SIMD tutorial

06.12.2008

Håvard Bjerke
Outline Part 1

- SIMD on Intel architecture
- SSE Vector formats
- Data parallelism
- Strategy
- High-precision performance measurement

- Tools & technologies
  - Automatic vectorization
  - ASM
  - Intrinsics
  - C++ classes
  - Valarray
Outline Part 3

- SIMD techniques
  - Reduction
  - Data alignment
  - Control flow

- Prefetch
Outline Part 4

- Valarray
- Ct
- Future SIMD
- Lessons
- Conclusion
Part 1
- SISD: x87
- MMX
  - First SIMD support on x86
  - AMD responded with 3dnow
  - 64 bit MMX registers
  - Supports 8, 16 and 32 bit elements
- As of P4
  - X87 and SIMD FP on same logical unit
As of EM64T and AMD64, SSE and SSE2 always supported
  ▪ no reason to use x87 any longer

As of Prescott (P4+): SSE3

As of Penryn (Core2 Duo+): SSE4

LRB

AVX (2010)
  ▪ 256 bit registers
SSE FP vector formats

- **Scalar (SISD) double,**
  - “x87”
  - 1 scalar double precision calculation per instruction

- **Packed double**
  - 2 scalar double precision calculations per instruction

- **Packed single**
  - 4 scalar single precision calculations per instruction

128 bits
SSE integer formats

- **EPI64**
  - 2 64-bit integer calculations per instruction
  - 128 bits

- **EPI32**
  - 4 32-bit integer calculations per instruction

- **EPI8**
  - 16 8-bit integer calculations per instruction
### Case: HLT Track Fitter

- **Woodcrest @ 2.4 GHz using ICC 9.1**

<table>
<thead>
<tr>
<th></th>
<th>Calculation time per track / us</th>
<th>Incremental speedup</th>
<th>Total speedup from scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>2.6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>double</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
</tr>
<tr>
<td>single</td>
<td>0.7</td>
<td>2.3</td>
<td>3.7</td>
</tr>
</tbody>
</table>
- SPSD
  - SISD
  - Pre-MMX PCs
- MPMD
  - Asynchronous
  - Task-level parallelism
  - Example: web server
  - Constraints: Shared I/O
- **MPSD**
  - Lock synchronization
  - Task-level parallelism
  - Example: Producer-consumer

- **SPMD**
  - Synchronization:
    - Intrinsic (SIMD)
    - Barriers
  - Data parallelism
  - Examples
    - MPI (typically)
    - OpenMP (typically)
    - SIMD
Example: Game of Life

- Cellular automaton, based on Conway's Game of Life
Game of Life data parallelism

- Dependency *between* iterations
- Data parallelism *within* iterations
- When use SIMD?
  - When the computation is the bottleneck
  - Find the bottleneck first!
- Single Operation Multiple Data is a prerequisite for SIMD
• SIMD in an overall optimization strategy
  • Develop with benchmarking in mind
  • Make sure it's correct and verifiable
  • Develop algorithm with complexity in mind
  • Work on hotspots
  • Develop algorithm with parallel model in mind
  • Develop SIMD
  • Develop multi-threading
  • Get more / faster computers
  • …
Track fitter speedup

- Total speedup w/ both optimizations:
  \[ 3.7 / 0.12 = 30 \]
Part 2: Tools & techniques
Tools & paradigms

- ASM
- Intrinsics
- C++ classes (ICC)
- Valarray (ICC >= 11)
- Auto-vectorization (GCC 4.x, ICC)

More control

More convenience
Exercises in Part 2

- High precision performance measurement
  - Pfmon
  - Rdtsc instrumentation

- SSE Tools & paradigms
  - Auto-vectorization
  - ASM
  - Intrinsics
  - C++ classes
  - Valarray
Required tools

- GCC 4.X – the newer the better
  - GCC 4.3 available on tutorial machine
- ICC 11 – available on AFS and tutorial machine
- Pfmom - available on tutorial machine
Performance measurement

