SIMT through SIMD

GPU behavior with CPU vector

Author: José Moreira

jmoreira@us.ibm.com

Table of Contents

About This File

This file was created to illustrate one approach to emulating GPU behavior on CPUs that have vector architecture. It is adapted from material used by the author while teaching CSEE 4824 (Computer Architecture) at Columbia University during the Spring 2023 term.

SECTION 1

Vector architecture

1.1 The architecture

This is a n-way SIMT architecture. In GPU nomenclature, it supports the execution of n-thread warps. We do not specify the number of warps that can be simultaneously executed. That would be a function of the number of cores available and the simultaneous multi-threading degree of each core. There is a unified $128 \times n$-byte register file ($\mathsf{R}\,[0:127]\,\langle 0:8n-1\rangle$). The n SIMT threads share the unified register file. An 8-bit data element in each thread is represented as a single register. Instructions on 8-bit data elements get translated into a single operation.

For example, and add-byte instruction

$$\mathsf{addb}(\mathsf{RT},\mathsf{RA},\mathsf{RB}) \tag{1}$$

gets translated into an add-byte operation

$$\mathsf{addb}(P(\mathsf{RT}),P(\mathsf{RA}),P(\mathsf{RB}),\mathsf{AV}\,\langle 0:n-1\rangle) \tag{2}$$

where $P(\mathsf{Rx})$ is the physical register mapped to architected register Rx and $\mathsf{AV}\,\langle 0:n-1\rangle$ is the activation vector (mask) for the n SIMT threads.

Correspondingly, and add-word instruction

$$\mathsf{addw}(\mathsf{RT}, \mathsf{RA}, \mathsf{RB}) \tag{3}$$

gets translated into four add-word operations

$$\mathsf{addw}(P(\mathsf{RT}+0), P(\mathsf{RA}+0), P(\mathsf{RB}+0), \mathsf{AV}\left\langle 0 : \frac{n}{4}-1\right\rangle) \tag{4}$$

$$\mathsf{addw}(P(\mathsf{RT}+1), P(\mathsf{RA}+1), P(\mathsf{RB}+1), \mathsf{AV}\left\langle \frac{n}{4} : \frac{n}{2}-1\right\rangle) \tag{5}$$

$$\mathsf{addw}(P(\mathsf{RT}+2), P(\mathsf{RA}+2), P(\mathsf{RB}+2), \mathsf{AV}\left\langle \frac{n}{2} : \frac{3n}{4}-1\right\rangle) \tag{6}$$

$$\mathsf{addw}(P(\mathsf{RT}+3), P(\mathsf{RA}+3), P(\mathsf{RB}+3), \mathsf{AV}\left\langle \frac{3n}{4} : n-1\right\rangle) \tag{7}$$

since it takes $4 \times n$-byte registers to represent $n \times 32$-bit words.

The supported data types are shown in Table 1. All arithmetic operations, including compares, are qualified with the data type. For example,

$$\mathsf{add[b,h,w,l,s,d,c,z]}(\mathsf{RT}, \mathsf{RA}, \mathsf{RB}) \tag{8}$$

For simplicity, let $\mathsf{[t]} \equiv \mathsf{[b,h,w,l,s,d,c,z]}$.

suffix	type	width	registers per instance
b	byte	8-bit	1
h	halfword	16-bit	2
w	word	32-bit	4
l	long	64-bit	8
s	single-precision	32-bit	4
d	double-precision	64-bit	8
c	single complex	64-bit	8
z	double complex	128-bit	16

Table 1: Data types supported by the architecture.

1.1.1 Branch facility instructions

Conditional branch facility instructions (Table 2) are implemented through manipulation of the SIMT activation vector. Unconditional branches are implemented directly, without modifying the activation vector.

Conditional	beq T	branch to T if equal
	bne T	branch to T if not equal
	bgt T	branch to T if greater than
	blt T	branch to T if less than
	bge T	branch to T if greater than or equal
	ble T	branch to T if less than or equal
Unconditional	b T	branch to T
	bl T	branch to T and link
	blr	branch to link register

Table 2: Branch instructions.

