Vectorization for SIMD Architectures

Hwansoo Han (SKKU)
Single Instruction Multiple Data (SIMD)

- **SIMD computation**

```plaintext
for (i=0; i<N; i++)
    a[i] = b[i] + c[i];
```
SIMD Units in Processors

- All levels of processors have one
  - Supercomputer: BlueGene/L
  - PC: MMX/SSE (x86), AltiVec/VMX (PowerPC), VIS (SPARC)
  - Embedded systems: Neon (ARM), VLIW+SIMD DSPs

- Short vector extensions in ILP processors
  - Accelerate loops in multimedia & DSP codes

- Varying support for char to double computations
- Varying support for aligned or misaligned memory accesses
- Varying support for byte permutations
SIMD Extensions for ILP Processors

- **Cost-efficient**
  - <10% in die area and power increase,
    1.5-4x performance increase in multimedia kernels [Pollack1999]
  - Simple control and data path

- **Weaker Memory Unit**
  - Alignment constraints
    - Aligned at a multiple of vector-length
  - Only accesses contiguous chunk

- **Special-purpose instruction set**
  - Certain operations may not supported
SIMD – Short Vector Computing

- Accelerate multimedia and signal processing algorithms
  - Video encode/decode
  - 2D/3D graphics
  - Gaming
  - Audio and speech processing
  - Image processing
  - Telephony
  - Sound synthesis
SIMD Programming

- SIMD programming tools
  - Auto-vectorization (GCC 4.1+, ICC, RealView v3.1+)

- SIMD programming languages/extensions
  - Assembly languages
  - Intrinsics
  - Ct, Array Building Block (C++ extension) – Intel
SIMD on IA32

- **X87** – FP operations

- **MMX (MultiMedia eXtension)**
  - First SIMD support on x86
  - AMD supports 3DNow!
  - Supports 8, 16, 32 bit integers
  - Eight 64-bit MMX registers (mm0~mm7) share x87 FP registers
  - Cannot operate FP and MMX at the same time

- **Pentium4**
  - x87 and SIMD FP on the same logical unit
SSE (Streaming SIMD Extensions) is added to x86 ISA
- Eight 128-bit xmm registers (xmm0 ~ xmm7)
- Extra registers to save on context switch
- Two 64-bit FP numbers
- Four 32-bit FP numbers
- Two 64-bit integers
- Four 32-bit integers
- Eight 16-bit integers
- Sixteen 8-bit bytes or characters
SIMD on IA32 (cont’d)

- **SSE2**
  - More math instructions for 64-bit FP
  - Extend MMX instructions to operate on XMM registers

- **SSE3, SSE4**
  - DSP-oriented math instructions are added
  - More math instructions

- **SSE5**
  - Proposed by AMD, not fully compatible with Intel’s SSE4

- **AVX (Advanced Vector Extensions)**
  - Announced by Intel, followed by AMD
  - 256-bit registers
MMX/SSE Instructions

- Integers, 32-bit FP, 64-bit FP vector operations
  - Basic arithmetic, reciprocal instructions
  - Comparison
  - Conversion (pack/unpack)
  - Logical (AND, AND-NOT, OR, XOR)
  - Shift/shuffle instructions
  - Data movement (SIMD reg-to-reg, 32-/64-bit load/store to mem)
FP arithmetic

- ADDPS - add packed single precision FP
- ADDPD - add packed double precision FP
- ADDSS - add scalar single precision FP
- ADDSD - add scalar double precision FP
- ...

128 bit register

INT arithmetic

- PADDB, PADDW, PADD (8-bit, 16-bit, 32-bit addition)
- PADDSB, PADDSW (add with saturation)
- PMULHW, PMULLW, PMADDWD
- ...

