Awareness of Architecture in Programming: Intel Sandy Bridge Memory and Vectorization (Case Study)

Oscar Peredo

FCFM, DCC, Universidad de Chile

2015 - Santiago, Chile

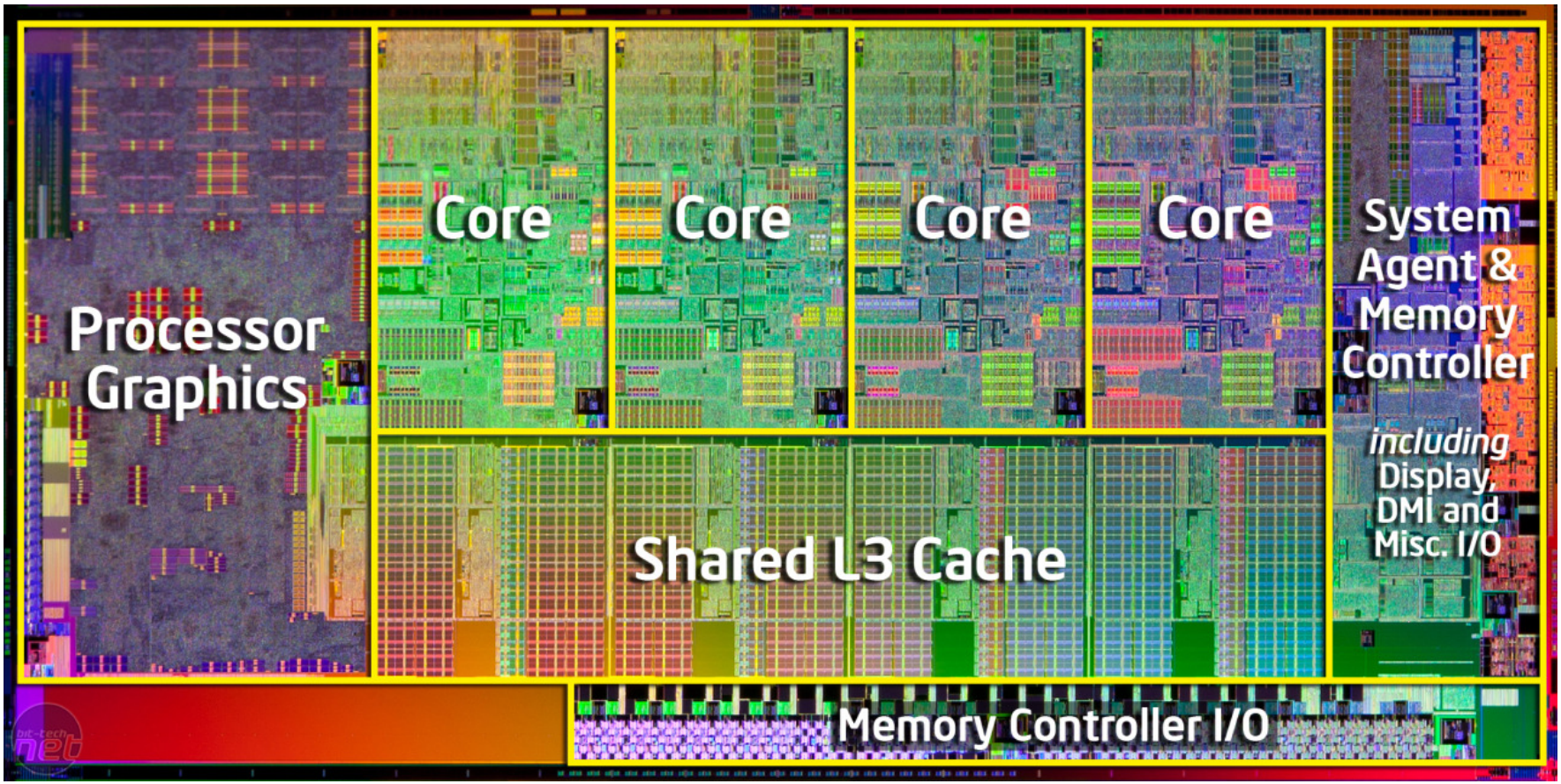
Outline

Outline

Introduction

- Real scenario: Intel Sandy Bridge Microarchitecture.
- Intel Sandy Bridge Microarchitecture.
- Review of Intel 64 and IA-32 Architectures Optimization Reference Manual:
  - Optimizing memory accesses.
  - Vectorization.
  - We focus in high-level optimizations.
- Our second approach to understand technical documents on code optimization (for Intel processors).