- Why?
  - It is difficult to predict the performance impact of design changes
- Knowledge about behaviour can be gained indirectly
  - E.g. intrapolation, extrapolation
- High precision performance measurement
  - Allows us to investigate directly if we are exploiting the hardware correctly
- Uses the Performance Monitoring Unit in the CPU
- Event counters
  - Counts many aspects of the CPU's behaviour
  - Very small impact on performance
- Profiling
- Instrumentation with libpfm
### SIMD performance counters

- **Using PMU hardware counters with pfmon**
  - (billion instructions)

<table>
<thead>
<tr>
<th>instruction type</th>
<th>scalar</th>
<th>double</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>computational scalar double</td>
<td>10.6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>computational packed double</td>
<td>0</td>
<td>5.5</td>
<td>0</td>
</tr>
<tr>
<td>total packed double</td>
<td>0</td>
<td>5.5</td>
<td>0</td>
</tr>
<tr>
<td>computational packed single</td>
<td>0</td>
<td>0</td>
<td>2.8</td>
</tr>
<tr>
<td>total packed single</td>
<td>0</td>
<td>3.7</td>
<td>4.6</td>
</tr>
<tr>
<td>total SIMD</td>
<td>16.9</td>
<td>9.5</td>
<td>4.7</td>
</tr>
<tr>
<td>total</td>
<td>24.7</td>
<td>17.2</td>
<td>10.9</td>
</tr>
</tbody>
</table>
Exercise 0: Pfmon (1)

- Explore pfmon options
  - “pfmon -h”
  - Try it on a simple program
    - “pfmon factor 100”
  - List available event counters
    - “pfmon -l”
  - Show info about a specific event
    - “pfmon -i UNHALTED_CORE_CYCLES”
Exercise 0: Pfmon (2)

- Assess efficiency of “factor n”
  - What is the Cycles Per Instruction ratio?
    - UNHALTED_CORE_CYCLES
    - INSTRUCTIONS RETIRED
  - How many x87 ops retired?
  - How many SIMD instructions retired?
  - How many branch instructions retired?
  - How many of the branch instructions are mispredicted?

- Try profiling: see “--short-smpl-period”
Exercise 1: Rdtsc instrumentation

- Time stamp counter
  - Counts #cycles (ticks) since boot

- Exercise
  - Convert #cycles to time
    - Hint: see /proc/cpuinfo
  - Is the TSC correct
    - On a multi-core machine?
    - With variable CPU frequency?
- GCC and ICC can automatically vectorize loops for you
- Supported in ~ GCC > 4.1
  - The newer the better
- But with constraints
  - Loops must be countable
  - Dependencies within / between loops
Auto-vectorization

- Things that may cause problems
  - Loop external dependencies
  - Uncountable loops
  - Control flow in loop
Exercise 2: Auto-vectorization

- Try auto-vectorization with GCC
  - `G++ -O3 -msse3 -ftree-vectorize -ftree-vectorizer-verbose=2`

- Which loops are auto-vectorized?

- Try with ICC

- Try to modify the control flow and see if it works better
Explicit methods for vectorization:

- Explicit asm
- Intrinsics, provided by ICC and GCC
- Operator overloading, provided by ICC for C++

Operator overloading allows seamless change of data types, even between primitives (e.g. float) and classes

- Example classes provided by `fvec.h` and `dvec.h`
  - P4_F32vec4 – packed single
  - P4_F64vec2 – packed double
Assembly

- GCC “-s” flag gives assembler output
- Useful to study compiler and hardware behaviour
- Infeasible to use on a large scale
  - Can be useful for hotspots
- More control, less convenience
  - But for SIMD, intrinsics usually give enough control
- **Data movement**
  - MOV* - Either operand is either memory or register
  - MOVAPS – move 16 bytes of aligned data
  - MOVUPS – makes no assumption about alignment
Some examples (2)

- **ADDPS**
  - Add packed single precision – 4 32-bit floating point add per instruction

- **ADDPD**
  - Add packed double precision – 2 64-bit floating point add per instruction