...
MMX/SSE ISA Examples (cont’d)

- Compare instructions
  - CMPccPS xmm1, xmm2  \((cc = EQ, LT, LE, NE, NLT, NLE, …)\)
    - Compare packed 32-bit FP values in xmm1 and xmm2
    - If true, all 1s are stored in xmm1, otherwise all 0s are stored
      \((0xFFFFFFFF \text{ or } 0x00000000 – 32 \text{ bit mask values})\)
  - PCMPccD – INT comparison \((cc = EQ, LT, LE, NE, NLT, NLE, …)\)

- Bit-wise logical operations
  - ANDPS, ORPS
  - Logical bit-wise and / or for packed 32-bit FP values
  - Can be used to combine comparison mask values
    - No SIMD branch instructions
ARM NEON

- Types of vector elements
  - 8/16/32/64-bit signed/unsigned integer
  - Single precision floating point

- NEON features
  - Aligned and unaligned data accesses
  - 128-bit/64-bit dual view NEON register file

- Tools
  - OpenMAX DL library
  - RealView v3.1 or later, GNU compiler v4.1 or later
    - Automatic vectorization
    - C intrinsics
ARM NEON (cont’d)

- NEON support in open source community
  - Android – NEON optimization
    - Skia library (2D graphics), S32A_D565_Opaque (x5 faster)
  - Ubuntu 09.04
    - NEON versions of critical shared libraries
  - Bluez – Bluetooth protocol stack
    - NEON SBC audio encoder
  - Pixman (Cairo 2D graphic library)
    - Compositing/alpha blending
    - X.org, Mozilla Firefox, Fennec and Webkit browsers
    - fbCompositeSolidMask_nx8x0565neon (8x faster)
  - Ffmpeg – libavcodec, media player
  - X264 – Google Summer of Code 2009
    - H.264 encoder for video conferencing
ARM NEON (cont’d)

- Partners supporting ARM NEON technology

<table>
<thead>
<tr>
<th>Company</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>ingenient technologies</td>
<td>H.264, VC1, MPEG-4</td>
</tr>
<tr>
<td>on2 technologies</td>
<td>VP6/7, MPEG-4, VC1, H.264, video stabilization</td>
</tr>
<tr>
<td>Ittiom</td>
<td>MPEG-4, MPEG-2, H.263, H.264, WMV9, VC1</td>
</tr>
<tr>
<td>ARICENT®</td>
<td>MPEG-4, H.263, H.264, WMV9, audio</td>
</tr>
<tr>
<td>Tata Elxsi Limited</td>
<td>H.264, VC1</td>
</tr>
<tr>
<td>Spirit DSP</td>
<td>TEAMSpirit voice and video</td>
</tr>
<tr>
<td>VisualOn</td>
<td>H.264, MPEG-4, H.263, WMV</td>
</tr>
<tr>
<td>Actimagine</td>
<td>MobiClip</td>
</tr>
<tr>
<td>Fraunhofer IIS</td>
<td>Video and audio codecs</td>
</tr>
<tr>
<td>Dolby</td>
<td>Multichannel audio processing</td>
</tr>
<tr>
<td>Algorithm Specialist TMC</td>
<td>MPEG-4</td>
</tr>
<tr>
<td>Espico</td>
<td>Audio and consulting</td>
</tr>
</tbody>
</table>
Cell Broadband Engine

- Sony, Toshiba, IBM – Cell architecture
  - 1 PPU + 8 SPUs
  - 8 SPUs
    - 128-bit SIMD-only unit

- Server, PS3, CellTV
Compilers for SIMD

- **Automatic SIMDization (vectorization)**
  - Many commercial compilers support
  - But in a limited capacity

- **Automatic parallelization**
  - Share the same analysis
  - But new considerations should be taken
Automatic Vectorization Compilers

- Compilers auto-vectorize loops
  - GNU compiler, Intel compiler, RealView compiler

- But with constraints
  - Loops must be countable
  - No loop-carried dependences (parallel loops)