1.1.2 Fixed- and floating-point instructions

Fixed- and floating-point arithmetic instructions have the general form shown in Table 3. Compare instructions are similar, but update the flags register instead of a target register. In the table, RT, RA, and RB define collections of between 1 and 16 registers, depending on the element size (see Table 1).

add[t](RT,RA,RB)	$\mathsf{R[RT]} \leftarrow \mathsf{R[RA]} + \mathsf{R[RB]}$
sub[t](RT,RA,RB)	$\mathsf{R[RT]} \leftarrow \mathsf{R[RA]} - \mathsf{R[RB]}$
mul[t](RT,RA,RB)	$\mathsf{R[RT]} \leftarrow \mathsf{R[RA]} \times \mathsf{R[RB]}$
div[t](RT,RA,RB)	$\mathsf{R[RT]} \leftarrow \mathsf{R[RA]} \div \mathsf{R[RB]}$
addi[t](RT,RA,IM)	$\mathsf{R[RT]} \leftarrow \mathsf{R[RA]} + \mathsf{IM}$
subi[t](RT,RA,IM)	$\mathsf{R[RT]} \leftarrow \mathsf{R[RA]} - \mathsf{IM}$
muli[t](RT,RA,IM)	$\mathsf{R[RT]} \leftarrow \mathsf{R[RA]} \times \mathsf{IM}$
divi[t](RT,RA,IM)	$\mathsf{R[RT]} \leftarrow \mathsf{R[RA]} \div \mathsf{IM}$

Table 3: Arithmetic instructions.

1.1.3 Addressing facility

Pointers specify both a single address and a region of the address space. Pointers are always 128-bit global quantities represented in one register. A 128-bit pointer in register RA consists of three fields: (i) a 48-bit N; (ii) a 48-bit field A; and a 32-bit field σ.

$$\mathsf{R[RA]}\langle 0:127\rangle \equiv N\langle 0:47\rangle \parallel A\langle 0:47\rangle \parallel \sigma\langle 0:31\rangle \tag{9}$$

The address in the pointer is directly encoded in field A. The region is specified as follows:

1. The size (number of bytes) of the region is given by field N.
2. The *alignment* α of the region is the smallest power of 2 greater than or equal to the size:
$$\alpha = 2^{\lceil \log_2 N \rceil}. \tag{10}$$
3. The *base* B of the region is defined by
$$B = \left\lfloor \frac{A}{\alpha} \right\rfloor \times \alpha. \tag{11}$$
4. The region consists of all addresses in the range $[B, B+N)$.
5. σ is a *cryptographic signature* that guarantees authenticity of the pointer. One cannot manufacture pointers. The signature will not match. Authentic pointers to a new region of the address space can only be obtained through system services.
6. You can subset a region. For example, let the following pointer identify the region for a 64×64 matrix of doubles:
$$\mathsf{R[RA]} = \mathrm{0x8000} \parallel \mathrm{0x1000\ 0000} \parallel \sigma_1 \tag{12}$$
If the matrix is stored in column-major order, then this is the region for column 16 of the matrix:
$$\mathsf{R[RA]} = \mathrm{0x0200} \parallel \mathrm{0x1000\ 2000} \parallel \sigma_2 \tag{13}$$
It has a different signature.

The addressing facility is responsible for subsetting regions and verifying authenticity of pointers.

1.1.4 Load/store instructions

Load/store instructions specify the data type and have the form shown in Table 4.

l[t](RT,RA,RB)	load one element of type t in each active thread
st[t](RS,RA,RB)	store one element of type t in each active thread

Table 4: Load and store instructions.

For load/store instructions, RA is a pointer and RB identifies four registers (RB+0,RB+1,RB+2,RB+3) containing n words.

The load instructions load n elements of the specified type, one for each thread. The address for thread i is computed as follows:

$$i \in [0, n) \tag{14}$$

$$j = i/4 \tag{15}$$

$$k = i\%4 \tag{16}$$

$$\mathsf{EA}(i) = \mathsf{R}[\mathsf{RA}].A + (\mathsf{R}[\mathsf{RB} + j]).\mathrm{word}[k] \tag{17}$$

Every $\mathsf{EA}(i)$ must belong to the address range defined by $\mathsf{R}[\mathsf{RA}]$.