Intel Sandy Bridge

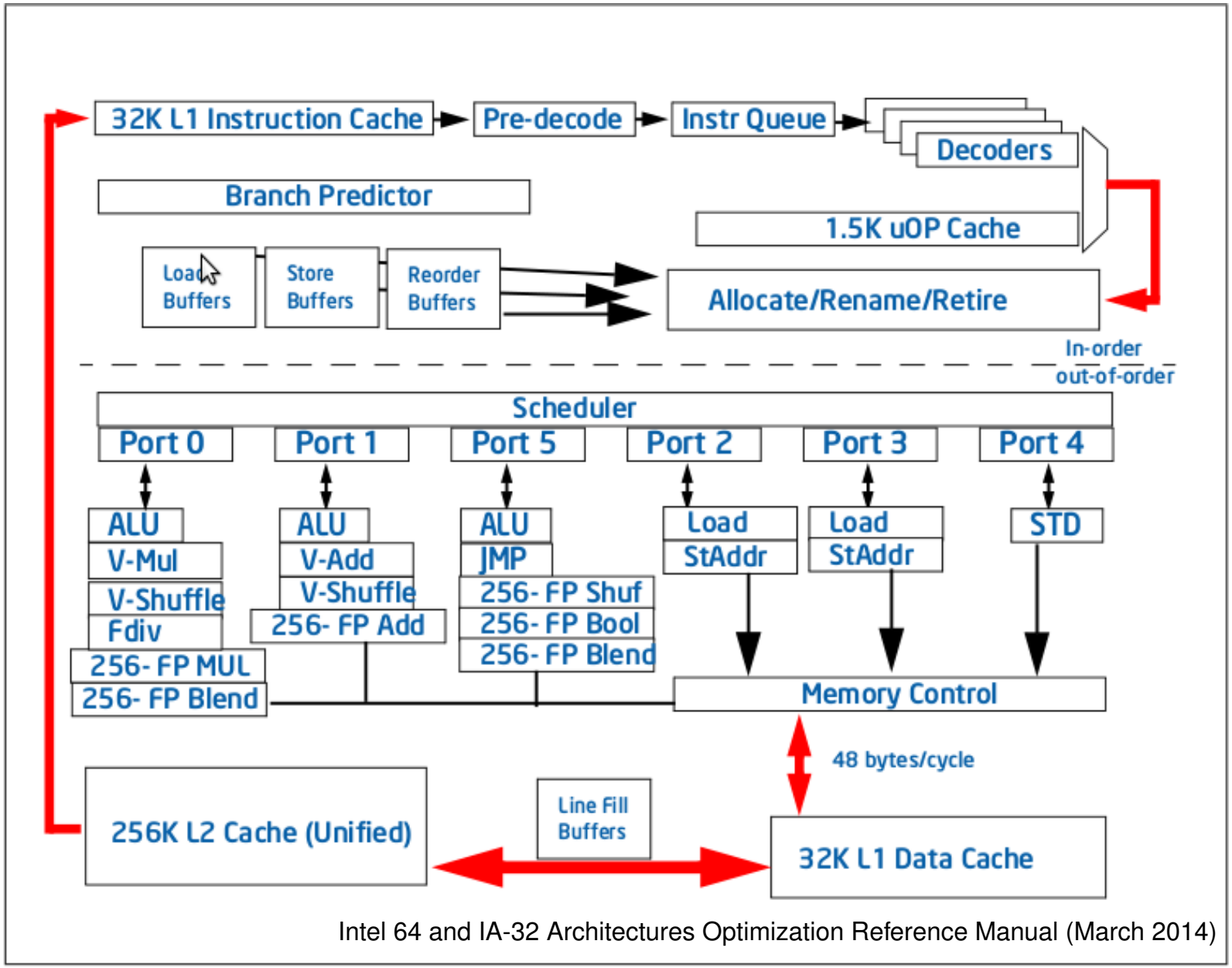


http://techreport.com/review/20188/intel-sandy-bridge-core-processors

Outline

Outline

Pipeline overview



Intel 64 and IA-32 Architectures Optimization Reference Manual (March 2014)

Pipeline overview

The pipeline consists of:

- An in-order issue **front end** that fetches instructions and decodes them into micro-ops (micro-operations). The front end feeds the next pipeline stages with a continuous stream of micro-ops from the most likely path that the program will execute.
- An **out-of-order, superscalar execution engine** that dispatches up to six micro-ops to execution, per cycle. The allocate/rename block reorders micro-ops to *dataflow* order so they can execute as soon as their sources are ready and execution resources are available.
- An in-order retirement unit that ensures that the results of execution of the micro-ops, including any exceptions they may have encountered, are visible according to the original program order.

