DSPL: An Overview

Jeffrey Cropsey – jdc2103@colmbia.edu
David Lariviere – dal2103@columbia.edu
Michael Lynch – mtl2103@columbia.edu
Varun Maithel – vm2117@columbia.edu
Varun Mehta – vkm2101@columbia.edu

I. Introduction

Digital Signal Processing Language was created and designed for the purpose of providing an efficient language that can easily be compiled to make use of vector instructions that have been introduced in Intel's recent instruction sets. The language receives its name from the strong applicability these improvements have to many operations common in digital signal processing.

Since the introduction of Intel's MMX technology processor families, Intel has introduced four extensions into their architectures that support single-instruction multiple-data (SIMD). These extensions provide a group of instructions that perform SIMD operations on packed integer and/or packed floating-point data elements. Using these instructions enhances the performance of compatible processors for a variety of uses, including advanced 2-D and 3-D graphics, motion video, image processing, speech recognition, audio synthesis, telephony, and video conferencing.[1]

II. Brief Overview of Intel SIMD

Intel first introduced SIMD with MMX, which was available on some Pentiums as well as Pentium II's. MMX offered instructions for parallel operations on packed byte, word, or double word integers.

SSE (Streaming SIMD Execution) was first introduced in Pentium III, offering instructions for packed single-precision floating-point values.

SSE2 was introduced in the Pentium 4 and Intel Xeon processors, offering instructions for packed double-precision floating-point values, as well as new instructions for 128-bit integer operations.

SSE3 was most recently introduced in the Pentium 4 supporting Hyperthreading (P4+HT), which further added additional instructions, some of whose utility are demonstrated in Section XI.

DSPL will specifically target P4+HT and later architectures, in order to take full advantage of all SIMD instructions (MMX, SSE, SSE2, and SSE3).

---

[1] *IA-32 Intel® Architecture Software Developer's Manual*:
ftp://download.intel.com/design/Pentium4/manuals/25366520.pdf

III. Vector Operations

The main feature that DSPL offers is its ability to take advantage of SIMD (Single-instruction Multiple-Data) instructions when compiled. These instructions allow for the parallel execution of multiple operations that would otherwise require separate instructions. The DSPL compiler uses these instructions when the language performs operations on arrays. For example, if `a`, `b`, and `c` are arrays of type `int` with an arbitrary number of elements, in DSPL it is possible to perform the following operation:

```
a = b + c; //where a[0] = b[0] + c[0], a[1] = b[1] + c[1], etc
```

The compiler uses SIMD instructions when creating assembly code for this statement, such that several additions can be performed simultaneously. In prior languages a `for` loop would be required, which would add each of the elements individually, resulting in many more instructions. Further, if the array sizes are known at compile time, it is possible to unroll the loop, further improving speed. Although this may seem like a rather simple improvement, in the applications noted above these operations are performed frequently and consequently optimization of this process has a dramatic overall effect.

IV. Portability

A driving motivation behind DSPL is its use of all available SIMD instructions available on current-generation Intel processors. Therefore, DSPL has been designed to offer superior efficiency on modern systems (supporting SSE3), rather than worry about backwards compatibility with previous architectures. It is, however, conceivable to port the DSPL compiler to other architectures.

V. Complex Numbers

As complex arithmetic is prevalent in digital signal processing, a new native type, `complex`, has been introduced, for storing two floating-point values representing the real and imaginary components of a complex number.

VI. Data Types

As the focus of DSPL language is to increase ease and efficiency of vector calculations, DSPL uses a small set of data types. The data types implemented are `int`, `float`, `complex`, and `string`, as well as the single dimension array types `int[]`, `float[]`, and `complex[]`.

VII. Operators

DSPL uses operators in very powerful ways. It implements the basic mathematical operators `+`, `-`, `*`, `/` (addition, subtraction, multiplication, and division), which are capable of operating on every data type (including single-dimension arrays) with the exception of `string`. It also introduces a strictly-array operation of convolution (represented by ~). The use of array operators leads to cleaner, more elegant code that is less prone to mistakes by the programmer and more easily optimized by the compiler.

VIII. Control Keywords

The language supports a basic set of control characters needed for manipulating sets of data in calculations, consisting of `for`, `while` and a conditional structure `if – else`.

