H1119 - Introduction to AltiVec - Ten easy ways to Vectorize your code
What is a Vector Architecture?

- A vector architecture allows the simultaneous processing of multiple data items in parallel.
- Operations are performed on multiple data elements by a single instruction.
  - Referred to as Single Instruction Multiple Data (SIMD) parallel processing.

![Diagram of vector architecture with operations on multiple data elements]
What is AltiVec?

• **SIMD** extension to **PowerPC** Architecture
  – … no tradeoffs … just additions
• **Provides a high-performance RISC microprocessor with DSP-like compute power**
  – Allows highly parallel operations for the simultaneous execution of up to 16 operations in a single clock cycle
• **Offers a programmable solution for controller and signal processing functions**
  – …which can easily migrate via software upgrades to follow changing standards and customer requirements
AltiVec’s Vector Execution Unit

- Concurrent with PowerPC integer and floating-point units
- Separate, dedicated 32 128-bit vector registers
- Approximately 11% of the silicon area
- No penalty for mixing integer, floating point and AltiVec operations
SIMD Intra-element Instructions

Diagram showing the execution of SIMD intra-element instructions with operations (op) on VA, VB, VC, and VT.
AltiVec Instruction Set Features

• **162 new instructions added to the PowerPC ISA**
  – Intra and inter-element *arithmetic* instructions
  – Intra and inter-element *conditional* instructions
  – Powerful Permute, Shift and Rotate, Splat, Pack/Unpack and Merge instructions

• **4-operand, non-destructive instructions**
  – Up to three source operands and a single destination operand
  – Supports advanced “multiply-add/sum” and permute primitives

• **All instructions fully pipelined with single-cycle throughput**
  – Simple ops: 1 cycle latency
  – Compound ops: 3-4 cycle latency
  – No restriction on issue with scalar instructions
Enabling AltiVec in your applications

• Let Compiler do the job
  – Currently there are no practical Autovectorizers out-there
  – It is a guess how much performance will be actually extracted
  – and they still require certain expertise to work with
• Using C intrinsic
  – There is a standardized list of intrinsics supported in all PowerPC enabled compilers
  – Can actually guarantee good level of control, up to the level of register assignment
• Using Assembly Language programming
  – The most effective, and the most laborious way
  – Can provide 100% performance extraction
The ways...
Method 0 – Where to start

• There are two possible starting points:
  – Mathematical description of the algorithm
  – Existing C code written for serial execution
• Designing algorithms in Vector form from scratch is the best approach…
• … but the practice shows that much more often it is not the case
  – Most of the time starting point is C code written for “serial” execution
• C code is an excellent starting point
  – It guarantees etalon execution model
• … but you should remember that vectorization is not trivial
  – Vectorized code often needs completely different approach
• Finally set your goals clear
  – Do you want the fastest code possible?
  – Do you want the most compact code possible?
  – Combination of both?
Method 1 – Beware of your Bottlenecks

- One of the first things to realize is whether the algorithm is Compute or I/O bound
  - If the code entirely bounded by memory performance it would not help to reduce the latency of computations
  - If the code is computation intensive, we probably want to reduce that latency and then revisit memory bandwidth
  - If the code is control intensive, it might not be the best candidate for vectorization unless you can use predication and convert control dependency to data dependency

- If memory is the bottleneck the question is –
  - Is it streaming data case (system bus is the bottleneck)
  - … or is it cache resident scenario (same data processed more than once, and then flushed back to main memory)

- If system bus is the bottleneck…
  - Is it saturated?
    - The theoretical 60x bus throughput is 640 Mbytes/sec @ 100MHz
    - The theoretical MPX bus throughput is 800 Mbytes/sec @ 100MHz
  - If you have already reached your bus capacity, and you are sure you are doing just the necessary work – what else can you do?
Method 1 – Beware of your Bottlenecks

- If we are dealing with cache resident data, you should strongly consider using AltiVec
  - AltiVec has **FOUR TIMES** the bandwidth
    - One vector load brings 128 bit of data
    - One FP load - 64 bit
    - Once scalar load only 32 bit
  - **Beware, that there is no direct “link” between GPRs and Vector Registers (VRs)**
  - **To “copy” a value between the two you need to store-load it**
    - This means that you do need to keep majority of computations in one place
  - **One exception from this rule is memcpy/memmove/memset kind of operation**
    - If you copy large amounts of data between two locations, it might be useful to use AltiVec just as a transport manager
- If computations are the critical path…
  - Is there true data dependency between values being computed?
  - If there is not, you could be almost certain that AltiVec will deliver performance improvement
Method 2 – Look at the loops

• The easiest way to reduce computation latency is to restructure loops
  – This is the main optimization performed by autovectorizers
• Loop Unrolling
  – Do multiple iterations in one pass

```
int res[n],a[n],b[n];
for (i=0; i<n; i++){
    res[i] = a[i] + b[i];
}
```

```
int res[n],a[n],b[n];
for (i=0; i<n/4; i+=4){
    res[i]     = a[i]     + b[i];
    res[i+1] = a[i+1] + b[i+1];
    res[i+2] = a[i+2] + b[i+2];
    res[i+3] = a[i+3] + b[i+3];
}
```

```
vector int vres[q_n],va[g_n],vb[q_n];
for (i=0; i<n/4; i+=4){
    vres[i]     = vec_add(va[i],vb[i]);
}
```