Outline

Memory Hierarchy

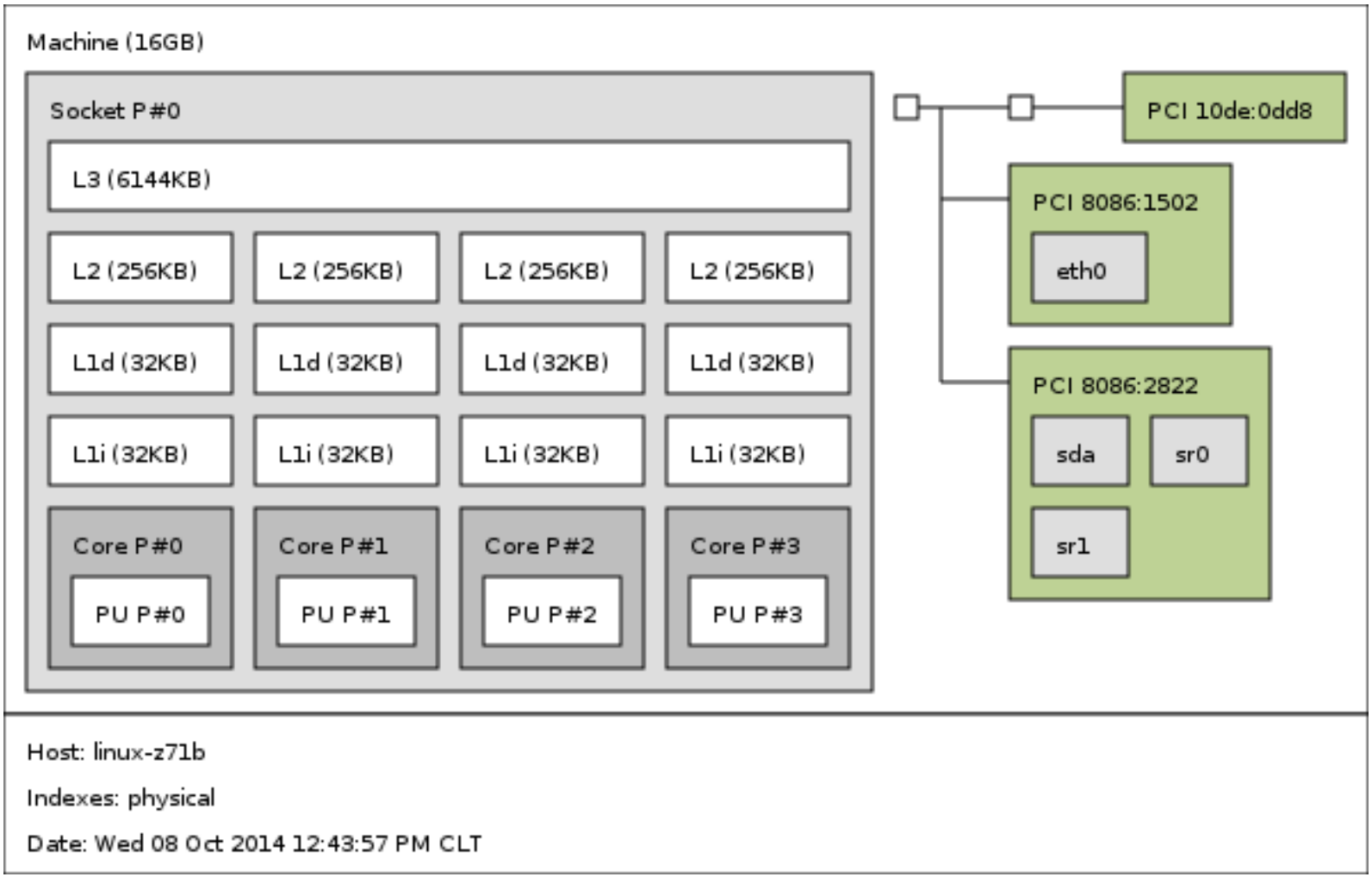
- Instruction, L1 Data (L1D) and L2 Cache per core
- **L1D may be shared by 2 logical processors if Hyper-Threading is supported.**
- L2 is shared by instructions and data.
- Last-level cache (LLC) is shared by all cores in the physical processor package by a ring connection.
- ITLB, DTLB ans STLB (shared) are used to translate linear to physical addresses.