- **ADDSS**
  - Add scalar single precision – 1 32-bit floating point add per instruction – not SIMD
Some examples (3)

- **Compare instructions**
  - **CMPEQPS xmm1, xmm2**
    - Compare PS values in xmm1 and xmm2
    - If true, a true-mask (0xFFFFFFFF) is stored in xmm1, otherwise 0
  - The mask can be used later to validate result
    - There is no SIMD branch instruction!
  - **ANDPS, ORPS**
    - Logical bit-wise and
    - Same as &, but for SIMD
    - Can be used to combine masks
Exercise 3: ASM

- Examine assembler output from Exercise 2 with and without auto-vectorization
- See TODO in asm.cpp
- Examine the assembler output from asm.cpp
Intrinsics

- Less verbose than assembly
- Blends more naturally with the rest of the code
- Almost direct translation of the SSE Instruction Set Architecture
Intrinsics example

```c
__mm128 a = _mm_set_ps(3.14159265, .3183098865, ...

MOVUPS xmm0, [rax]

a = mm_rcp_ps(a);

RCPPS xmm0, xmm0
```
Exercise 4: Intrinsics

- `xmmmintrin.h` implements SSE intrinsics
  - Useful reference
- See TODO in `intrinsics.cpp`
- Is RCP correct?
- Try `div_ps`
Intel C++ classes

- dvec.h, fvec.h – ICC specific headers for vector classes
- Higher level of abstraction than __m128x data types
- Classes have operator overloading
  - Allows seamless interchange with native data types
Exercise 5: Intel C++ classes

- Compile classes.cpp with icpc
- See TODO
- Since ICC 11
- Arbitrarily sized vectors
- Also has native-like operators
Exercise 6: Optimized Valarray

- See TODO in valarray.cpp
- What is the performance?
- What is the benefit of valarray over vector classes?
- Try shifting a large vector with more than 4 elements
  - With vector classes
  - With valarray
Part 3
Exercises in Part 3

- **SSE Techniques**
  - Reduction
  - Vector formats
  - Alignment
  - Control flow
  - Prefetch
**Shuffle**

- SHUFxx xmm1, xmm2/mem, IMM8
Exercise 7: Reduction

- **Matrix normalization**
  - Same as vector normalization
  - \( N = \frac{V}{|V|} = \frac{V}{\sqrt{V^2}} \)

- **Use shuffle to reduce**

- **TODO**
  - Fill in missing sum reduction
  - Fill in missing division
    - Hint: remember rcp?
Exercise 8: Vector formats

- SSE 2 allows double precision floating point SIMD operations
- See emmintrin.h
- formats.cpp
  - Reimplement matrix normalization with packed double instead of packed single
  - One instruction is missing?
- Pfmon
  - Count PS and PD instructions
Exercise 9: Alignment

- alignment.cpp: See TODO
- Align memory access on SSE register width boundary (16 bytes)

Example

- Replace _mm_loadu_ps with _mm_load_ps
- _mm_malloc
  - See mm_malloc.h
Exercise 9: Alignment

- Is there any difference in performance?
  - If so, then why?
  - Investigate with pfmon
  - Try increasing the size of the grid
Conditional branches are impossible in SIMD
- If you branch into n streams, then it's not SIMD any longer
- Which of the elements would you branch on?

Also, branches are bad
- Mispredicted branches flush the pipeline

Solutions
- Conditional moves
- Masks
Example: A fruit basket

For each fruit

\[
\text{if(is\_fruit(fruit) \&\& !has\_seeds(fruit)) eat(fruit)}
\]

<table>
<thead>
<tr>
<th>fruit</th>
<th>apple</th>
<th>orange</th>
<th>pear</th>
<th>carrot</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_fruit</td>
<td>0xFF</td>
<td>0xFF</td>
<td>0xFF</td>
<td>0</td>
</tr>
<tr>
<td>has_seeds</td>
<td>0xFF</td>
<td>0</td>
<td>0xFF</td>
<td>0</td>
</tr>
<tr>
<td>eat = is_fruit &amp; ~has_seeds</td>
<td>0</td>
<td>0xFF</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fruit = fruit &amp; ~eat</td>
<td>apple</td>
<td>0</td>
<td>pear</td>
<td>carrot</td>
</tr>
</tbody>
</table>
Exercise 10: Control flow (1)