• Change the amount of computations in the loop
  – traditionally you might want to remove loop invariant computations from the loops and do several smaller loops as opposed to single large one…
  – but paradoxically it might sometimes HELP to move MORE computations into loops
Method 3 – Look at data dependency

- True Data dependency is often preventing vectorization
  - as well as some classical code optimizations
- But in some cases it could be eliminated
- Let us consider Dot Product of two Matrixes
  - X,Y vectors size N

\[
\text{Dot} \_\text{Product}(x[n], y[n]) = \sum_{i=1}^{n} x[i] \times y[i]
\]

```c
int DotProduct( int *X, int *Y, int length ){
    int temp = 0;

    // N Iterations
    for( int i = 0; i < length; i++ ) {
        temp = X[i]*Y[i] + temp;
    }

    return temp;
}
```
Method 3 – Look at data dependency

• Same function could be written in vector form
  – Note that v1 and v2 are size of N/4
  – so is the “length” == four times fewer iterations

```c
int VectorDotProduct( vector int *v1, vector int *v2, int length ){
    vector int temp = (vector int) vec_splat_u32(0);
    int result;
    // Loop over the length of the vectors multiplying like terms and summing
    // Number of iterations is N/4
    for( int i = 0; i < length; i++ )
        temp = vec_madd( v1[i], v2[i], temp);
    // Still have four ints splat across a vector
    // Add across the vector
    temp = vec_add( temp, vec_sld( temp, temp, 4 )); // Vector Shift Left Double
    temp = vec_add( temp, vec_sld( temp, temp, 8 ));
    vec_ste( temp, 0, &result );

    return result;
}
```
Method 3 – Look at data dependency

- But is this the best possible way of doing it?
  - `vec_madd` takes 4 cycles to complete…

```c
int VectorDotProduct( vector int *v1, vector int *v2, int length ){
    vector int temp = (vector int) vec_splat_u32(0);
    int result;
    // Loop over the length of the vectors multiplying like terms and summing
    // Number of iterations is N/4
    for( int i = 0; i < length; i++ )
        temp = vec_madd( v1[i], v2[i], temp ); // true data dependency
             // only 1 madd every 4 cycles
        temp = vec_add( temp, vec_sld( temp, temp, 4 )); // Vector Shift Left Double
        temp = vec_add( temp, vec_sld( temp, temp, 8 ));
    vec_ste( temp, 0, &result );

    return result;
}
```
Method 3 – Look at data dependency

• Now eliminate the data dependency…

```c
int FastVectorDotProduct( vector float *v1, vector float *v2, int length ){
    vector float temp = (vector float) vec_splat_s8(0);
    vector float temp2 = temp;    vector float temp3 = temp;
    vector float temp4 = temp;    vector float result;
    for( int i = 0; i < length; i += 4){ //Loop over the length of the vectors,
        temp   = vec_madd( v1[i], v2[i], temp);    //this time doing 4 vectors in parallel
        temp2 = vec_madd( v1[i+1], v2[i+1], temp2);
        temp3 = vec_madd( v1[i+2], v2[i+2], temp3);
        temp4 = vec_madd( v1[i+3], v2[i+3], temp4);
    } //Sum our temp vectors
    temp    = vec_add( temp, temp2 );
    temp3   = vec_add( temp3, temp4 );
    temp    = vec_add( temp, temp3 );
    //Add across the vector
    temp    = vec_add( temp, vec_sld( temp, temp, 4 ));
    temp    = vec_add( temp, vec_sld( temp, temp, 8 ));
    //Copy the result to the stack so we can return it via the IPU
    vec_ste( temp, 0, &result );
    return result;
}
```
Method 4 – Look at your Data Layout

- Often algorithm calls for loading of blocks of data in certain order
  - Images (pixels orders) are good example
  - Let us look at RGB to YCbCr conversion

<table>
<thead>
<tr>
<th>Layout 1</th>
<th>Layout 2</th>
<th>Layout 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 2 2 2</td>
<td>10 10 10 11 11</td>
<td>8 1 8 1 8 9 10 16</td>
</tr>
<tr>
<td>11 12 12 13 13</td>
<td>20 21 21 21 22</td>
<td>16</td>
</tr>
<tr>
<td>22 22 23 23 24</td>
<td>31 31 32 32 32</td>
<td>24</td>
</tr>
</tbody>
</table>

…”
Method 4 – Look at your Data Layout

- One vector load from Layout 1 yields this:

```
1 1 1 2 2 2
5 5 5 6
```

- One solution (and maybe the only) is to get 3 vectors worth, and then use `vec_perm` instruction

```
1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6
6 6 7 7 7 8 8 8 9 9 9 10 10 10 11
11 12 12 12 13 13 13 14 14 15 15 15 16 16 16
0 3 6 9 12 15 18 21 1 4 7 10 13 16 19 22
2 5 8 11 14 17 20 24 x x x x x x x x x
```

```
1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 x x x x x x x x x
```
Method 4 – Look at your Data Layout

• One vector load from Layout 2 yields 16 bytes of one “color”

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

• Which means that only three vector loads will yield 16 full pixels
  – In three vector registers

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

• This is the fastest way to get data “in” BUT only if processing is done on “chars” and no extra precision is needed
Method 4 – Look at your Data Layout