IX. Built-in Functions

In order to provide a simple and clean way for a DSPL programmer to work with sets of data, the language includes the following built in functions:

`print`	outputs data to standard output
`read`	reads data from standard input

X. Syntax Example

```
int scale;
complex[2] inputA;     //2 element complex array
complex[2] inputB;
complex[2] result;
read(inputA);          //read in first 2 complex #s
read(inputB);          //read in second 2 complex #s
result = inputA * inputB;
if (result[0].r <= 1) {
   scale = 1;
}
else {
   scale = 2;
}
result *= scale;  //multiply all values by scale

print(result);
```

The above sample program would print out the value of result, which should contain two complex numbers, the first representing inputA[0] * inputB[0] * scale, and the second

equal to inputA[1] * inputB[1] * scale, where scale would be 1 if the real part of result[0]<=1, else 2 if the real part of result[0]>1.

XI. Comparison between C and DSPL

A subsection of the DSPL program given in Section X, the multiplication of two pairs of complex numbers, could be equivalently coded in C as follows:

```
typedef struct {
  float real;
  float imag;
} ComplexFloat;

void main() {
  ComplexFloat multiplicand[4]; //contains four complex floats
  ComplexFloat product[2]; //will store the product of two pairs of complex floats

  multiplicand[0].real = 1.34;
  multiplicand[0].imag = -3.7;
  multiplicand[1].real = 0.4;
  multiplicand[1].imag = 7.83;

  multiplicand[2].real = 834;
  multiplicand[2].imag = -8.8;
  multiplicand[3].real = -0.4;
  multiplicand[3].imag = 7.7;

  product[0].real = (multiplicand[0].real * multiplicand[1].real) - (multiplicand[0].imag * multiplicand[1].imag);
  product[0].imag = (multiplicand[0].real * multiplicand[1].imag) + (multiplicand[0].imag * multiplicand[1].real);

  product[1].real = (multiplicand[2].real * multiplicand[3].real) - (multiplicand[2].imag * multiplicand[3].imag);
  product[1].imag = (multiplicand[2].real * multiplicand[3].imag) + (multiplicand[2].imag * multiplicand[3].real);
}
```

Compiling the above program with gcc 3.4.4 yields the following x86 assembly for the actual floating point multiplications:

```
flds    -40(%ebp)       #load multiplicand[0].real
fmuls   -32(%ebp)       #multiply multiplicand[0].real * multiplicand[1].real
flds    -36(%ebp)       #load multiplicand[0].imag
fmuls   -28(%ebp)       #multipliy multiplicand[0].imag * multiplicand[1].imag
fsubrp  %st, %st(1)     #subtract from first multplication the second
fstps   -56(%ebp)       #store the real for the first complex result
flds    -40(%ebp)       #load and calculate imag for the first complex result
fmuls   -28(%ebp)
flds    -36(%ebp)
fmuls   -32(%ebp)
faddp   %st, %st(1)
fstps   -52(%ebp)       #store the imag for the first complex result
flds    -24(%ebp)       #load and calculate the real for the second complex result
fmuls   -16(%ebp)
flds    -20(%ebp)
fmuls   -12(%ebp)
fsubrp  %st, %st(1)
fstps   -48(%ebp)       #store the real for the second complex result
flds    -24(%ebp)       #load and calculate the imag for the second complex result
fmuls   -12(%ebp)
flds    -20(%ebp)
fmuls   -16(%ebp)
faddp   %st, %st(1)
fstps   -44(%ebp)       #store the imag for the second complex result
```

The equivalent functionality, taking advantage of Intel's SSE3 introduced in the Pentium4+HT platform is as follows[2]:

```
movsldup xmm0, [eax]        ; multiplier real (a1, a1, a0, a0)
movaps xmm1, [ebx]          ; multiplicand (d1, c1, d0, c0)
mulps xmm0, xmm1            ; temp1 (a1d1, a1c1, a0d0, a0c0)
shufps xmm1, xmm1, 0xB1     ; shuf multiplicand(c1, d1, c0, d0)
movshdup xmm2, [eax]        ; multiplier imag (b1, b1, b0, b0)
mulps xmm2, xmm1            ; temp2 (b1c1, b1d1, b0c0, b0d0)
addsubps xmm0, xmm2         ; b1c1+a1d1, a1c1-b1d1, a0d0+b0d0, a0c0-b0c0
movaps [edx], xmm0          ; store the results (y1,x1,y0,x0)
```

Note that the gcc-generated assembly is shown using AT&T assembly syntax, where operands are reversed from those in the Intel style assembly format directly above.

This simple example demonstrates the extreme reduction in the number of instructions that need to be executed to perform equivalent functionality, in this case the multiplication of two pairs of complex numbers.

[2] *Using SSE3 Technology in Algorithms with Complex Arithmetic:*
http://www.intel.com/cd/ids/developer/asmo-na/eng/66717.htm