clusters of Imagine, and is repeated for each successive element in the input streams to produce output streams (which may be the input stream for the next kernel in the application). Kernels are coded in a programming language called kernelC, using the expression syntax of the C language. Kernels may access local variables, read input streams, and write output streams, but may not make arbitrary memory references. Kernels are compiled into microcode programs that sequence the units within the arithmetic clusters to carry out the kernel function on each stream element in turn. A separate memory space is reserved for storing the microcode programs, which the clusters execute in parallel. The clusters can only obtain data from the stream register file, and can only retrieve instructions from the microcode store. An Imagine application is a set of kernels connected by streams. The application level programming is done using streamC language, which uses the syntax of C++ language.

While the Imagine stream processor executes a microcode assembly language (μasm), the general-purpose processor (residing in host) executes a streamC code, complied in its native instruction set. The microcode carries out the bulk of the computation, by executing kernels in the clusters, whereas the streamC program manages the overall operation with the help of Imagine's stream controller, by initializing the stream register file,

setting up the memory for the microcode, directing the stream processor to execute it, and then storing back its result to main memory.

1.2 Imagine programming model and tools

The StreamC language, having close resemblance to C++, is used for writing programs that utilize the Imagine stream processing system to operate on streams (series of elements). StreamC includes commands for transferring streams of data to and from the Imagine system and between Imagine processors, for defining control and data flow between kernels and for executing kernels (essentially calling a series of kernels).

The kernels, written in kernelC, are functions that operate on streams by looping over streams (operating on a single element at a time). Random data access is not allowed and a limited amount of control flow exists.

Details of the StreamC and KernelC language specifications are available in section 4 and section 3 respectively, of ips_user.pdf.

Since the Imagine chip hasn't been built yet, we have to run Imagine applications under a simulator program, isim. This program can be supplied with a series of commands either through the keyboard or from a file that tell it to load and store memory to and from files and to execute applications. There are four available Imagine programming tools:-

1. **Kernel scheduler (iscd):** Single-phase VLIW scheduler, which is optimized for individual kernels by handling functional unit (cluster) assignment and communication scheduling between the clusters and modulo-software pipelining. It generates the microcode for Imagine.

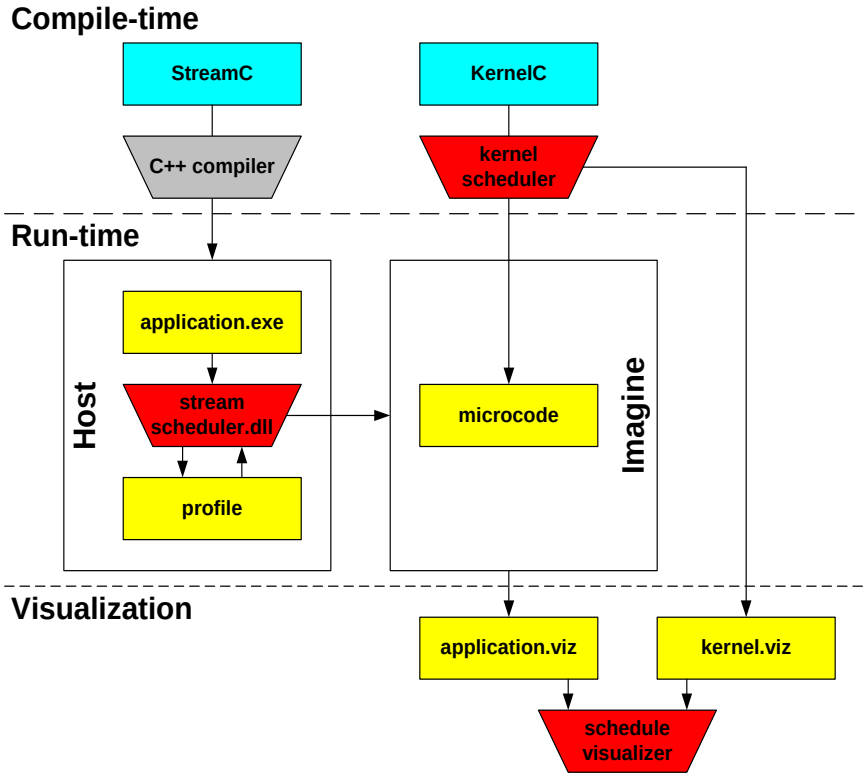
   The VC++ preprocessor converts the kernelC file into a **.i** (*intermediate*) file, which the KernelC compiler actually compiles to produce a **.uc** (*microcode*) file. This code is sent to Imagine at the start of the application. It sits in Imagine memory until it is needed, at which point it is loaded into the on-chip microcode store.

2. **Stream scheduler (istream):** Converts StreamC functions into Imagine operations. It determines the allocation of the Stream Register File (SRF), handles large streams (using Strip-mining or Double-buffering), resolves dependencies between operations and performs other such high-level optimizations. Stream scheduler is profile based, running once with simple allocation, collecting usage information, performing good allocation and running repeatedly with good allocation.

   The streamC program runs on the host. When it calls a kernel, a bunch of high-level operations are sent to Imagine, which load the input streams, execute the kernel, and save the output streams. The stream scheduler (istream) generates these operations such that all of the streams can fit in the SRF at the same time. It allocates space in the SRF for all streams that pass through the SRF regardless of where they come from or go. Ideally, an application sends some initial input to Imagine from the host, and Imagine does a lot of processing on it, keeping in the SRF when possible and the off-chip memory if its too big, and then sends back only the final results.

3. **Schedule visualizer (schedviz):** Used for application and kernel visualization. Enables one to visualize SRF allocation among different streams and the kernels operating on them.