The store instructions store n elements of the specified type, one for each thread. The addresses are computed as for load instructions.

RT and RS define collections of between 1 and 16 registers, depending on the element size.

1.2 Control-flow

We will focus on two forms of structured control-flow: if-then-else statements and while statements.

```
[L] if (RA op RB)
    {
      ⟨then code⟩
    }
    else
    {
      ⟨else code⟩
    }
```

```
[L] while (RA op RB)
    {
      ⟨loop body⟩
    }
```

In the above, L is a label we will use in the assembly code and op $\in$ {eq, ne, gt, lt, ge, le}. Also the condition **if** (RA) is equivalent to **if** (RA $\neq 0$).

Before we show how to translate those constructs to assembly code, let us define the flags register for our processor. Each thread gets a 4-bit field: one bit for each of the EQ, GT, and LT flags, and one bit of spare for future use. Let $T(i)$.flags denote this 4-bit field for thread i. We concatenate the fields for the n threads we have into a $4n$-bit F register:

$$\mathsf{F} = T(0).\mathsf{flags} \parallel T(1).\mathsf{flags} \parallel \ldots \parallel T(n-1).\mathsf{flags} \tag{18}$$

We then define the $8n$-bit AVF register as the concatenation of the n-bit AV register, the $4n$-bit F register, and a padding of $3n$ zero bits:

$$\mathsf{AVF} = \mathsf{AV}\langle 0 : n-1\rangle \parallel \mathsf{F}\langle 0 : 4n-1\rangle \parallel 0\langle 0 : 3n-1\rangle \tag{19}$$

The AVF register can be saved and restored in the stack using the pushavf and popavf instructions.

With this model, we can translate the if-then-else and while statements as follows:

```
              pushavf
              cmp(RA, RB)
              $\overline{\text{bop}}$(ELSE : L)
              ⟨then code⟩
ELSE : L      popavf
              pushavf
              bop(END : L)
              ⟨else code⟩
END : L       popavf
```

```
BEGIN : L     pushavf
              cmp(RA, RB)
              $\overline{\text{bop}}$(END : L)
              ⟨loop body⟩
              b(BEGIN : L)
END : L       popavf
```

A conditional branch instruction bop(L) can be implemented with the following sequence of operations:

$$\mathsf{TMP} \leftarrow [T(0).\mathsf{flags.op}, T(1).\mathsf{flags.op}, \ldots, T(n-1).\mathsf{flags.op}] \tag{20}$$

$$\mathsf{AV} \leftarrow \mathsf{AV} \wedge \overline{\mathsf{TMP}} \tag{21}$$

$$\textbf{if } (\mathsf{AV} = 0)\ \mathsf{b}(L) \tag{22}$$

where $T(i)$.flags.op is the result of applying operation op to the flags of thread i.

1.3 STRMV - single-precision triangular matrix vector multiply

The main function for STRMV is shown in Figure 1. The uplo, trans, and diag arguments produced 8 different variants of matrix-vector multiply. For each variant, we apply a different kernel to the iteration space $i \in [0, n)$. Figure 2 shows the C++ kernel for uplo = 'U', trans = 'N', and diag = 'N'. That C++ code translates to the assembly code in Figure 3.

Whenever a computation is performed *in place*, like for the vector x of STRMV, care must be taken when scheduling the kernel in the iteration space. After all, you do not want to modify an element of x before you are done using it. In our case, the matrix can be either lower or upper diagonal and each case requires a different schedule.

Let us take the case of $x \leftarrow Ux$:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \leftarrow \begin{bmatrix} u_{0,0} & u_{0,1} & u_{0,2} & u_{0,3} \\ & u_{1,1} & u_{1,2} & u_{1,3} \\ & & u_{2,2} & u_{2,3} \\ & & & u_{3,3} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{23}$$

In this case, it is safe to compute x in place if we proceed through the iteration space in *forward* order, because:

1. x_0 is only used to compute x_0;
2. x_1 is only used to compute x_0 and x_1.
3. x_2 is only used to compute x_0, x_1, and x_2.
4. x_3 is only used to compute x_0, x_1, x_2, and x_3.