- The actual formula for RGB to YCbCr conversion is:

\[
Y = \frac{77}{256} \left( 219 \cdot \frac{R_y}{256} + 16 \right) + \frac{150}{256} \left( 219 \cdot \frac{G_y}{256} + 16 \right) + \frac{29}{256} \left( 219 \cdot \frac{B_y}{256} + 16 \right)
\]

\[
Cb = -\frac{44}{256} \left( 219 \cdot \frac{R_y}{256} + 16 \right) - \frac{87}{256} \left( 219 \cdot \frac{G_y}{256} + 16 \right) + \frac{131}{256} \left( 219 \cdot \frac{B_y}{256} + 16 \right) + 128
\]

\[
Cr = \frac{131}{256} \left( 219 \cdot \frac{R_y}{256} + 16 \right) - \frac{110}{256} \left( 219 \cdot \frac{G_y}{256} + 16 \right) - \frac{21}{256} \left( 219 \cdot \frac{B_y}{256} + 16 \right) + 128
\]

- Which is equivalent to…

\[
Y = \frac{8432}{32768} (R_y) + \frac{16425}{32768} (G_y) + \frac{3176}{32768} (B_y) + 16
\]

\[
Cb = -\frac{4818}{32768} (R_y) - \frac{9527}{32768} (G_y) + \frac{14345}{32768} (B_y) + 128
\]

\[
Cr = \frac{14345}{32768} (R_y) - \frac{12045}{32768} (G_y) - \frac{2300}{32768} (B_y) + 128
\]

- Which needs 16 bits precision for accurate computation…
Method 4 – Look at your Data Layout

- This could be done by “unpacking” bytest to shorts:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

- vec_unpack_2sh and vec_unpack_2sl

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

- On which the computations are performed… and packed back to bytes by vec_packsu()

- This means that by “reverse engineering” the necessary order of bytes back to memory we will get Layout3, so there is no need for permute instructions after vector loads

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>8</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>10</td>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>32</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>32</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>32</td>
<td>25</td>
<td>26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
</tr>
<tr>
<td>32</td>
</tr>
</tbody>
</table>
Method 4 – Look at your Data Layout

• Do the global Data Layout analysis
  – Caches are only working well on data streams which exhibit spatial and time locality
  – Remember that two vector loads == one cache line
    ▪ If loading from multiple cache lines, do one load from each line, then go back and load the second half (also works for scalar access)
    ▪ Do not group too many loads and stores
      – 8-16 vector stores in a row can overflow CSQ (completed store queue) and cause processor to stall
      – but remember of store merging – put two stores to the same cache line together

• See if “in place” computations are possible
  – sometimes reduces memory traffic in half
Method 5 – Look at the Data Types

• Similarly to the optimization method 4 we can see that Data Type analysis can affect algorithm mapping

• In addition to “normal” or forward data type analysis…
  – If you multiply two bytes, you better use short as result

• … there could be a “reverse” data analysis
  – If the actual precision of the result being used is LESS then the precision provided by extended data types, maybe simple rounding will suffice

• A good example of this rule is use of double precision floating point in many embedded algorithms
  – Often original algorithms are developed for “generic” conditions, which might not meet exact use of the algorithm in this specific instance
  – In this case it is a variation of the Method 0 – know (profile) your application and possible data set
Method 5 – Look at the Data Types

• One case of data type consideration (and partially data layout) is aligning allocated data to quad word boundaries
  – Different compilers have different means of achieving it, but all of them DO
    ▪ Here is GCC example…

```c
typedef union{
    vector unsigned int vec;
    int elements[4];
}LongVector __attribute__ ((aligned (16))) ;

unsigned char 8bitBuf[] __attribute__ ((aligned (16))) ={
    #include "attribute_table.txt"
};
```

• In this example every variable of data type LongVector will be aligned on quad-word boundaries and 8bitBuf is already aligned
• Data Alignment is absolutely critical for mapping algorithms on AltiVec
Method 5 – Look at the Data Types

• Why?

For a series of array elements: A0, A1, A2, A3

\[
\begin{align*}
\text{lvx} & \quad v00, \& A0 \\
\text{lvx} & \quad v01, \& A1 \\
\text{lvsl} & \quad v02, \& A0 \\
\text{vperm} & \quad v10, v00, v01, v02 \\
\text{lvx} & \quad v10, v00, v01, v02 \\
\text{lvx} & \quad v00, \& A2 \\
\text{vperm} & \quad v11, v01, v00, v02 \\
\text{lvx} & \quad v01, \& A3 \\
\text{vperm} & \quad v12, v00, v01, v02 \\
\text{lvx} & \quad v00, \& (A3+16) \\
\text{vperm} & \quad v13, v01, v00, v02
\end{align*}
\]
Method 5 – Look at the Data Types

• Loading Unaligned Data requires getting twice (or more) the data you really need

```c
vector unsigned char vectorLoadUnaligned( vector unsigned char *v ){
  vector unsigned char permuteVector = vec_lvsI( 0, (int*) v );
  vector unsigned char low = vec_ld( 0, v );
  vector unsigned char high = vec_ld( 16, v );
  return vec_perm( low, high, permuteVector );
}
```
Method 5 – Look at the Data Types