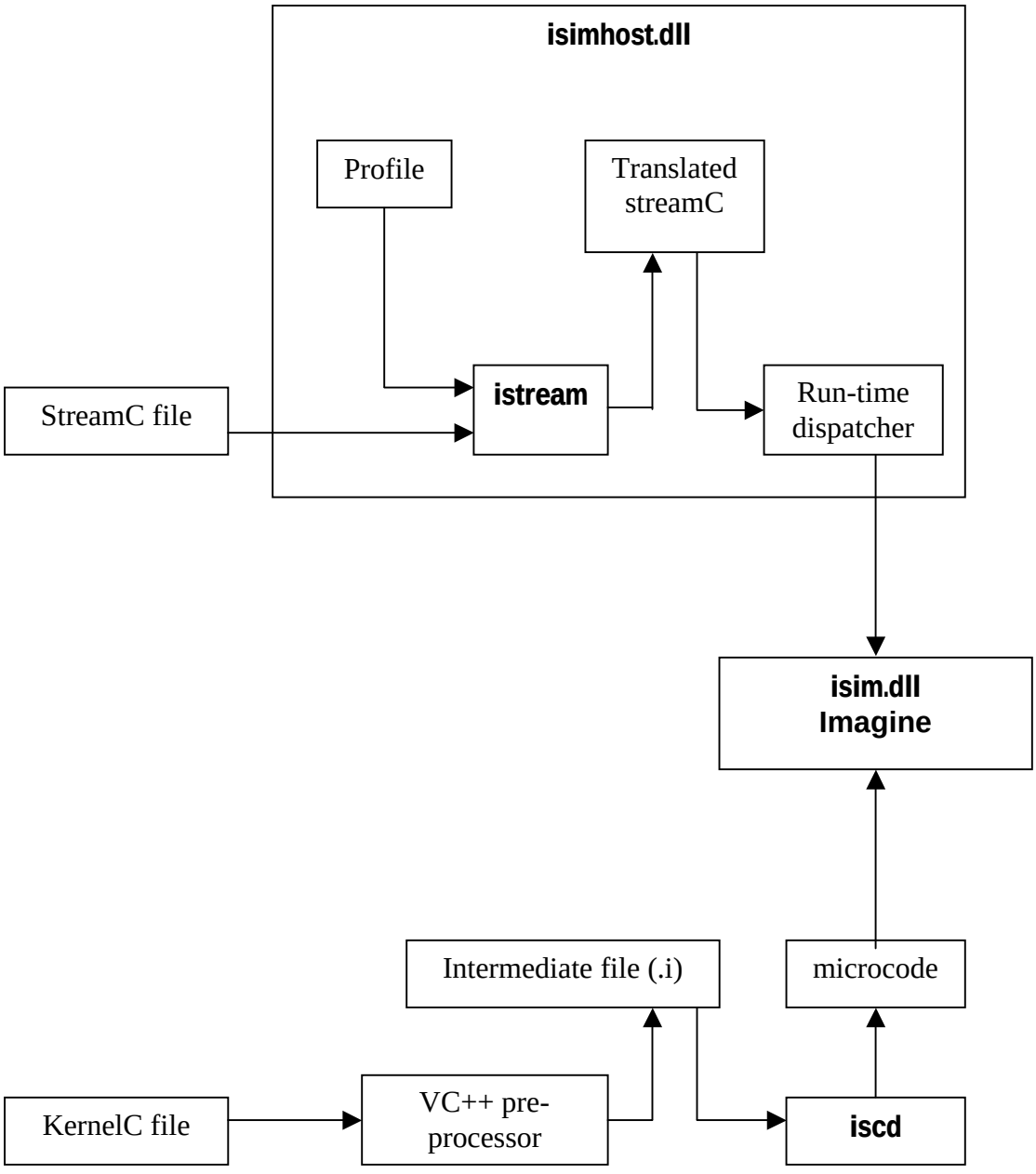
4. **Interactive debugger (idebug):** Provides debug functionality for program development. This only simulates functionality to test the working of streamC and kernelC.



Any Imagine application runs using two DLLs:

- **isimhost.dll** -- run-time linked library that contains stream scheduler. With real hardware, it would run on the host processor.
- **isim.dll (** or **isimcore.dll)** -- run-time linked library that simulates the Imagine processor.

The interaction between the different features are shown in the following figure:



Before we attempt to learn about writing applications, we must first get the code for all these tools and create the DLLs.

1.2.1 Getting the Most Recent tools (and other files)

For people in Stanford: You should use Sourcesafe to get these. The //smorgasboard/programs/README.PROGRAMS file describes how to install Source Safe and Visual C++, which are required to build the Scheduler and isim tools. One important point to note is that these programs require that Visual C++, Service Pack 2 or Service Pack 5 be used. Any other service pack, even Service Pack 3, may cause build errors. Once Visual C++ and Source Safe are installed, use Source Safe to download the most recent version of tools. Select all the files in the /working directory and press CTRL+G (Get Most Recent Version). Source Safe may prompt you for a working directory, in which case you can provide c:\working or d:\working, which is a directory on your hard drive, that should be automatically backed up periodically. In the rest of the document, this working directory will be referred to as <your working dir>.

For people outside Stanford: You should get the archive in /pub/imagine from our ftp server. Try doing ftp cva.stanford.edu and then cd /pub/imagine.

1.2.2 Using Visual C++ to Compile the Tools

1. Open tools.dsw file using Visual C++ by going to <your working dir>\tools (eg. D:\working\tools or C:\working\tools). This is a workspace file that will compile the tools for you, which are included in it as projects.
2. We need to build the following projects: isimdll, isimhostdll, isimexe and iscd. Let us first get the project settings correct. The steps needed are as follows (steps indicate where your mouse button should click):

- Add the following to your system path (using Settings > Control Panel > System):

  <your working dir>\tools\isim\isimexe\Release

  <your working dir>\tools\isim\isimexe\Debug

  <your working dir>\tools\iscd\Release

- Add the following environment variables (using Settings > Control Panel > System):

  iscd_preproc = C:\Program Files\Microsoft Visual Studio\VC98\Bin\CL.EXE

  iscd_includes = <your working dir>\tools\isim\isimexe\blank_headers

  These are required to provide "C++ preprocessor support" to handle the #includes, #defines, etc. in your source files.

- Project settings > Link > Customize (pulldown) > Output file name.

  Fill-up in order of the projects mentioned above (i.e isimdll, isimhostdll, isimexe and iscd):

  Release:

  ..\isimexe\Release\isimcore.dll

..\isimexe\Release\isimhost.dll
Release\isim.exe
Release\iscd.exe

Debug:
..\isimexe\Debug\isimcoredbg.dll
..\isimexe\Debug\isimhostdbg.dll
Debug\isim.exe
Debug\iscd.exe

- Project Settings > Link > General > Output file name
  Fill-up the same settings in the same order.