Table 2-10. Cache Parameters

Level	Capacity	Associativity (ways)	Line Size (bytes)	Write Update Policy	Inclusive
L1 Data	32 KB	8	64	Writeback	-
Instruction	32 KB	8	N/A	N/A	-
L2 (Unified)	256 KB	8	64	Writeback	No
Third Level (LLC)	Varies, query CPUID leaf 4	Varies with cache size	64	Writeback	Yes

Memory Hierarchy

Diagram generated with `hwloc`:



Aside: Hyper-Threading (HT) technology

- Enables software to take advantage of task-level, or thread-level parallelism by providing multiple logical processors within a physical processor package.
- **In its first implementation in Intel Xeon processor, Hyper-Threading Technology makes a single physical processor appear as two logical processors.**
- The two logical processors each have a complete set of architectural registers while sharing one single physical processor's resources. By maintaining the architecture state of two processors, an HT Technology capable processor looks like two processors to software, including operating system and application code.
- Reality: the individual threads cannot really run concurrently...

Aside: Hyper-Threading (HT) technology (Drepper, 2007)

How it works:

- The CPU is responsible for time-multiplexing the threads.
- This is not so new...
- The real advantage is that the CPU can schedule another hyper-thread and **take advantage of available resources such as ALUs when the currently running hyper-thread is delayed.**
- **In most cases this is a delay caused by memory accesses.**
- The cache hit rate of an application defines its capability to speedup using HT.

Aside: Hyper-Threading (HT) technology (Drepper, 2007)

The execution time can be modeled as:

$$T_{exe} = N * \{(1 - F_{mem})T_{proc} + F_{mem}(G_{hit}T_{cache} + (1 - G_{hit})T_{miss})\}$$

with:

$$\begin{aligned} N &= \text{Number of instructions.} \\ F_{mem} &= \text{Fraction of N that access memory.} \\ G_{hit} &= \text{Fraction of loads that hit the cache.} \\ T_{proc} &= \text{Number of cycles per instruction.} \\ T_{cache} &= \text{Number of cycles for cache hit.} \\ T_{miss} &= \text{Number of cycles for cache miss.} \\ T_{exe} &= \text{Execution time for program.} \end{aligned}$$

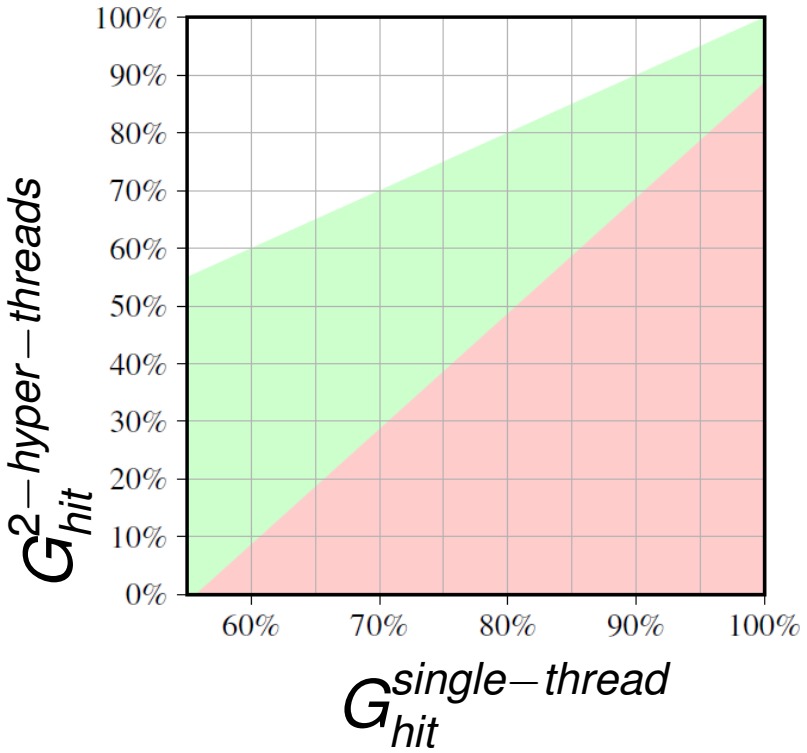
We want that

$$T_{exe}^{single-thread} = 2 * T_{exe}^{2-hyper-threads}$$

so the only variables are $G_{hit}^{single-thread}$ and $G_{hit}^{2-hyper-threads}$...

Aside: Hyper-Threading (HT) technology (Drepper, 2007)

We can solve the previous equation, obtaining solutions in green area. For values of $G_{hit}^{single-thread} < 55\%$, the program can benefit in all cases from HT execution. The program is more or less idle enough due to cache misses to enable running a second hyper-thread.