- GNU compiler flags
  - -ftree-vectorize (by default at -O3)
  - For IA32: -msse -msse2 -msse3, for PowerPC: -maltivec
    For ARM: -mfpu=neon -ftree-vectorize
  - Vectorization of FP reduction: -ffast-math or -fassociative-math

- RealView compiler for ARM NEON
  - armcc --cpu=Cortex-A9 -O3 -Otime --vectorize
Compiler Support for SIMD Extensions

- Vectorization
  - Succeeded for vector processors in 70~80’s
  - Large body of research (Allen & Kennedy)
    - Data parallel loops
    - Breaking loop-carried dependence
      - Scalar expansion and replacement
      - Scalar and array renaming
      - Loop distribution
    - Reduction recognition
      - Scalar expansion for scalar reduction
      - Scatter-gather for array reduction
    - Control dependence
      - If-conversion
  - Most of them are also applicable to SIMD compilation
SIMDization for Modern Architectures

- Sources of SIMD parallelism
  - Loop level
  - Basic block level
  - Entire short loop

- Overheads in SIMD computation
  - Alignment
  - Data conversion
  - Non-consecutive data
SIMD Parallelism in Loops

- Loop level parallelism
  - SIMD for a single statement across consecutive iterations
- Handles
  - Misaligned data
  - Patterns such as reduction, linear recursion
  - Employ cost models to amortize overhead in versioning, alignment
Loop Level Parallelism

```c
for (i=1; i<N; i++) {
    a[i] = b[i] + c[i];
    d[i] = a[i-1] * c[i];
}
```

loop carried dependence: a[ ]

```c
for (i=1; i<N; i++)
    a[i] = b[i] + c[i];
for i=1; i<N; i++)
    d[i] = a[i-1] * c[i];
```

vectorization

```c
for (i=1; i<N; i+=4)
    a[i:i+3] = b[i:i+3] + c[i:i+3];
for (i=1; i<N; i+=4)
    d[i:i+3] = a[i-1:i+2] * c[i:i+3];
```

loop fusion

```c
for (i=1; i<N; i+=4) {
    a[i:i+3] = b[i:i+3] + c[i:i+3];
    d[i:i+3] = a[i-1:i+2] * c[i:i+3];
}
```
Loop Level Parallelism (cont’d)

- **Vectorization** [Sreraman 2000]
  - Finds inter-iteration parallelism
    - ‘Loop-based approach’
  - Transforms data-parallel loops into vector instruction sequences
    - Requires loop distribution to prune dependence cycles
    - Focused on memory subscripts that are affine functions of the index variable

```c
for(i=0; i<N; i++) {
    c[i] = a[i] + b[i];
}
```

`c[0:N-1] = a[0:N-1] + b[0:N-1];`

```c
for(i=0; i<N; i++) {
    a[i+1] = b[i] + c;
    d[i] = a[i] + e;
}
```

`a[1:N] = b[0:N-1] + c;
 d[0:N-1] = a[0:N-1] + e;`

```c
for(i=0; i<N; i++) {
    a[i+1] = b[i] + c;
    d[i] = a[i] + e;
}
```

`d[0:N-1] = a[0:N-1] + e;`
SIMD Parallelism in Basic Blocks

- Superword level parallelism (SLP) in basic blocks
  - SIMD across multiple isomorphic operations within a BB
- Handles
  - Unrolled loops (manually or by compilers)
  - Extraction of SIMD parallelism within structures
  - Extraction of SIMD parallelism within a statement
Superword Level Parallelism

for (i=0; i<N; i+=4) {
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
}

for (i=0; i<N; i++) {
    p = &a[i]; q = &b[i];
    p.x = q.x + ...
    p.y = q.y + ...
    p.z = q.z + ...
}

s += a[i]*b[i] + a[i+1]*b[i+1] + a[i+2]*b[i+2] + a[i+3]*b[i+3];

t[i:i+3] = a[i:i+3] * b[i:i+3];
s += t[i]+t[i+1]+t[i+2]+t[i+3];
Superword Level Parallelism (SLP) [Larsen 2000]