- Project Settings > Link > General > Object/Library modules
  Only for isimhostdll and isimexe (in order):

  Release:
  isimcore.lib
  isimcore.lib isimhost.lib

  Debug:
  isimcoredbg.lib
  isimcoredbg.lib isimhostdbg.lib

3. Now, we build the projects in the following order: isimcore.dll > isimhost.dll > isim.exe (due to the dependency) and iscd. (Instead of building iscd, you can use <your working dir>\im_apps\scd.bat)

Now you are ready to write an aplication.

2.0 Writing application (vector addition used as example)

As in every other program, first you need to figure out the inputs, outputs and the algorithm to be used and after that writing an Imagine application is as easy as 1-2-3-4:

1. Write a shared header file declaring the records that need to be used and the kernel declarations. As in C, treat the kernels as function calls.
2. Write a StreamC program to implement everything except the computationally intensive portion of the problem: it should just declare the streams (inputs and outputs), setup the stream register file, call the appropriate kernels (for the required computation), and store the result in most cases.
3. Write the kernel in kernelC to carry out the actual computation (the algorithm).
4. Write a simulator script to load test data into memory and store the final result to disk.

The steps are explained below with the help of a vector addition example.

2.1 Shared header file

The file naming convention is <name>.hpp (eg. vect_add.hpp). This file consists of the records to be used and the kernel declarations. Write the header file according to the syntax given in section 2.3.1 of ips_user.pdf.

Since the kernel runs on the Imagine processor, we should obviously put as much of the computational load there as possible, leaving the general purpose processor to mundane tasks such as transferring data between main memory and the stream register file for processing. So, let us declare a kernel called vadd, which takes two streams of vectors as inputs and produces a stream of vectors by adding the corresponding vectors in the input streams. Moreover, Imagine being a stream-based architecture, the kernel here should not just add two individual vectors, but instead should accept two streams of many vectors, add them, and output a single stream of the resulting vectors. We will consider a 4-variable (floating point) vector. We will call our vector as vvector, which is defined using record as shown below:

```
#ifndef VECT_ADD
#define VECT_ADD

#include "idb_types.hpp"
#include "idb_deftypes.hpp"

record vvector {
    float x, y, z, w;
};
// vector declaration

kernel vadd(istream<vvector> in_1, istream<vvector> in_2,
            ostream<vvector> out);
```

```
KERNELDECL(vadd);
// kernel declaration

#define vadd KERNELCALL(vadd)
//defines the kernel call

#include "idb_undeftypes.hpp"
#endif
```

2.2 streamC program

The file naming convention is <name>_sc.cpp (eg. vect_add_sc.cpp). The syntax is very much like C++ apart from the imagine basic types and special functions to operate on streams. Now that we've decided what functionality the kernel is going to give us, we can write the streamC code to call it with the required streams. The input streams are assumed to be available in files im_apps\vect_add\vector1.vec and vector2.vec. In order to keep things simple, we take a stream of 8 vectors. The streamC code is as follows:

```
#include "idb_streamc.hpp"
#include "vect_add.hpp"
//shared files

STREAMPROG(vect_add);
// defining stream program

void vect_add(StreamSchedulerInterface& scd, String args)
{
  if(args == ""){
  cout << "Sorry, have to pass a string to do example");
  }

  // A simple vector addition example
  else if(args == "doExample"){

  // Load the first stream of vectors
  // declare a stream of that size first
  im_stream<vvector> input1 = newStreamData<vvector>(8);
  streamLoadFile("vect_add\vector1.vec", "txt", "", input1);

  // Load the second stream of vectors
  // declare a stream of that size first
  im_stream<vvector> input2 = newStreamData<vvector>(8);
  streamLoadFile("vect_add\vector2.vec", "txt", "", input2);

  // declare output stream
  im_stream<vvector> data_out = newStreamData<vvector>(8);
```

```
  // Print a message telling that we're about to run the kernel
  cout << "Beginning computation." << endl;

  // Call the kernel to do the computation.
  vadd(input1, input2, data_out);

  // Print a message telling we’ve finished running the kernel
  cout << "Finished computation." << endl;

  // save and verify final output data
  streamSaveFile("vect_add\output.vec", "txt", "E", data_out);
  streamCompareFile("vect_add\add.vec", data_out, 0.001f, "a");
 }
}
```

The first thing we notice is that the streamC subroutine accepts a string argument. This string can be set from the simulator command line in the simulator script (shown later). To call our vector addition example, we set this string to "doExample". We could easily add other if...else if clauses to respond with different actions to different strings (commands).

Looking into the code for the addition, we see that the first thing it does is declare the input streams load the two sets of vectors into the stream register file. Remember that the arithmetic clusters (the processors on which the kernel executes) can only access the stream register file and their own internal registers, but not the main memory space of the Imagine board.

Let us look more closely into the streamSaveFile and streamCompareFile syntax:-

streamSaveFile(file, type, args, in1)

"*file*" (the file to save into)

"*type*" (there's a bunch of types: "txt", "bin", "binPtr", "pgm", "ppm", "pnm", "raster", "rtl" :- mostly you'll be interested in "txt")

"*args*" (they depend on what type you have, but for "txt", you probably want "d" (decimal), "X" (hex), or "E" (float))

"*in1*" (the stream to save)

streamCompareFile(scd, file, in1, threshold, args)

"*file*"(the comparison file)

"*in1*"(the stream to compare against)

"*threshold*"(how close does it have to match?This is a float)

"*args*"(must be \"a\" for absolute comparison, \"r\" for relative comparison, \"rle\" for run length encoded comparison, or \"mitre\" for the mitre comparison)

Thus, streamSaveFile is used to save the output generated while streamCompareFile is used to test the contents of the output generated against the expected output contained in "vect_add\add.vec". We allow the differences a tolerance of 0.005 (0.5%); if the differences were above our indicated tolerance then an error message prints out.

Though in the stream files we used have *.vec* extension, they are essentially text files (look them up in im_apps\vect_add). Hence, we use the type as *txt* and use *"a"* for absolute comparision.