- **Store Unaligned Data is even ‘better’ ...**
  - You need to **LOAD** in order to be able to Store!

```c
void vectorStoreUnaligned( vector unsigned char v, vector unsigned char *where){
    vector unsigned char permuteVector = vec_lvsr( 0, (int*) where );
    vector unsigned char low,high,tmp,mask;
    vector signed char ones                  = vec_splat_s8( -1 );
    vector signed char zeroes              = vec_splat_s8( 0 );

    vector unsigned char low   = vec_ld ( 0, where ); //Load the surrounding area
    vector unsigned char high  = vec_ld ( 16, where );
    //Make a mask for which parts of the vectors to swap out
    mask = vec_perm( zeroes, ones, permuteVector );
    tmp  = vec_perm( tmp, tmp, permuteVector ); //Right rotate our input data
    low  = vec_sel( tmp, low, mask ); // Insert masked data to aligned vector
    high = vec_sel( high, v, mask );

    vec_st ( low, 0, where );    //Store aligned results
    vec_st ( high, 16, where );
}
```
Method 6 – Eliminate Branching

• You cannot proceed at full speed unless you know exactly where you are going…
• When processor encounters a branch instruction, and condition data is not available processor guesses…
  – And if it guessed wrong, it will back track to the decision point and start over
• There are some general guidelines on how processor will guess…
  – Static branch prediction: Forward branch – not taken, backward branch is taken
    ▪ Which means in if-then-else place LIKELY section in “then”
  – Dynamic branch prediction – after one or two invocations of the same branch instructions enough HISTORY is accumulated to make good prediction next time around…
    ▪ But branch predictor is vulnerable to aliasing…
• Try to avoid branches even if it means more computations – it is likely to be faster!
Method 6 – Eliminate Branching

• A simple example of finding maximum of two numbers
  – Assuming it is used to compare two arrays

```c
int Max( int a, int b ) {
    int result;
    if( a < b )    result = b;
    else            result = a;
    return result;
}
```

```c
//Return the maximum of two vector integers:  result = (a & ~mask) | (b & mask);
vector signed int Max(vector signed int a, vector signed int b ){
    vector bool    int mask = vec_cmplt ( a, b );  //If ( a < b)...
    vector signed int result = vec_sel( a, b, mask );  //Select a or b
    return result;
}
```
Method 6 – Eliminate Branching

- Some compare instructions write their results to integer unit registers
  - `vec_any_xx()` functions that return 1 if any element satisfies the test
  - `vec_all_xx()` functions that return 1 if all of the elements in the vector satisfy the test

//Return true if the second and third floats in v are greater than 0.0
Boolean AreSecondAndThirdElementsPositive( vector float v ){
    vector unsigned int compare = (vector unsigned int)( QNaN, 0, 0, QNaN);
    return vec_all_nle( v, (vector float) compare );
}
Method 7 – Look at Memory data rate

- Memory bandwidth is the natural (and actually desirable) limit of productivity for I/O based applications
  - If it is a streaming application we are mainly talking about system bus throughput
    - Maximum Actual 60x bus throughput is about 640 Mbytes/sec @ 100MHz
    - Maximum Actual MPX bus throughput is about 800 Mbytes/sec @ 100MHz
  - If we are reusing data already fetched, we are probably dealing with Cache hierarchy
    - Approximate L1 cache scalar throughput is around 3.7 Gbytes/sec @ 1GHz
    - AltiVec provides 4x the bandwidth accessing it at 15 GBytes/sec @ 1GHz
- For the case when system bus is the bottleneck, we need to guarantee that it is used all the time
  - In AltiVec it is achieved with Data Stream Touch instruction (dstx)
- For the cases when we mainly depend on Cache performance, we need to improve locality and eliminate unnecessary requests
  - This is done in part with careful layout
  - and dcba/dcbt instructions
Method 7 – Look at Memory data rate

- AltiVec allows up to four prefetch streams, independent and asynchronous
  - addressed by a two bit ID tag
  - \( \text{Vec}_{\text{dst}}(a,b,c) \); where \( a \) is initial address, \( b \) is control constant, \( c \) is ID tag

![Data Stream Prefetch Diagram]

- Block Size = 0-32 Vectors
- 0-256 Blocks
- Stride = ±32KBytes
Method 7 – Look at Memory data rate

- One to four cache streams could be set using
  - `vec_dst` (Data Stream Touch for load) and...
  - `vec_dstst` (Vector Data Stream Touch for Store == load+store)
- Transient variants `vec_dstt()` and `vec_dststt()`
  - Mark their blocks to be flushed straight to RAM instead of L2 and L3 caches
- Use `vec_dst()` and `vec_dstt()` just to read a block of data
- Use `vec_dstst()` and `vec_dststt()` to read and modify a block of data
- Do not use prefetch for write only
  - All writes go into a Store Miss Merge Queue
  - When making two adjacent vector stores to the same cache line, they could be merged into one cache line store, so no memory read is needed
    - This is also more efficient then using dcbz (less an instruction)
Method 7 – Look at Memory data rate

- Prefetch thread is a low priority process
  - There are multiple events that can delay or stop prefetch
- Prefetch streams will also stop **silently** if they step on memory that is either unmapped or would cause a protection violation
- Prefetch engine could be shared by user and OS
Method 7 – Look at Memory data rate