- Finds intra-iteration parallelism
  - ‘Basic-block-based approach’
  - Loop-unrolling is used to transform inter- to intra- iteration parallelism
    - Usually subsumes vectorization
- Adjacent memory references are the basis of finding parallelism
  - Data already packed in memory
  - Packs isomorphic statements by following def-use and use-def chains
  - Leaves unpacked statements in scalar forms
    - Partial vectorization without loop distribution

```c
do {
    dst[0] = (src1[0]+src2[0]) >> 1;
    dst[1] = (src1[1]+src2[1]) >> 1;
    dst += 4; src1 += 4; src2 += 4;
} while (dst != end);
```

```c
do {
    dst[0:3] = (src1[0:3]+src2[0:3]) >> v(1);
    dst += 4; src1 += 4; src2 += 4;
} while (dst != end);
```
SIMD Parallelism in Short Loops

- Parallelism in short loops
  - SIMD across entire loop iterations
  - Effectively collapse innermost loop
  - Allow to extract SIMD at the next loop level

```
for (k=0; k<N; k++) {
  ...other code...
  for (i=0; i<8; i++)
    r[k] += i[k+i] * c[k+i];
}
```

```
for (k=0; k<N; k++) {
  ...other code...
  r[k:k+3] = i[k+i:k+3+i]*c[k+i:k+3+i];
  r[k+4:k+7] = i[k+4+i:k+7+i]*c[k+4+i:k+7+i];
}
```
Alignment Constraints

- Alignment for multiple inputs in SIMD units
  - If they are not aligned each other, more computations are needed (e.g., permutation)
  - Recent SIMD architectures support unaligned accesses
Data Conversion

- **Scalar–vector conversion**
  - Results of scalar operation is fed to vector operations
  - Vice versa

\[
\begin{align*}
\text{add} & \quad a[\ ] = k + d \\
\text{vload} & \quad b[\ ] = \ldots \\
\text{vadd} & \quad c[\ ] = a[\ ] + b[\ ]
\end{align*}
\]
Data Conversion (cont’d)

- **Size of data** in SIMD registers
  - A SIMD register can hold 8 short integers (short = 2 bytes)

![Diagram of SIMD registers](image)

- Only 4 int integers will fit in the same space (int = 4 bytes)

- **When we compute short and int**
  - Shorts need to be converted to ints
  - Use 2x integer SIMD operations
1. Prepare for DFG construction
   ▶ If-conversion, Pre-optimizations
2. Construct a DFG
   ▶ Use-def analysis and memory dependence analysis
3. Identify non-vectorizable nodes
   ▶ Either inherently or due to loop-carried dependency
4. Insert data reorganization nodes
   ▶ Partitions vector and scalar nodes so that the reorganization cost is minimized
5. Exploit superword level parallelism on scalar nodes
   ▶ Data transfer patterns
6. Generate vectorized code
   ▶ Software pipelining and unaligned load-compute-store patterns
DFG Construction

- Data Flow Graph
  - Nodes: Live-in/out variables, constants, and operations
  - Edges: register and memory dependences
- Memory dependence analysis
  - Alias analysis
    - Data Structure Analysis (DSA) [Lattner 2003] embedded in LLVM
    - Also takes disambiguation from programmers (ex. #pragma noalias)
  - Subscript analysis
    - ZIV (zero-index variable) and SIV (single-index variable)
    - inner-most loops only
DFG - example
Selective Vectorization

- Vector resources do not overwhelm the scalar resources
  - Scalar: 2 FP ops / Dual FP units
  - Vector: 4 FP ops / SIMD unit
- Full vectorization may underutilize scalar resources
- ILP techniques do not target vector resources
- Need both