2.3 kernelC program

The file naming convention is <name>_kc.cpp (eg. vect_add_kc.hpp). The syntax is very much like in C++, apart from the new Imagine basic types just like in StreamC. However the control flow is different. We can now examine the kernel that carries out the addition of the vectors:

```
#include "idb_kernelc.hpp"

//shared files
#include "vect_add.hpp"
#include "idb_kernelc2.hpp"

KERNELDEF(vadd, "vect_add\vect_add.uc");
// The microcode file to be generated

kernel vadd(istream<vvector> in_1, istream<vvector> in_2,
            ostream<vvector> out)
{
  loop_stream(in_1)
  // loop till all the elements of stream in_1 are not
  // exhausted (since in_1 and in_2 are stream of same length,
  // both are exhausted after the loop.
  {
   vvector v0, v1, v2;
   // These variables exist in the cluster's register space
   // there are separate copies of these on each cluster

   in_1 >> v0;
   in_2 >> v1;
   // Read in one vector from stream in_1 and another from
   // stream in_2. This operation is functionally equivalent to
   //  reading each component separately.

   v2.x = v0.x + v1.x;
   v2.y = v0.y + v1.y;
   v2.z = v0.z + v1.z;
   v2.w = v0.w + v1.w;
```

```
    // Add the two vectors
     out << v2;
    //put the result in the output stream.
   }
 }
```

Notice that there are no if statements or conditional jumps in this code. From the code, it appears that the kernel operates on a single record at one time. Actually, each of the 8 clusters on Imagine run the same kernel on different elements of the stream simultaneously. The Kernel Scheduler compiles KernelC code into VLIW instructions. Thus, under this architecture, each cluster always executes the same instruction, though on a different data. If conditional jumps occurred, different arithmetic units could end up trying to execute different instructions at the same time.

So, the only control structures available are loops. The kernels loop on a stream till it completes operating on all the elements of a stream. The exact syntax for a loop and the various allowable options are described in section 3 of ips_user.pdf).

The next set of operations retrieve data from the stream register file into the registers. It is important to realize that all clusters share the input streams and hence a read instruction such as "in_1 >> v0" will read 8 elements out of the stream (recall that there are 8 clusters in Imagine). The elements are distributed among the clusters in a round robin fashion and are executed upon at the same time (SIMD operation). This makes reading data more complicated than on a single cluster machine. Take the case of reading the stream of vectors from in_1 : a, b, c, d, e, f, g, h in our code. Intuitively, we'd think of memory being structured as follows:

TABLE 1. Non-Interleaved Stream Layout

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010
a.x	a.y	a.z	a.w	b.x	b.y	b.z	b.w	c.x	c.y	c.z
1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021
c.w	d.x	d.y	d.z	d.w	e.x	e.y	e.z	e.w	f.x	f.y

But when we execute a command like, "in_1 >> v0", where v0 is a variable stored in an arithmetic unit's internal register file, v0 will assume in_1.x as a.x on cluster 0, but cluster 1's copy of in_1.x will assume a.y, and cluster 2's copy of in_1.x will be set to a.z. Clearly, this is not the intended result.

In order to correct this problem, we must interleave the data as we transfer it from main memory to the stream register file, so the SRF will be organized as follows:

TABLE 2. Interleaved Stream Layout

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010
a.x	b.x	c.x	d.x	e.x	f.x	g.x	h.x	a.y	b.y	c.y
1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021
d.y	e.y	f.y	g.y	h.y	a.z	b.z	c.z	d.z	e.z	f.z

So, memory is now organized as 8 x coordinates, followed by 8 y coordinates, followed by 8 z coordinates, etc. Since there are 8 clusters, a stream read will now result in cluster 0 getting a.x, cluster 1 getting b.x, etc. The next stream read will get cluster 0 a.y, cluster 1 a.y, etc. We now get the expected result. A single iteration of the loop of the vector addition routine will add 8 vectors in parallel, with each cluster doing a single vector addition. Thus, for 16 vector additions, 2 iterations of the loop are necessary. Since the stream write operations are also executed in parallel, the output stream will be interleaved in the same format as the input stream.

Now, we are presented with the problem of converting the vector data from its "natural" representation (x, y, z, w) to the interleaved representation described above. This could easily involve some complex data manipulation, if it weren't for the fact that streamLoadFile and streamSaveFile can do this automatically from a file written orderly. It is clear that if we only had three vectors and tried to interleave them as in table 1, we would have to pad each component with 5 zeros in order to allow an 8-cluster read to stay aligned. The usual solution for the case where we want to process a number of records that is not a multiple of the machine cluster size is to pad the data with null elements to the correct length.

Now, let us look at how the input files are written for streamLoadFile (streamCompareFile and streamSaveFile also follow the same format. The files consist of one or more regions. Each region starts with a ":", followed by a "T" as the first two characters on the line, and then a space (':T '). The next character is a sscanf formatting character, which indicates the format of the numbers in the following lines. Thus, if you look at vect_add\vector1.vec, which consists of floating point numbers represented in an exponential form, the formatting character is "E". Also, an optional number can come right after the formatting character (no intervening space), to indicate that the number of bytes required to represent the numbers. Thus we use a '2' for 16-bit and a '4' for 8-bit numbers. streamCompareFile checks each 8-bit or 16-bit values.to the correct length.

You may find that the output file generated by streamSaveFile has more than one column. In such cases, the file is read row-wise, i.e first we read the entries (going from left to right) in the the first row and then the second row and so on.