• To improve Cache dependent case, we might want to “help” it
  – AltiVec is capable of ‘hinting’ Cache on future usage of data
• vec_ldl() and vec_stl() (Vector Load/Store Indexed LRU) mark the new cache blocks as those least recently used
  – They will be the first to be flushed when more space in the cache is needed
  – They also mark their cache blocks as transient, which means that they will be flushed directly to memory rather than take up space in the L2
• Next step is to use cache management instructions – dcba/dcbl/dcbz
  – dcba (Data Cache Block Allocate) – allocates cache block without fetching it from memory
    ▪ this is telling Cache “I will soon store into this cache block”
  – dcbl (Data Cache Block Touch) – fetches the cache block from memory
    ▪ this means “I will soon load from this location”
  – dcbl (Data Cache Block Clear to Zero) – same as dcba, but zeroes out allocated block
• There are many more dcbx instructions for a variety of scenarios
Method 8 – Get rid of memory accesses…

• If you do not like it… do not do it!
• All the color conversions do the simple calculation for each color of output space using values of colors from input space
  – cmyk_ycck_convert() from GNU GhostScript
  – The equation looks like the following
    ▪ OutColor1 = C1 * InColor1 + C2 * InColor2 + C3 * InColor3
      – where C1, C2 and C3 are the constants
• To speed up the calculation original GNU code uses the pre calculated table to exchange C1 * InColor1 with Table[InColor1]
  – The same trick is used to limit range of the output value (to one byte length)
• AltiVec version calculate the output value “as is” using original equation and “mradd” instructions.
  – It is match faster than memory byte access operations
  – And we do such calculation for eight pixels simultaneously
• The result is **5x Faster**…
Method 8 – Get rid of memory accesses…

while (--num_rows >= 0) {
  ...
  for (col = 0; col < num_cols; col++) {
    r = MAXJSAMPLE - GETJSAMPLE(inptr[0]);
    g = MAXJSAMPLE - GETJSAMPLE(inptr[1]);
    b = MAXJSAMPLE - GETJSAMPLE(inptr[2]);
    /* K passes through as-is */
    outptr3[col] = inptr[3];
    inptr += 4;
    /* Y */
    outptr0[col] = ((ctab[r+R_Y_OFF] + ctab[g+G_Y_OFF] + ctab[b+B_Y_OFF]) >> SCALEBITS);
    /* Cb */
    outptr1[col] = ((ctab[r+R_CB_OFF] + ctab[g+G_CB_OFF] + ctab[b+B_CB_OFF]) >> SCALEBITS);
    /* Cr */
    outptr2[col] = ((ctab[r+R_CR_OFF] + ctab[g+G_CR_OFF] + ctab[b+B_CR_OFF]) >> SCALEBITS);
  }
}
Method 8 – Get rid of memory accesses…

```c
while (--num_rows >= 0) {
    source0 = vec_ld(0, inptr);
    for (col = 0; col < num_cols; source0 = source4, col+=16) {
        *
        /* Perform 16-bit arithmetic conversion of R,G,B to Y,Cb,Cr
         * Y = 0.29900 * R + 0.58700 * G + 0.11400 * B
         * Cb = -0.16874 * R - 0.33126 * G + 0.50000 * B + CENTERJSAMPLE
         * Cr = 0.50000 * R - 0.41869 * G - 0.08131 * B + CENTERJSAMPLE */
         y0   = vec_mradds(ss_ry,  r0, vec_mradds(ss_gy,  g0, vec_mradds(ss_by,  b0, ss_zero)));
         y1   = vec_mradds(ss_ry,  r1, vec_mradds(ss_gy,  g1, vec_mradds(ss_by,  b1, ss_zero)));
         cb0  = vec_mradds(ss_rcb, r0, vec_mradds(ss_gcb, g0, vec_mradds(ss_bcb, b0, ss_center)));
         cb1  = vec_mradds(ss_rcb, r1, vec_mradds(ss_gcb, g1, vec_mradds(ss_bcb, b1, ss_center)));
         cr0  = vec_mradds(ss_rcr, r0, vec_mradds(ss_gcr, g0, vec_mradds(ss_bcr, b0, ss_center)));
         cr1  = vec_mradds(ss_rcr, r1, vec_mradds(ss_gcr, g1, vec_mradds(ss_bcr, b1, ss_center)));
         */
         y    = vec_packsu(y0,  y1);
         cb   = vec_packsu(cb0, cb1);
         cr   = vec_packsu(cr0, cr1);
         ...
         *outptr0++ = y;
         *outptr1++ = cb;
         *outptr2++ = cr;
         *outptr3++ = k;
    }
}
```
Method 9 – Get rid of computations…

- Well, the same idea, do not like it, don’t do it…
  - You have some control over where the bottleneck is, so move it around a bit
- **Data slicing** is a very effective technique to be used in AltiVec
- It is best explained with an example
- Let’s consider Byte-wise Bit reversal algorithm
  - for each byte return bit reversal version of the input:

```c
unsigned char reverse (unsigned char in){
    unsigned char out = ((in & 0x01)<<7) |
                        ((in & 0x02) <<5) |
                        ((in & 0x04) <<3) |
                        ((in & 0x08) <<1) |
                        ((in & 0x10) >>1) |
                        ((in & 0x20) >>3) |
                        ((in & 0x40) >>5) |
                        ((in & 0x80) >>7);
    return out;
}
```