The white area can never be reached (upper bound for hyper-threads) and the red area doesn't solve the equation.

Aside: Hyper-Threading (HT) technology (Drepper, 2007)

- A single-thread program with hit rate of 60% requires a hit rate of +10% in the 2-hyper-threaded program. This is easy.
- A single-thread program with hit rate of 95% requires a hit rate of +80% in the 2-hyper-threaded program. This is harder.
- Why it is harder? Because the cache is cut in half, so the hit rate for both hyper-threads hardly will be high.
- **Hyper-threads are therefore only useful in a limited range of situations.**

Outline

Vectorization

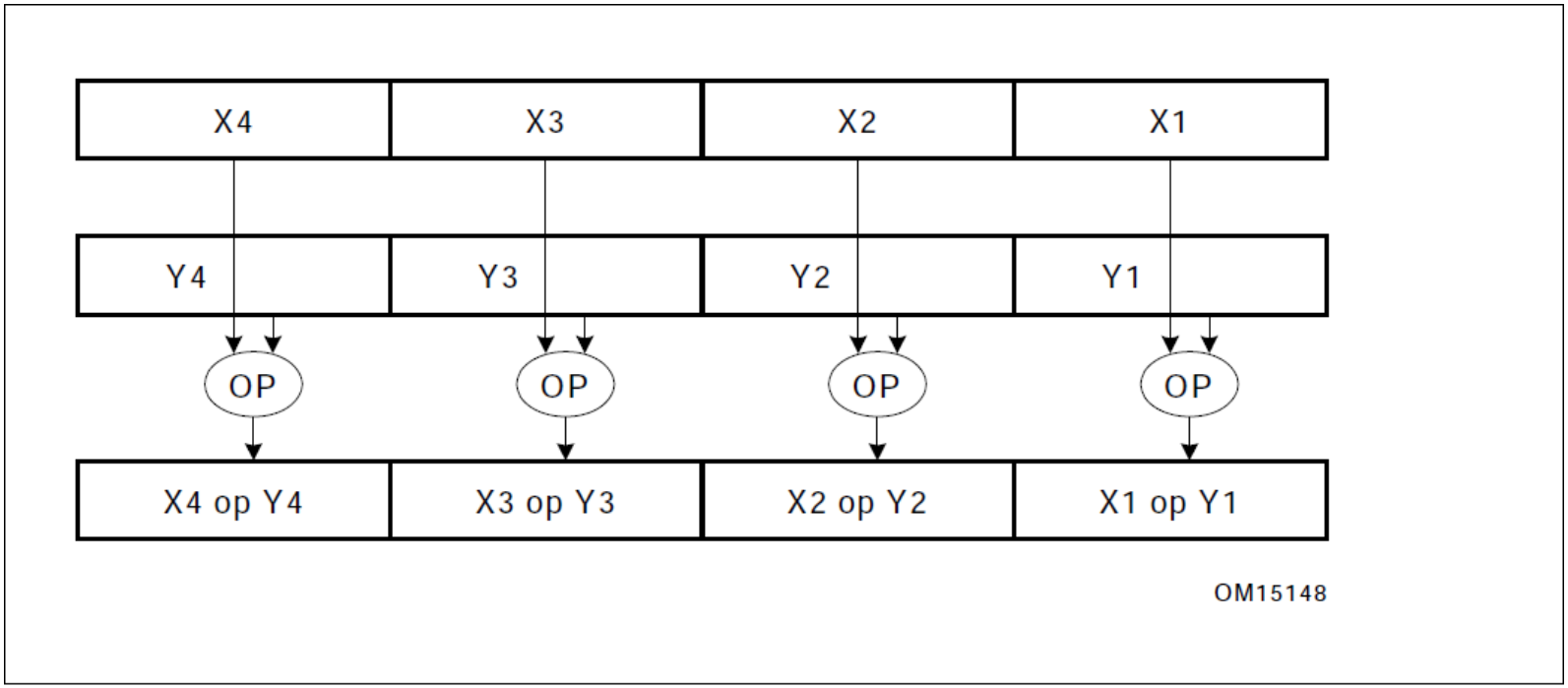
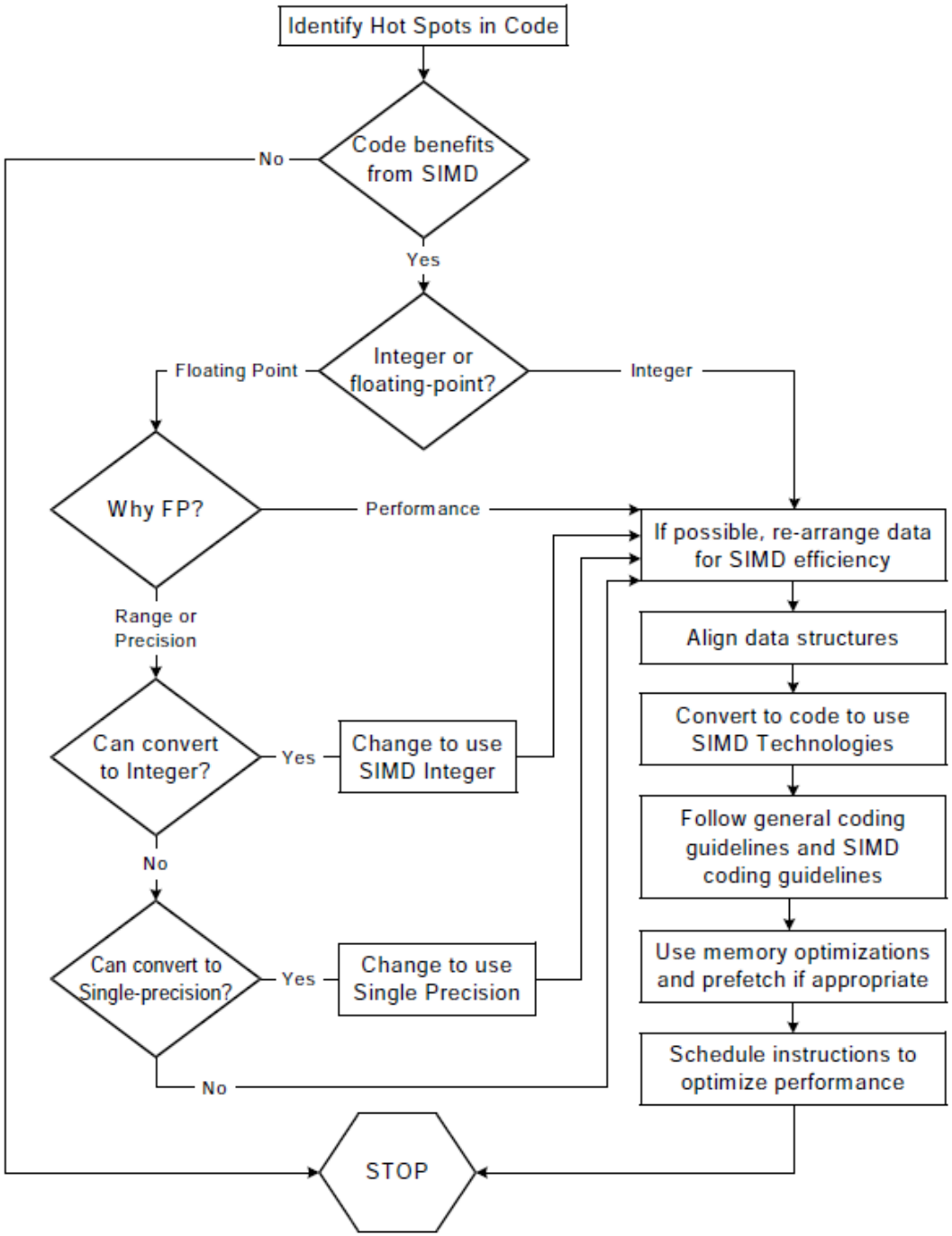