2.4 Simulator script (.sim file)

The last thing we need to write is a simulator script to exercise the program. This script consists of a series of commands to the simulator. A good analogy is:

csh shell : shell scripts :: isim simulator : .sim files

So, instead of typing all the commands in the shell, a .sim file can be written. A script usually loads some test data into the main memory, calls the kernel, and then stores the result to disk for examination. The script for our example is shown below. The commands are indicated by "isim> " prompt, the explanations are preceded by "//', while isim's response are preceded by "> ". Note that we haven't yet learned how to open the shell, but the explanations here have been included to make things easier to understand.

```
isim> t im
```

// Instantiates an Imagine simulated processor

```
isim> p /it/im/
```

// This is the 'prefix' command. Since all pieces of the Imagine simulator are instantiated as modules in a hierarchical manner, the "p" command is used to navigate to different levels of the heirarchy. "/it/im" is, in some sense, the "root directory" of the Imagine processor, which we're simulating. isim responds to the 'prefix' command with:

```
> prefix: /it/im/

isim> read txt ./ms/data "vect_add\vector1.vec" 0x0000

> Read 8 values.

isim> read txt ./ms/data "vect_add\vector2.vec" 0x0020

> Read 8 values.
```

// Read in the stream of vectors into memory at memory address 0x0000 and 0x0020 respectively. The input file with our data set is in text format ("txt") and it is in source file "vect_add\vector1.vec" and "vect_add\vector2.vec" respectively (files are accessed relative to the current working directory). The length of the read is determined by the length of the source file, and the simulator responds to the read command with the length read. The reads initialize Imagine memory, the destination for these input data set being our memory system's data ("./ms/data").

NOTE: When the Stream Scheduler encounters a streamLoadFile it doesn't do any host transfers, rather just looks up the address of the initialized data based on the file name.

NOTE: If we had not called "p /it/im" earlier, the read command would have had to look like this:

```
isim> read txt im/it/ms/data "vect_add\vector2.vec" 0x0020
```

// Now we run our vadd kernel:

```
isim> run vect_add ../hp "doExample"
```

// "vadd" names a kernel instance, which in turn maps to the kernel we called "vadd" defined in vect_add_kc.cpp. The kernel will run on the host processor ("../hp", or alternatively, "/im/hp") and take arguments "doExample". Then we kick off the kernel with:

```
isim> go

> [     1] Imagine Starting

> ...
```

//"go" runs the kernel to completion; it prints out progress messages, and by default the simulator prints its cycle number every 10000 cycles (Use "go" with an argument to change that, try "help").

// Finally, if there are errors they'd be contained in the error printlog:

```
isim> printlog error
```

// At the end we quit ISIM.

```
isim> q
```

NOTE: Details on the individual commands can be obtained by typing "help" or "help <command>" at the ISIM command line.

2.5 Debugging applications and extracting useful statistics

ISim is actually a cycle accurate simulator, and hence can be used to gather performance results. For debugging purpose, debug_info allows the programmer to look at the register state within isim. Like other modules, debug_info can be viewed using the "d" (display) command in isim. There are two types of debug_info modules. Each cluster has a debug_info module of its own that stores register state in that module. The cluster array also has an "umbrella" module that has pointers to each of the cluster modules. For details on debug_info, please refer to section 8.3 in ips_user.pdf.

In order to retrieve useful statistics, stats allows us to get instruction counts and percentages on a per-function-unit basis. It has the same structure as debug_info; it's a module, and displaying it shows the stats it's compiled. Like debug_info, it can be printed from one of two places: in the cluster from an umbrella module (each cluster will work, but they should all be the same) or from a functional unit. For details on stats, please refer to section 8.4 in ips_user.pdf.

I will have an example to explain both in the next version.

3.0 Integrating a Program Into the Simulator

Now that we have the various program modules implemented, we must integrate them into the simulator before we actually attempt to execute them.

1. Make a directory under <your working dir>\im_apps\ <your application>, eg. D:\working\im_apps\vect_add
2. Make sure that the input files (to be loaded into streams) and the files for stream comparisions are in the correct directory.
3. Add the following to your system path (using Settings > Control Panel > System):

   <your working dir>\tools\isim\isimexe\Release

   <your working dir>\tools\isim\isimexe\Debug
4. Follow the directions given in section 2.2 of ips_user.pdf under the heading '*Creating a project*'. The last step can be skipped by instead copying the main.cpp file, since it is the same for every application. Make sure the rest of the source files are added into the project.
5. Build the project by specifying working directory (project settings) as <your working dir>\im_apps and generate <name>.exe. In the example taken, it is vect_add.exe.
6. iscd (KernelC compiler) compiles <name>_kc.cpp files to produce <name>_kc.uc files, the microcode that can be executed by Imagine (simulated by isim). The command to generate <name>_kc.uc file is:

   iscd -m gold8.md <name>_kc.cpp

   which in the example taken will look as follows:

   iscd -m gold8.md vect_add\vect_add_kc.cpp

   Note that you need to be in the directory <your working dir>\im_apps to run iscd. If you are using scd.bat, you also need to append the -pre flag to support preprocessing, as follows:

   ./scd.bat –i vect_add_kc.cpp -pre

   This generates the following files:- vect_add_kc.uc, vect_add_kc.i and vect_add_kc.viz, which are the microcode, the intermediate file (inline C source code) and the visualization file respectively. Use the following comamand to view the .viz file:

   i:\tools\schedviz.exe
7. iscd and isim requires a machine description file as an argument to run an application. The -m argument specifies the machine that we're compiling the microcode for. The application programmer doesn't have to worry about the contents of the machine description file, with the one exception of knowing the number of clusters on the machine, which is necessary for correctly handling interleaving. 'gold8.md' is the required machine description file, which is a description of the entire Imagine machine - what functional units there are, what functions they run, how long functions take, how big the SRF is, the speed of the memory, all the latencies between units, etc. (open and read to find more details). Add the machine description file in program arguments (project settings) in the given format:

   –m gold8.md

8. If you try to run <name>.exe now, a shell opens with the following prompt:

[x] isim>

Here, x denotes the current simulation time. Now type help to get the list of all commands. You can type "help <commandname>" for more specific info. Use them in accordance with need of application. Since we have already decided on the commands we are interested in the .sim file, we add this in the program argumnets. The program arguments (in project settings) will now look as:

–m gold8.md –s <name of file>.sim

9. Now run <name>.exe, which in our example will be vect_add\vect_add.exe.

Another way to achieve all this is to not include the program arguments, but use the foolowing command:

<name>.exe –m gold8.md –s <name of file>.sim

10. Read this only if you need to use Source Safe. After verifying that the simulator still compiles and doesn't crash when executed, we should use Source Safe to integrate our changes. This will both allow others in the group to share in the fruits of our labor and preserve our changes for later use:

Add the vect_add directory to the im_apps directory in Source Safe. This operation is not entirely intuitive. First, select the im_apps directory. Next, choose "Add Files" from the File menu. Use the directory navigation bar on the right of the dialog box to open the vect_add folder. You should now see the files it contains in the left partition of the dialog box. Finally, click on the Add button to complete the operation. You should now see the vect_add folder in Source Safe. Note that you should not include the output files (eg. .uc, .i, .viz, .opt etc.) since they become read-only files now and hence if the program is run again, it may fail to generate these output files.