Now let us take the case of $x \leftarrow Lx$:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \leftarrow \begin{bmatrix} l_{0,0} & & & \\ l_{1,0} & l_{1,1} & & \\ l_{2,0} & l_{2,1} & l_{2,2} & \\ l_{3,0} & l_{3,1} & l_{3,2} & l_{3,3} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{24}$$

In this case, it is safe to compute x in place if we proceed through the iteration space in *backward* order, because:

1. x_3 is only used to compute x_3;
2. x_2 is only used to compute x_2 and x_3.

3. x_1 is only used to compute x_1, x_2, and x_3.

4. x_0 is only used to compute x_0, x_1, x_2, and x_3.

```
void strmv
(
    char        uplo,   // 'U' or 'L'
    char        trans,  // 'N' or 'T'
    char        diag,   // 'U' or 'N'
    u32         n,      // problem size
    float       *A,     // A matrix
    u32         ldA,    // leading dimension of A
    float       *x,     // x vector
    i32         incx    // increment of x
)
{
    if (uplo == 'U')
    {
        if (trans == 'N')
        {
            if (diag == 'N')
                Kernel(strmv_unn)[n](n, A, ldA, x, incx);
            else
                Kernel(strmv_unu)[n](n, A, ldA, x, incx);
        }
        else
        {
            if (diag == 'N')
                Kernel(strmv_utn)[n](n, A, ldA, x, incx);
            else
                Kernel(strmv_utu)[n](n, A, ldA, x, incx);
        }
    }
    else
    {
        if (trans == 'N')
        {
            if (diag == 'N')
                Kernel(strmv_lnn)[n](n, A, ldA, x, incx);
            else
                Kernel(strmv_lnu)[n](n, A, ldA, x, incx);
        }
        else
        {
            if (diag == 'N')
                Kernel(strmv_ltn)[n](n, A, ldA, x, incx);
            else
                Kernel(strmv_ltu)[n](n, A, ldA, x, incx);
        }
    }
}
```

Figure 1: Main function of STRMV.

```
void strmv_unn(u32 n, float *A, u32 ldA, float *x, i32 incx)
{
    u32 i = simt::i();
    float s = 0;
    u32 j = i;
    while (j < n)
    {
        s = s + A[i + j*ldA]*x[j*incx];
        j++;
    }
    x[i*incx] = s;
}
```

Figure 2: Kernel (in C++) strmv_unn of STRMV.

```
void strmv_unn
(
    u32          n,     // (R03, R04, R05, R06)
    float       *A,     // (R07)
    u32          ldA,   // (R08, R09, R10, R11)
    float       *x,     // (R12)
    i32          incx   // (R13, R14, R15, R16)
)
{
            simti(r20);           // i (r20, r21, r22, r23) = simt::i()
            muli(r13, r13, 4);    // adjust incx to bytes
            muli(r08, r08, 4);    // adjust ldA to bytes
            mul(r24, r20, r13);   // (r24, r25, r26, r27) = i*incx
            zfs(r28);             // s (r28, r29, r30, r31) = 0
            addi(r32, r20, 0);    // j (r32, r33, r34, r35) = i
            mul(r40, r32, r08);   // j * ldA
            add(r40, r40, r20);   // i + j * ldA
            muli(r40, r40, 4);    // i + j * ldA in bytes
            pushavf();            // save AV
begin_j:    cmp(r32, r03);        // while (j < n)
            bge(end_j);
            lfs(r36, r12, r24);   // (r36, r37, r38, r39) = x[j * incx]
            lfs(r44, r07, r40);   // (r44, r45, r46, r47) = A[i + j*ldA]
            mul(r44, r44, r36);   // A[i + j*ldA]*x[j*incx]
            add(r28, r28, r44);   // s = s + A[i + j*ldA]*x[j * incx]
            add(r40, r40, r08);   // advance i + j * ldA
            add(r24, r24, r13);   // advance j * incx
            addi(r32, r32, 1);    // j++
            b(begin_j);           // next j
end_j:      popavf();             // restore AV
            blr();                // return
}
```

Figure 3: Kernel (in assembly) strmv_unn of STRMV.