Figure 2-14. Typical SIMD Operations


Identify Hot Spots in Code
Code benefits from SIMD
No
Yes
Integer or floating-point?
Floating Point
Integer
Why FP?
Performance
Range or Precision
If possible, re-arrange data for SIMD efficiency
Align data structures
Convert to code to use SIMD Technologies
Follow general coding guidelines and SIMD coding guidelines
Use memory optimizations and prefetch if appropriate
Schedule instructions to optimize performance
Can convert to Integer?
Yes
Change to use SIMD Integer
No
Can convert to Single-precision?
Yes
Change to use Single Precision
No
STOP

Code Benefits by Conversion to SIMD Execution?

- Identifying code that benefits by using SIMD technologies can be time-consuming and difficult.
- Likely candidates:
  - Speech compression algorithms and filters
  - Speech recognition algorithms
  - Video display and capture routines
  - Rendering routines
  - 3D graphics (geometry)
  - Image and video processing algorithms
  - Spatial (3D) audio
  - Physical modeling (graphics, CAD)
  - Workstation applications
  - Encryption algorithms
  - Complex arithmetics
- Generally, good candidate code is code that contains small-sized repetitive loops that operate on sequential arrays (sequential in memory) of integers of 8, 16 or 32 bits, SP 32-bit FP data, DP 64-bit FP data.