4.0 Additional Topics

NewStreamData restrictions: The size argument to new streamData must be a constant, and data-dependent control flow cannot be used to allocate more than one chunk of stream data at the same time. The following are not legal:

1. im_stream<Foo> out = newStreamData(in.getLength());
2. im_stream<Foo> a[100];
   for_VARIABLE(int i = 0; ...) {
   a[i] = newStreamData(10);
   }

Data-dependent streams: Data-dependent streams involve noting stream derivations for which the start and/or end values, which change depending on data. This is done using the data dependence field, where one puts "true" for a variable length stream, and some combination of the following flags:

im_var_size: The start is fixed and the end is data dependent but known when the stream is derived. This requires the stream to be a derived stream.

im_var_countup:The start is fixed and the end is data dependent, determined by the number of output elements produced by a stream operation. Note that it still needs an upper bound on its length.

im_var_pos:Both the start and end are data dependent.

im_var_align: Used in combination with im_var_pos, but it is assumed that the start is always divisible by SRF block sizes.

*im_var_cover:*Used in combination with im_var_pos, when a write or read is treated as an access to the entire stream.

The following examples illustrate the use of the mentioned flags:

1. Variable size streams are used most often to contain the output of kernel that consumes a stream with a data-dependent number of elements and produces the same number of elements. Since the size of the output is known, we can derive the output stream using *im_var_size* as follows:

   ```
   im_stream<Foo> out_data = newStreamData<Foo>(100);
   im_stream<Foo> out = out_data(0, in.getLength(), im_var_size);
   myKernel(in, out);
   ```

2. Countup (or variable length) streams are used whenever the end varies depending on the number of elements produced by a stream operation. Note that all countup streams are inherently variable size streams. Taking the previous example, if we now consider a case when the kernel doesn't produce the same number of elements as the stream it consumes, the data-dependent nature of the input stream needs the output stream to be derived with *im_var_countup*:

```
im_stream<Foo> out = newStreamData<Foo>(maxFoo, im_countup);
myKernel(in, out);
```

Note that *maxFoo* was specified to provide upper bound on the stream length.

3. A variable position stream is used whenever the start varies, most often to iterate over parts of a stream with data-dependent length or do some sort of lookup into a particular stream:

```
im_stream<Foo> out = newStreamData<Foo>(maxFoo);
for_VARIABLE(i = 0; i < in.getLength(); i+= 32) {
  im_stream<Foo> inIter = in(i, max(i + 32, in.getLength()),
                             im_var_pos | im_var_align | im_var_cover);
  im_stream<Foo> outIter = out(i, max(i + 32, in.getLength()),
                             im_var_pos | im_var_align | im_var_cover);
  myKernel(inIter, outIter)
 }
```

In the case of this example, the im_var_align flag can be used because the start is always divisible by the SRF block size of 32. This flag is important for good performance because it allows multiple variable position accesses to the same buffer in the SRF.

Further, the im_var_cover flag can be used because the loop accesses the entire stream even though each iteration does not. Hence, if the stream out can fit in the SRF then a use of out after the loop does not need to load any of the stream from memory.

Data-dependent control flow: Marking data-dependent control flow requires replacing if with if_VARIABLE (note that else is not supported) and while or for with while_VARIABLE or for_VARIABLE, if the condition is data-dependent. For example:

```
while_VARIABLE(a.getLength() > 0) {
    ...
 }
```

- A stream with a derivation that varies between iterations of a data-dependent (but not a fixed) loop is also data-dependent. For example:

```
for_VARIABLE(i = 0; i < a.getLength() ; i+= 100) {
    im_stream<Foo> b = a(i * 100, i * 100 + 100, im_var_pos);
 }
```

- Data dependent streams and control-flow are the only data-dependent variations allowed in a profile. In particular, the following is not legal:

```
im_stream<Foo> b = newStreamData(100);
```

```
im_stream<Foo> c = newStreamData(1000);
kernelFoo(a, a.getLength() > 100 ? c : b);
```

- But this is:

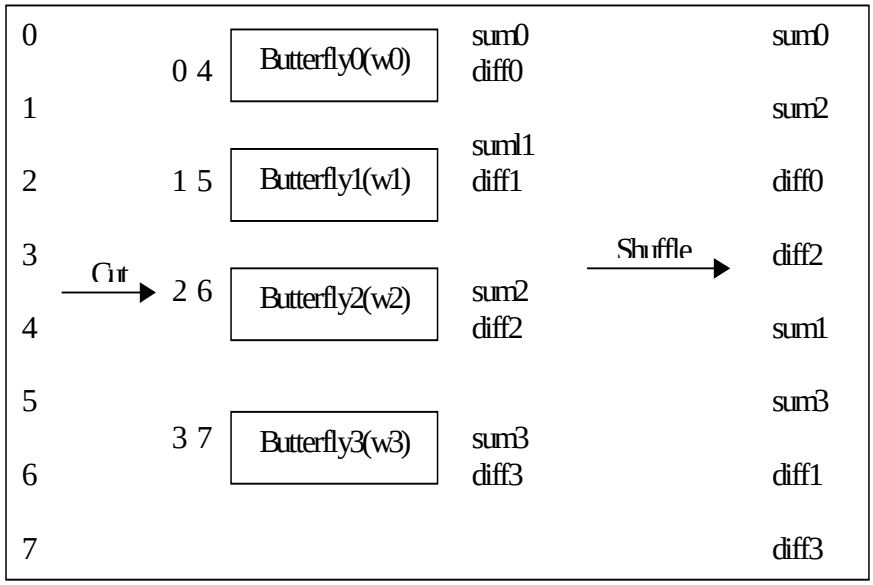
```
im_stream<Foo> b = newStreamData(100);
im_stream<Foo> c = newStreamData(1000);
if_VARIABLE(a.getLength() <= 100) {
    kernelFoo(a, b);
 }
if_VARIABLE(a.getLength() > 100) {
    kernelFoo(a, c);
 }
```