- This straightforward method yields 0.11 Bytes/Cycle
Method 9 – Get rid of computations…

- Alternative implementation could be *Big Lookup Table*:
  - 256 entry byte table holding the “reversed” values
  - So, the computation for each byte is converted into a single “load”
    - `reversed[j] = big_lookup[j];`

  ```c
  unsigned char big_lookup[256] = {
    0x00,0x80,0x40,0xc0,0x20,0xa0,0x60,0xe0,0x10,0x90,0x50,0xd0,0x30,0xb0,0x70,0xf0,  
    0x08,0x88,0x48,0xc8,0x28,0xa8,0x68,0xe8,0x18,0x98,0x58,0xd8,0x38,0xb8,0x78,0xf8,  
    0x04,0x84,0x44,0xc4,0x24,0xa4,0x64,0xe4,0x14,0x94,0x54,0xd4,0x34,0xb4,0x74,0xf4,  
    0x0c,0x8c,0x4c,0xc0,0x2c,0xac,0x6c,0xec,0x1c,0x9c,0x5c,0xdc,0x3c,0xbc,0x7c,0xfc,  
    0x02,0x82,0x42,0xc2,0x22,0xa2,0x62,0xe2,0x12,0x92,0x52,0xd2,0x32,0xb2,0x72,0xf2,  
    0x0a,0x8a,0x4a,0xca,0x2a,0xaa,0x6a,0xea,0x1a,0x9a,0x5a,0xda,0x3a,0xba,0x7a,0xfa,  
    0x06,0x86,0x46,0xc6,0x26,0xa6,0x66,0xe6,0x16,0x96,0x56,0xd6,0x36,0xb6,0x76,0xf6,  
    0x0e,0x8e,0x4e,0xce,0x2e,0xae,0x6e,0xee,0xe,0xe,0xe,0xe,0xe,0xe,0xe,0xe,0xe,0xe,  
    0x01,0x81,0x41,0xc1,0x21,0xa1,0x61,0xe1,0x11,0x91,0x51,0xd1,0x31,0xb1,0x71,0xf1,  
    0x09,0x89,0x49,0xc9,0x29,0xa9,0x69,0xe9,0x19,0x99,0x59,0xd9,0x39,0xb9,0x79,0xf9,  
    0x05,0x85,0x45,0xc5,0x25,0xa5,0x65,0xe5,0x15,0x95,0x55,0xd5,0x35,0xb5,0x75,0xf5,  
    0x0d,0x8d,0x4d,0xcd,0x2d,0xad,0x6d,0xe1,0xd1,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,  
    0x03,0x83,0x43,0xc3,0x23,0xa3,0x63,0xe3,0x13,0x93,0x53,0xd3,0x33,0xb3,0x73,0xf3,  
    0x0b,0x8b,0x4b,0xcb,0x2b,0xab,0x6b,0xeb,0x1b,0x9b,0x5b,0xdb,0xb,0xbb,0x7b,0xfb,  
    0x07,0x87,0x47,0xc7,0x27,0xa7,0x67,0xe7,0x17,0x97,0x57,0xd7,0x37,0xb7,0x77,0xff,  
    0x0f,0x8f,0x4f,0xcf,0x2f,0xaf,0x6f,0xef,0x1f,0x9f,0x5f,0xdf,0x3f,0xbf,0x7f,0xff
  };
```

- This method yields **0.33 Bytes/Cycle** or 3x faster then original
Method 9 – Get rid of computations…

• Another method is *Small Lookup Table*
  – based on splitting each byte into two nibbles
  – looking up values for both of them independently, and merging result later

```c
unsigned char small_lookup_l[16] __attribute__((aligned (16))) = {
    0x00,0x08,0x04,0x0c,0x02,0x0a,0x06,0x0e,0x01,0x09,0x05,0x0d,0x03,0x0b,0x07,0x0f
};

unsigned char small_lookup_h[16] __attribute__((aligned (16))) = {
    0x00,0x80,0x40,0xc0,0x20,0xa0,0x60,0xe0,0x10,0x90,0x50,0xd0,0x30,0xb0,0x70,0xf0
};

reversed[j] = small_lookup_l[(j&0xf0)>>4] | small_lookup_h[(j&0x0f)];
```

• This method uses less memory, but runs a bit slower: **0.25 Bytes/Cycle**
Method 9 – Get rid of computations…

• The true advantages comes from observation that small tables will fit into two ALtiVec registers…
• … and ALL lookups are completely independent, so 16 of them could be performed in parallel !!!

```c
void reverse_vector(vector unsigned char *in,vector unsigned char *out, int num_elements){
    int i;
    vector unsigned char st_l, st_h;
    vector unsigned char four = vec_splat_u8(4);
    vector unsigned char v_in,vl,vh, v_out;

    st_l = vec_ld (0,(vector unsigned char *) small_lookup_l);
    st_h = vec_ld (0,(vector unsigned char *) small_lookup_h);

    for(i=0; i<num_elements; i+=16){
        v_in = vec_ld (i,in);
        vh = vec_sr(v_in,four);
        vl = vec_sl(v_in,four);
        vh = vec_perm(st_l,st_l,vh);
        vl = vec_perm(st_h,st_h,vl);
        v_out = vec_or(vh,vl);
        vec_st(v_out,i,out);
    }
}
```