Coding techniques

To vectorize your code and thus take advantage of the SIMD architecture, do the following:

- **Determine if the memory accesses have dependencies that would prevent parallel execution.**
- ”Strip-mine” (blocking/tiling) the inner loop to reduce the iteration count by the length of the SIMD operations (for example, four for single-precision floating-point SIMD, eight for 16-bit integer SIMD on the XMM registers).
- Re-code the loop with the SIMD instructions.

Coding methodologies: Performance/Ease of programming

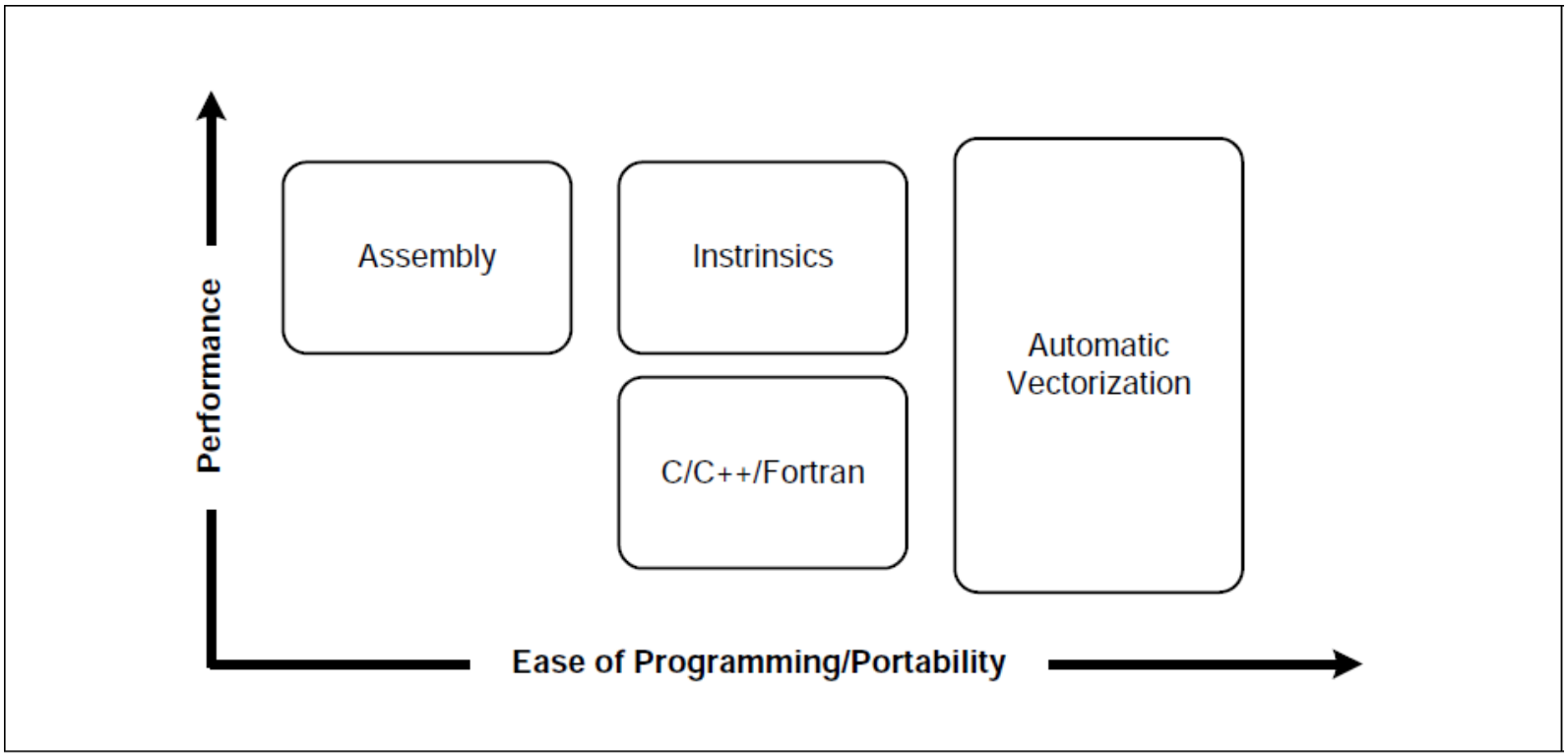


Figure 4-4. Hand-Coded Assembly and High-Level Compiler Performance Trade-offs

Coding methodologies: baseline example

Example 4-13. Simple Four-Iteration Loop

```
void add(float *a, float *b, float *c)
{
int i;
for (i = 0; i < 4; i++) {
    c[i] = a[i] + b[i];
  }
}
```