Using type qualifier DOUBLE: We are all aware of the fact that data types have their own size limitations. Hence an *int* data type can't be always used to store the result of a multiplication of two variables of *int* data types. **DOUBLE<type>** is a type qualifier that can be used to achieve the same. It allows the user to concatenate two instances of the qualified type, and then access any of the high word or low word, using **HI** and **LO** respectively. Taking an example, let *a* and *b* are the two multiplicative operands and we are interested in storing the low word of the result in *c*, all of which are of data type *int*. This can be achieved as follows:

```
DOUBLE<int> d;
d = a * b;
c = LO(d);
```

5.0 A more involved example: fft

Let us now try to implement the radix-2, perfect shuffle, decimation in frequency butterfly algorithm in our kernels. For the forward fft, an input stream of ordered elements is taken as input and the output stream produced is in bit-reversed order. The fft for a N-input stream can be considered to be a result of log2(N) butterfly stages, where each stage does N/2 butterfly operations and the index of inputs to each butterfly gets shuffled between successive stages. The operation of a single stage is illustrated below for a 8-element stream.



We first divide, or cut, the input stream into 2 stacks, feed them into the 4 butterflies and then then shuffle them to create the correct fft stage pairings. The output of this stage will be fed to the next stage, which will perform the same operation. After $\log_2(N)$ such stages, we would achieve our desired fft output. This scenario can easily be incorporated in Imagine, where the kernel performs 8 butterfly operations in each stage, one each in a cluster. The shuffling operation can be achieved by taking advantage of inter-cluster communication. Though the butterfly and shuffling operation remains the same at every stage, the inputs and twiddle factors change. Note that the inputs to a stage are the results of the previous stage and hence pose no problem. However, we need the stream of twiddle factors to provide the correct factors to each cluster as the stages change.

The header file for fft_new:

```
#include "idb_types.hpp"
#include "idb_deftypes.hpp"
record complex {
  float r;
  float i;
```

```
};
kernel fft8c(istream<complex> in_a, istream<complex> in_b,
             istream<complex> in_w,
             uc<int>& uc_stage_num, uc<int>& uc_tw_idx_inc,
             ostream<complex> out);
KERNELDECL(fft8c)
#define fft8c KERNELCALL(fft8c)
#include "idb_undeftypes.hpp"
#endif
```

The streamC file, named fft_new_sc.cpp.

```
#include "idb_streamc.hpp"
//shared files
#include "fft_new.hpp"
STREAMPROG(fft_new)

unsigned int log2( unsigned int x ){
  unsigned int i = 0;
  x >>= 1;
  while( x ) { x >>= 1; ++i; }
  return i;
 }

void fft_new(StreamSchedulerInterface& scd, String args){
  // load twiddle factors
  im_stream<complex> twiddle_factors =
  newStreamData<complex>(344,im_countup);
  //im_stream<complex> twiddle_factors;
  streamLoadFile("fft_new/twiddle8c1024.vfft", "txt", "",
                 twiddle_factors);
  // load input data
  im_stream<complex> data_in =
                    newStreamData<complex>(1024,im_countup);
  //im_stream<complex> data_in;
  streamLoadFile("fft_new/input.vfft", "txt", "", data_in);
  // compute some useful values
  int len = data_in.getLength();
  int half_len = len / 2;
  int pow2_len = log2(len);
  cout << "***** len = " << len << endl;
  cout << "***** half_len = " << half_len << endl;
  cout << "***** pow2_len = " << pow2_len << endl;
  // declare data output
```

```
  im_stream<complex> data_out;
  // loop
  int i;
  int tw_inc = 1;
  im_uc<im_int> uc_i;
  im_uc<im_int> uc_tw_inc;
  for (i = 0; i < pow2_len; i++){
    // output goes somewhere new
    data_out = newStreamData<complex>(len);
    uc_i=i;
    uc_tw_inc=tw_inc;
    // call kernel
    fft8c(data_in(0, half_len), data_in(half_len, len),
          twiddle_factors, uc_i, uc_tw_inc,
          data_out(0, len, im_acc_stride));
    // output becomes next input
    data_in = data_out;
    tw_inc = tw_inc * (i < 4 ? 1 : 2);
  }
  // save and verify final output data
  im_stream<complex> fin_out = data_out(0, len, im_fixed,
                                        im_acc_bit_reverse);
  streamSaveFile("fft_new/output.vfft", "txt", "E", fin_out);
  streamCompareFile("fft_new/bitrev.vfft", fin_out,
                    0.005f, "a");
}
```

One needs to ensure that the size passed to newStreamData is constant, thereby requiring to know the number of elements in the files, to be used in streamLoadFile. For example, there are 668 twiddle factors in the file twiddle8c1024.vfft and hence the size of the stream twiddle_factors should be made 668 during stream declaration.

Also, there is a provision for specifying loop unrolling and number of pipelined stages. Both these quantities have been taken as 1 in the fft_new_kc.cpp (kernelC file for fft_new, which can be found in im_apps/fft_new) to make matters simpler. The file has not been shown because of its sheer size.