• This method for the same conditions gets 1.9 Bytes/Cycle
• It is 17x faster then original Scalar
• …and 5.7x faster then BigLookupTable
Method 10 – Constant Generations

• Often a “standard” C declaration of an initialized variable is interpreted by compiler as declare + store…

\[
\begin{align*}
\text{vector signed int} & \quad \text{zero_vec\_32} = \{ 0,0,0,0 \}; \\
\text{vector float} & \quad \text{zero_vec\_fp} = \{ 0.0,0.0,0.0,0.0 \};
\end{align*}
\]

• In this case it is much more practical to generate these constants

\[
\begin{align*}
\text{vector signed int} & \quad \text{zero_vec\_32} = \text{vec\_splat\_s32}(0); \\
\text{vector float} & \quad \text{zero_vec\_fp} = \text{vec\_ctf( vec\_splat\_u32(0), 0 )}; \\
& \quad \text{// Vector Convert from Fixed-Point Word}
\end{align*}
\]

• These are trivial cases, but what about “fancy” constants…

\[
\begin{align*}
\text{vector float vec\_neg\_zero( void )}\{ & \quad \text{//Generate a vector full of –0.0.} \\
& \quad \text{vector unsigned int result} = \text{vec\_splat\_u32}(-1); \quad \text{// Vector Splat} \\
& \quad \text{return (vector float) vec\_sl( result, result );} \quad \text{// Vector Shift Left}
\end{align*}
\]
Method 10 – Constant Generations

- A very common reason for constant generation is their use for vector permute instruction

```c
vector unsigned char
vectorLoadUnaligned( vector unsigned char *v ){
    vector unsigned char permuteVector = vec_lvsl( 0, v );
    vector unsigned char low = vec_ld( 0, v );
    vector unsigned char high = vec_ld( 16, v );
    return vec_perm( low, high, permuteVector );
}
```

```c
vector unsigned char
vec_lvsl( int index, vector unsigned char *v ){
    int d = vec_lvsl(a, b);
    int EA = a + b;
    if( sh == 0x0 ) then d = 0x00102030405060708090A0B0C0D0E0F;
    if( sh == 0x1 ) then d = 0x012030405060708090A0B0C0D0E0F;
    if( sh == 0x2 ) then d = 0x0230405060708090A0B0C0D0E0F;
    if( sh == 0x3 ) then d = 0x03405060708090A0B0C0D0E0F;
    if( sh == 0x4 ) then d = 0x045060708090A0B0C0D0E0F;
    if( sh == 0x5 ) then d = 0x0560708090A0B0C0D0E0F;
    if( sh == 0x6 ) then d = 0x06708090A0B0C0D0E0F;
    if( sh == 0x7 ) then d = 0x078090A0B0C0D0E0F;
    if( sh == 0x8 ) then d = 0x0890A0B0C0D0E0F;
    if( sh == 0x9 ) then d = 0x09A0B0C0D0E0F;
    if( sh == 0xA ) then d = 0x0AH0B0C0D0E0F;
    if( sh == 0xB ) then d = 0x0BH0B0C0D0E0F;
    if( sh == 0xC ) then d = 0x0CH0B0C0D0E0F;
    if( sh == 0xD ) then d = 0x0DH0B0C0D0E0F;
    if( sh == 0xE ) then d = 0x0EH0B0C0D0E0F;
    if( sh == 0xF ) then d = 0x0FH0B0C0D0E0F10;
}
```

```c
vector unsigned char
vec_lvsr( int index, vector unsigned char *v ){
    int d = vec_lvsr(a, b);
    int EA = a + b;
    if( sh == 0x0 ) then d = 0x101112131415161718191A1B1C1D1E1F;
    if( sh == 0x1 ) then d = 0x0F101112131415161718191A1B1C1D1E1F;
    if( sh == 0x2 ) then d = 0x0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0x3 ) then d = 0x0D0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0x4 ) then d = 0x0C0D0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0x5 ) then d = 0x0B0C0D0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0x6 ) then d = 0x0A0B0C0D0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0x7 ) then d = 0x090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0x8 ) then d = 0x08090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0x9 ) then d = 0x0708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0xA ) then d = 0x060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0xB ) then d = 0x05060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0xC ) then d = 0x0405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0xD ) then d = 0x030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0xE ) then d = 0x02030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F10;
    if( sh == 0xF ) then d = 0x0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F10;
}
```
Method 10 – Constant Generations

- But there are number of cases that are much more complex...

```
vector unsigned char vec_perm1 =
    (vector unsigned char){0,1,2,3,16,17,18,19,4,5,6,7,20,21,22,23};
```

```
vector unsigned char a = vec_lvsl(0, 0);
vect unsigned char b = vec_lvsr(0, 0);
vect unsigned char vec_perm1 =
    (vector unsigned char)vec_mergeh((vector unsigned int)a, (vector unsigned int)b);
```
Method 10 – Constant Generations

00: (1) splat 00
01: (1) splat 01
...
10: (2) splat 08, add self
11: (3) splat 02, splat 0F, add together
12: (2) splat 09, add self
...
3D: (3) splat F4, splat 06, rol by
3E: (2) splat FA, srl self
3F: (3) splat F3, splat 04, rol by
...
B1: (4) splat F6, rol self; splat F4; add together
B2: (4) splat F6, rol self; splat F5; add together
B3: (3) splat 0B, rol self, subtract
...
FE: (1) splat FE
FF: (1) splat FF

32 sequences consist of 1 instruction
44 sequences consist of 2 instructions
152 sequences consist of 3 instructions
28 sequences consist of 4 instructions

Many thanks to Holger Bettag
Put it all together

• **Step back and take 10,000 foot view**
  • There is a logical sequence to be observed in implementation of these methods…
    – One can look at the optimization process as on moving the bottleneck around the processor –
      ▪ if computation takes longer then anything else – speed them up
      ▪ if system bus is underutilized – use prefetching
      ▪ if bus is 100% full, computations are at the minimum… reduce the code and data size?