Courtesy of IBM Corporation.
Advances on SIMD Compilation

- **Parallelism within a basic block**
  - Superword Level Parallelism (SLP) [Larsen 2000]
  - Code Selection [Leupers 2000]

- **Alignment constraints**
  - Data Reorganization Graph for stride-one accesses [Eichenberger 2004]
  - Advanced fetch unit for unaligned accesses (Intel’s Nehalem 2008)

- **Non-unit stride**
  - Constant strides that are power of 2 [Nuzman 2006]
  - Advanced memory unit for disjoint data (SIMdD) [Naishlos 2003]
  - **No software approach for arbitrary stride**

- **Competition with ILP**
  - SLP partially vectorizes loops without loop distribution
  - Selective vectorization for VLIW [Larsen 2005]

- **Diversity of ISA**
  - Framework for retargetable compilers [Hohenauer 2009]
Programming for SIMD

- **Compiler intrinsics**
  - Extension for types
  - Extension for dedicated SIMD operations
  - Compiler generates SIMD code with a guidance of intrinsics

- **Language extensions**
  - Support general SIMD programming
  - Examples
    - Ct
    - ArBB (array building block)
SIMD Intrinsics

- Less verbose than assembly instructions
- Blends more naturally with the high-level languages
- Almost direct translation of SSE ISA

GNU Compilers
- X86 Built-in Functions
- Header files for intrinsics
  - mmintrin.h, xmmintrin.h, emmintrin.h, tmmintrin.h, nmmintrin.h, smmintrin.h
  - MMX, SSE, SSE2, SSE3, SSSE3, SSE4

MS Visual Studio also has its own version of MMX, SSE intrinsics
**SIMD Intrinsics Example**

- **__m128**
  ```c
  typedef float __m128 __attribute__((__vector_size__(16)));
  __m128 a; // xmm register or float[4] unaligned
  __m128 b __attribute__((aligned(16))) ; // aligned by 16 bytes
  ```

- **__mm_set_ps()**
  ```c
  a = __mm_set_ps(3.14, 0.18, 2.19, 1.23); // movups xmm0, [rax]
  b = __mm_set_ps(3.14, 0.18, 2.19, 1.23); // movaps xmm0, [rax]
  ```

- **__mm_rcp_ps()**
  ```c
  a = __mm_rcp_ps(a); // rcpps xmm0, xmm0
  ```
Ct

- Not an extension to C++
- Platform independent programming language
- Ct specific containers
  - E.g., TVEC
- Exploit both SIMD and HW thread parallelism

```
TVEC<F64> Values, v;
TVEC<I32> RowIdx, ColP;

TVEC<F64> expv = distribute(v, ColP);
TVEC<F64> product = Values * expv;
Product = product.applyNesting(RowIdx, ctIndex);
TVEC<F64> result = proudct.addReduce();
```
Array Building Block (Intel ArBB)

- C++ extension for SIMD/Multicores
  - Dynamic binding of data with ArBB types
  - Map & Reduce
  - Dynamic compilation and cache with the ArBB Virtual Machine

```c++
float a[N], b[N], out[N];
dense<f32> vec_a, vec_b, vec_c;
bind(vec_a, a, N);
bind(vec_b, b, N);
bind(vec_c, c, N);

call (sum_kernel) (vec_a, vec_b, vec_c);

void sum_kernel (const arbb::dense<arbb::f32>& x1,
                 const arbb::dense<arbb::f32>& x2,
                 const arbb::dense<arbb::f32>& y)
{
    y = x1 + x2;
}
```
Summary

- **SIMD**
  - Short vector processing in modern processors
  - Support various types (byte, short, int, float, double)

- **Optimizations**
  - Alignment for vector loads
  - Data size conversion
  - Handling non-consecutive data

- **Programming models**
  - Assembly programming, intrinsics in high-level PL
  - Automatic simdization with compiler
  - New programming languages/extensions with SIMD support