Taking our example, the fft8c1024.sim file will look as following:-

```
t im
p /it/im/
read txt ./ms/data "fft_new/twiddle8c1024.vfft" 0x0
read txt ./ms/data "fft_new/input.vfft" 0x0
run fft_new ../hp "8c 1024"
go
```

```
vcmp ./ms/data "fft_new/bitrev.vfft" 0x2800 0.005
printlog error
```

There are 10 stages in our scenario since there are 1024 elements ($\log_2 1024=10$). This is why the loop in streamC file runs for 10 iterations, calling the kernel to perform the computation required in the current stage *i* (also passed to kernel). Each stage takes 2 input streams of 512 elements each, thus requiring 512 twiddle factors for the same number of butterflies computed. The twiddle factors required in each stage are as follows:-

1st -> $w^0,w^1,w^2,w^3,w^4,w^5,w^6,w^7,w^8.....,w^{511}$

2nd -> $w^0,w^0,w^2,w^2,w^4,w^4,w^6,w^6,w^8.....,w^{510}$

3rd -> $w^0,w^0,w^0,w^0,w^4,w^4,w^4,w^4,w^8.....,w^{508}$

4th -> $w^0,w^0,w^0,w^0,w^0,w^0,w^0,w^0,w^8.....,w^{504}$

5th -> w^0(16 times),w^{16}(16 times),w^{32}(16 times),.....,w^{496}(16 times)

6th -> w^0(32 times),w^{32}(32 times),w^{64}(32 times),.....,w^{480}(32 times)

7th -> w^0(64 times),w^{64}(64 times),w^{128}(64 times),.....,w^{448}(64 times)

8th -> w^0(128 times),w^{128}(128 times),w^{256}(128 times),.....,w^{384}(128 times)

9th -> w^0(256 times),w^{256}(256 times)

10th -> w^0(512 times)

Instead of loading a different stream of twiddle factors in each stage, we use the same stream of twiddle factors. Moreover, we don't load all the 512 required twiddle factors; instead interpolate the required factors from the given/loaded values. The stream of twiddle factors are divided into 3 categories, one following the other (can be verified by looking into twiddle8c1024.vfft). They are shown below, seperated ni groups of 8, to illustrate the values used by each of the 8 clusters:

32 x 8 entries: w^0(8 times),w^{16}(8 times),w^{32}(8 times),.....,w^{496}(8 times)

These are read into *twiddle_real* and *twiddle_imag* in the kernel.

10 x 8 entries:

1st stage -> $w^0,w^1,w^2,w^3,w^4,w^5,w^6,w^7$

2nd stage -> $w^0,w^0,w^2,w^2,w^4,w^4,w^6,w^6$

3rd stage -> $w^0,w^0,w^0,w^0,w^4,w^4,w^4,w^4$

4th stage -> $w^0,w^0,w^0,w^0,w^0,w^0,w^0,w^0$

5th stage -> $w^0,w^0,w^0,w^0,w^0,w^0,w^0,w^0$

6th stage -> $w^0,w^0,w^0,w^0,w^0,w^0,w^0,w^0$

7th stage -> $w^0,w^0,w^0,w^0,w^0,w^0,w^0,w^0$

8th stage -> $w^0,w^0,w^0,w^0,w^0,w^0,w^0,w^0$

9th stage -> $w^0,w^0,w^0,w^0,w^0,w^0,w^0,w^0$

10th stage -> w^0,w^0,w^0,w^0,w^0,w^0,w^0,w^0

These are read into *rotate_real* and *rotate_imag* in the kernel.

8 entries: w^8,w^8,w^8,w^8,w^8,w^8,w^8,w^8

This is read into *interp_rotate* in the kernel.

Let us now look at how the twiddle factors are generated at each stage. As can be seen clearly, the *rotate_real* and *rotate_imag* twiddle factors reflect the base twiddle factors required in each stage. We only have to multiply these with the right factor to generate all the twiddle factors required in a stage. This is achieved by using *twiddle_real* and *twiddle_imag*. The point to note is that there is no w^8 there. This is achieved by using *interp_rotate*. So, to summarize, the twiddle factors used are generated iteratively as follows (*real* and *imag* have not been explicitly shown):

twiddle * *rotate* : 0-7, 16-23,.....,496-503th twiddle factors

twiddle * *interp_rotate* * *rotate* : 8-15, 24-31,....,504-511th twiddle factors

However, you can see that there are 2 problems in using this expression. First, w^8 is not required after the 4th stage. To achieve this, the kernel uses *interp*, which is *interp_rotate* for the first 4 stages, and is 1 for the rest of the stages. Thus, the second expression is *twiddle* * *interp* * *rotate*.

Secondly, we cannot simply use *twiddle* after the 4th stage because we don't need all the *twiddle* entries. While the 5th stage requires all entries, the 6th stage requires every alternate entry (w^0,w^{32},w^{64} and so on), the 7th stage requires every 4th entry (w^0,w^{64},w^{128} and so on) and so on. This indexing is a geometric progression. For this reason, the kernel introduces 2 new flags: *tw_idx* and *tw_idx_inc*. The streamC program performs the geometric progression and passes on the current index to the kernel. Thus, if we look at *tw_inc* in the streamC code at each stage, they are found to take the following values in each iteration (stage):

1st -> 1 2nd -> 1 3rd -> 1 4th -> 1 (immaterial in first 4 stages)

5th -> 2^0

6th -> 2^1

7th -> 2^2

8th -> 2^3

9th -> 2^4

10th -> 2^5

The other interesting thing in the kernel is the shuffle part. Note that the output generated by the butterflies at each stage will be ordered in a bit-reversed manner. But since we are feeding the same output to the kernel as the next input, we need to make sure that the output is not in the bit-reversed order. This is taken care of by inter-cluster communication in the kernel, using *commucperm*. However, we want the final output (last stage) to be in bit-reversed order. This is taken care of by the streamC, using *im_acc_bit_reverse* to derive the final stream.

6.0 IScd and IDebug

As mentioned earlier, **IScd**, the kernel scheduler, is used to compile kernels for execution on the Imagine processor. All kernels must be compiled before using the functional simulator, IDebug, to create a profile, or using the cycle-accurate simulator. When we run iscd, it displays which scheduling pass it is executing, a progress indicator when actually scheduling operations, and other important information directly to the command line. In order to get details about the arguments iscd uses and the various information it displays during the scheduling pass, it is highly recommended to refer to section 5 of ips_user.pdf.

IDebug is a functional simulator built into isimhost.dll and isimcore.dll. It includes a set of classes and functions that allow the direct execution of a stream application without simulated or actual Imagine hardware, i.e even without scheduling kernels. It can be used in conjunction with a debugger such as that built into Visual C++ to debug a stream application. All of the conventional debugging tools can be used: breakpoints, single-stepping instructions, watches, stack traces, etc. This definately motivates one to learn more about using IDebug, which can be found in section 7 of ips_user.pdf.