- Example: Conway's Game of Life
  - control_flow.cpp
- Remove control flow
  - Solutions
    - Use masks
    - Avoid
Exercise 10: Control flow (2)

- **Remove control flow (stream)**
  - **First**: Try to remove all conditional branches in cell survival logic
    - Hint: use masks
  - **Second**: Try to remove all boundary checks
    - Hint: avoid control flow – don't use masks

- **Pfmon**
  - Compare branching behaviour in naïve and streaming implementations
  - What is the mispredicted / branches ratio?
Exercise 10: Control flow (3)

- Implement SIMD
  - What precision do we need?
    - Each cell is either dead (0) or alive (1)
  - Lowest SSE precision is 8 bits
    - Is it worth or possible to go lower than that?

- Extra challenges
  - Align memory accesses
  - Is it necessary to iterate each cell?
Exercise 11: Prefetch

- Memory access is expensive
- Superscalability: While we are waiting to finish loading from memory we can do other useful stuff in parallel
- SSE allows explicit prefetch from memory
  - Load a memory address that you expect to use in the future into cache
  - Difficult to gain any speedup, since the CPU prefetches automatically
Intrinsic: \texttt{\_mm\_prefetch}

- See \texttt{xmmintrin.h}

When does it pay off?

- Try different grid sizes
- Try prefetching for different strides into the future

Pfmon

- What impact does it have on memory events?

Try prefetch in the previous exercise
Part 4
**Future SIMD**

- **Forward-scaling for future architecture**
  - Many cores
  - Wider vector registers
    - e.g. AVX: 256 bits, new instructions

- **Forward-portable code**
  - Ct :)
  - C++ classes :)
  - Intrinsics :|
  - ASM :(
- Not an extension to C++
- Platform independent
- Ct specific containers, similar to valarray, with arbitrary vector lengths
  - e.g. TVEC2D
- Exploits both SIMD and hardware thread parallelism
TVEC2D<U8> current(grid, x_size, y_size);

TVEC2D<U8> neighbors = leftShiftPermute(current, 1) + rightShiftPermute(current, 1) ...

current = TVEC2D<U8>(neighbors == 3 || (neighbors == 2 && current == 1));
Many-core

- Hardware concurrency in instruction streams is increasing faster than other concurrency
- We need to think n-way parallelism
  - Not 2, 8 or 32
- We may need to search for the lowest level of parallelism in our algorithms
  - For SIMD, data parallelism is often necessary
    - Hard synchronization
Mapping to parallel hardware

large vector, data parallel operation

If vector is large enough, use multiple cores

Use SSE
Explicit vectorisation is sometimes necessary
  - Can't always trust the compiler to vectorise for you

Memory organization needs attention

Control flow needs attention

Custom vector types (classes) with operator overloading is an efficient and portable method
□ A lot of time (= money) can be saved by properly optimizing parallel code

□ Track fitter example

□ Example vectorization speedup from scalar double to packed single: 3.7

□ Example multithreading speedup on 8 cores: 7.2

□ Total w/ both optimizations: $3.7 / 0.12 = 30$

□ Proportional speedup increase can be expected with future architectures
A proactive approach to SIMD

- Proactive algorithm design
- Future-scalability
- Future-portability
Further reading

- Intel IA-32 software developer's manual
- Google “AVX”
- Google “LRB”
Competition

- Create the fastest GOL implementation
- Prize:

Rules

- Has to be correct
- Use any trick in the book
  - SIMD, prefetch, etc.
  - Multi-threading
  - Skipping cells
  - Performance is measured on a neutral machine
- Must handle arbitrary #iterations and x and y sizes