• **But the truly superior goal is to reach computational entropy** –
  – get rid of all the unnecessary computations through *algorithm* modifications
  – and balance added memory bandwidth with real data I/O
  – use predictability of the data streams to the full extent

• **Concentrate your effort, in large applications work with 10% of the code which accounts for 90% of execution time**
General Coding Strategy

• Use Vector algorithms
  – Aim for high throughput
• Align your Data
  – 16 bytes
  – Never hurts scalar code
  – Keep all data in close proximity
    ▪ Helps to improve memory performance
  – Try not to mix different data types in the same vector
• Do more work
  – On a cold cache assume having 40 cycles for each 32byte chunk of data
  – You are likely to achieve TOP performance when processing time exactly equal to the fetch time
  – Use prefetch and ‘hinting’ instructions
More ways…
AltiVec Library Offering

- **Telecomm**
  - FFT/IFFT, FIR, Autocorrelation, Convolution Encoder/Viterbi Decoder (GSM, 3G), Error Correction Codes (CRC 8, 12, 16, 24)
  - Voice Over IP (G723, G729) elements

- **MultiMedia**
  - DCT/IDCT, MPEG2, MPEG4, H.26x, AC3, MP3, JPEG/JPEG 2000, Quantization/Dequantization, SAD
  - Voice Recognition, Pattern Recognition,

- **Printer**
  - GhostScript Library elements, Color Management routines, Color Conversion (RGB to YCbCr), Scaling/Rotation, Filtering routines, FS Dithering

- **Networking**
  - OSPF, QOS, NAT, Route Lookup, IP Reassembly, TCP/IP,
  - Encryption (AES, DES, 3DES, MD5, HSA, RSA)
  - Wireless network (802.11), LZO

- **LibC (means could be “Linked” at compilation)**
  - Link level support for standard C functions (memcpy, strcmp etc.)

- **Mathematical primitives (Extension of LibC+)**
  - Matrix math, LargeNumber Lib,
  - Math.h, Log, Exp, Sin, Cos, Sqrt

- **OS enablement**
  - Linux (TCP/IP),
  - VxWorks elements

---

Legend

- **Green color** - available on the AV web site now (motorola.com/altivec)
- **Orange color** - available upon request
- **Light blue** - available soon
To summarize:

- **AltiVec™ Technology transparently adds SIMD functionality to a high speed RISC engine**
- **AltiVec enables a broad range of embedded and computing applications**
- **C level programming offers certain level of comfort while providing powerful way to extract parallelism from applications**
- **You must think in terms of Vector Processing throughout design cycle of an application**
  - **AltiVec is not a pixie dust to be sprinkled on an existing code**
- **Given that – 2x-4x-11x speedup is possible**
60x vs. MPX

- **Bandwidth**

<table>
<thead>
<tr>
<th>PLATFORMS</th>
<th>DESCRIPTION</th>
<th>LATENCY</th>
<th>THEORETICAL BANDWIDTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eximer/Maximer</td>
<td>FPGA60x SRAM, 100MHz</td>
<td>2</td>
<td>460 MB/sec</td>
</tr>
<tr>
<td>Sandpoint</td>
<td>60x, SDRAM, 1 level deep pipelining, 100MHz</td>
<td>8</td>
<td>640 MB/sec</td>
</tr>
<tr>
<td>MVP</td>
<td>60x pipelined, SDRAM, 100MHz</td>
<td>19</td>
<td>640 MB/sec</td>
</tr>
<tr>
<td>MVP</td>
<td>MPX pipelined, SDRAM, 100MHz</td>
<td>19</td>
<td>800 MB/sec</td>
</tr>
<tr>
<td>Ideal</td>
<td>MPX, SDRAM, data streaming, data intervention, out of order, 100MHz</td>
<td>6-8</td>
<td>800 MB/sec</td>
</tr>
<tr>
<td>Simple Memory Controller</td>
<td>60x non-pipelined, SDRAM, 100MHz</td>
<td>9</td>
<td>320 MB/sec</td>
</tr>
</tbody>
</table>

- **PCI I/O**

<table>
<thead>
<tr>
<th>PLATFORMS</th>
<th>DESCRIPTION</th>
<th>LATENCY</th>
<th>BANDWIDTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandpoint</td>
<td>60x, SDRAM, 1 level deep pipelining, 100MHz</td>
<td>61</td>
<td>290 MB/sec</td>
</tr>
<tr>
<td>MVP</td>
<td>MPX pipelined, SDRAM, 100MHz</td>
<td>267</td>
<td>172 MB/sec</td>
</tr>
</tbody>
</table>