Coding methodologies: Assembly

Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
  __asm {
    mov     eax, a
    mov     edx, b
    mov     ecx, c
    movaps  xmm0, XMMWORD PTR [eax]
    addps   xmm0, XMMWORD PTR [edx]
    movaps  XMMWORD PTR [ecx], xmm0
  }
}
```

Coding methodologies: Intrinsics

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

Coding methodologies: C++ classes

Example 4-16. C++ Code Using the Vector Classes

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
        F32vec4 *av=(F32vec4 *) a;
        F32vec4 *bv=(F32vec4 *) b;
        F32vec4 *cv=(F32vec4 *) c;
                *cv=*av + *bv;
}
```

Coding methodologies: Auto-vectorization

Example 4-17. Automatic Vectorization for a Simple Loop

```
void add (float *restrict a,
          float *restrict b,
          float *restrict c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

- Compiler flags are needed (this code works with Intel compilers).
- The `restrict` qualifier in the argument list is necessary to let the compiler know that there are no other aliases to the memory to which the pointers point.

Outline

Outline

General issues

Data Layout Optimizations

- Pad data structures defined in the source code so that every data element is aligned to a natural operand size address boundary
- Try to arrange data structures such that they permit sequential access.

Data Layout Optimizations

Example 3-47. Rearranging a Data Structure

```
struct unpacked { /* Fits in 20 bytes due to padding */
    int        a;
    char       b;
    int        c;
    char       d;
    int        e;
};

struct packed {  /* Fits in 16 bytes */
    int        a;
    int        c;
    int        e;
    char       b;
    char       d;
}
```

Data Layout Optimizations

Example 3-48. Decomposing an Array

```
struct {        /* 1600 bytes */
    int    a, c, e;
    char b, d;
} array_of_struct [100];

struct {        /* 1400 bytes */
    int    a[100], c[100], e[100];
    char b[100], d[100];
} struct_of_array;


struct {        /* 1200 bytes */
    int    a, c, e;
} hybrid_struct_of_array_ace[100];


struct {        /* 200 bytes */
    char b, d;
} hybrid_struct_of_array_bd[100];
```

General issues

Capacity Limits and Aliasing in Caches
Pentium M, Intel Core Solo, Intel Core Duo and Intel Core 2 Duo processors have the following aliasing case:
Store forwarding: If a store to an address is followed by a load from the same address, the load will not proceed until the store data is available. If a store is followed by a load and their addresses differ by a multiple of 4 KBytes, the load stalls until the store operation completes.
Recommendations:

- Consider using a special memory allocation library with address offset capability to avoid aliasing.
- When padding variable declarations to avoid aliasing, the greatest benefit comes from avoiding aliasing on second-level cache lines, suggesting an offset of 128 bytes or more.

General issues

Locality Enhancement

- Optimization techniques such as blocking, loop interchange, loop skewing, and packing are best done by the compiler. Optimize data structures either to fit in one-half of the first-level cache or in the second-level cache; turn on loop optimizations in the compiler to enhance locality for nested loops.

Minimizing Bus Latency

- If there is a blend of reads and writes on the bus, changing the code to separate these bus transactions into read phases and write phases can help performance.
- To achieve effective amortization of bus latency, software should favor data access patterns that result in higher concentrations of cache miss patterns, with cache miss strides that are significantly smaller than half the hardware prefetch trigger threshold.

Outline

General issues

- Use the smallest possible floating-point or SIMD data type, to enable more parallelism with the use of a (longer) SIMD vector. For example, use single precision instead of double precision where possible.
- Arrange the nesting of loops so that the innermost nesting level is free of inter-iteration dependencies. Especially avoid the case where the store of data in an earlier iteration happens lexically after the load of that data in a future iteration, something which is called a lexically backward dependence.
- Avoid the use of conditional branches inside loops and consider using SSE instructions to eliminate branches.
- Keep induction (loop) variable expressions simple.

Outline

Bibliography

- Drepper, U., *What Every Programmer Should Know About Memory*, Red Hat Inc., 2007.
- Intel 64 and IA-32 Architectures Optimization Reference Manual (March 2014).
- Hennessy, J. and Patterson, D., *Computer Architecture: A Quantitative Approach*, Morgan-Kauffman 4th Edition, 2007.

Thanks for your attention!

Contact: operedo [at] gmail.com