Introduction to IA-32 and IA-64: Architectures, Tools and Libraries

G. Hager

Regionales Rechenzentrum Erlangen (RRZE)
HPC Services

Outline

- IA-32 Architecture
  - Architectural basics
  - Optimization with SIMD operations
  - Cluster computing on IA-32 basis
- IA-64 Architecture
  - Intel's EPIC concept
  - Available system architectures
  - Peculiarities of IA-64 application performance
- Libraries
  - Optimized BLAS/LAPACK
- Tools
  - Intel IA-32 and IA-64 compilers
  - VTune Performance Analyzer
IA-32 Architecture Basics – a Little History

- IA-32 has roots dating back to the early 80s
  - Intel’s first 16-bit CPU: 8086 with 8087 math coprocessor (x86 is born)
  - Even the latest Pentium 4 is still binary compatible with 8086
- Loads of advances over the last 20 years:
  - Addressing range (1MB → 16MB → Terabytes)
  - Protected mode (80286)
  - 32 bit GPRs and usable protected mode (80386)
  - On-chip caches (80486)
  - SIMD extensions and superscalarity (Pentium)
  - CISC-to-RISC translation and out-of-order superscalar processing (Pentium Pro)
  - Floating-point SIMD with SSE and SSE2 (Pentium III/4)
- Competitive High Performance Computing was only possible starting with Pentium III
  - Pentium 4 is today rivaling all other established processor architectures
IA-32 Architecture Basics

- IA-32 has a CISC instruction set
  - "Complex Instruction Set Computing"
  - Operations like "load value from memory, add to register and store result in register" are possible in one single machine instruction
  - Huge number of assembler instructions
  - Very compact code possible
  - Hard to interpret for CPU hardware, difficulties with out-of-order processing
- Since Pentium Pro, CISC is translated to RISC (called μOps here) on the fly
  - "Reduced Instruction Set Computing"
  - Very simple instructions like "load value from memory to register" or "add two registers and store result in another"
  - RISC instructions are held in a reorder buffer for later out-of-order processing

IA-32 Architecture Basics: Pentium 4 Block Diagram

Source: http://www.top500.org/ORSC/
IA-32 Architecture Basics

- Pentium 4 seems to be quite an ordinary processor. It has
  - caches (instruction/data)
  - register files
  - functional units for integer & FP operations
  - a memory bus

- **What’s special about the Pentium 4?**
  - high clock frequency (currently 3.2 GHz)
  - L1 instruction cache is a "trace cache" and contains pre-decoded instructions
  - double speed integer units
  - special SIMD (Single Instruction Multiple Data) functional units and registers enable „vector computing for the masses“

Result: It's very hard to beat a Pentium 4…

---

IA-32 Architecture Basics:
What Makes the Pentium 4 so fast?

- **Very high clock frequencies**
  - every pipeline stage is kept as simple as possible, leading to very long pipelines (20 stages)
  - L1 data cache is tiny but fast (only 2 cycles latency)
  - lower instruction level parallelism (ILP) than previous IA-32 designs

- **L1 instruction cache**
  - long pipelines lead to large penalties for branch mispredictions
  - L1I cache takes pre-decoded instructions in order to reduce pipeline fill-up latency
  - P4 can very accurately predict conditional branches

- **Integer add, subtract, inc, dec, logic, cmp, test all take only 0.5 clock cycles to execute**
  - latency and throughput are identical for those operations
Before the advent of SIMD, IA-32 had a stack-based FP engine ("x87")

- Operations were done with stack top or stack top and another stack register
- Result always on stack top
- Advantage: easy tp program (trivial register allocation – UPN-like)
- Drawback: hard to do instruction-level parallelism
- All other modern architectures use a flat register model

Intel decided to kill two birds with one stone and combine flat register addressing with SIMD capability!

First SIMD implementation: Pentium MMX
- SIMD registers shared with FP stack
- Switch between SIMD and FP mode was expensive overhead
- Nobody should use this any more

"Sensible SIMD“ came about with SSE (Pentium III) and SSE2 (Pentium 4) – Streaming SIMD Extensions
- Register Model:
- Each register can be partitioned into several integer or FP data types
  - 8 to 128-bit integers
  - single (SSE) or double precision (SSE2) floating point
- SIMD instructions can operate on the lowest or all partitions of a register at once
IA-32 Architecture Basics: Floating Point Operations and SIMD

- Possible data types in an SSE register

  - 16x 8bit
  - 8x 16bit
  - 4x 32bit
  - 2x 64bit
  - 1x 128bit

  - 4x 32bit
  - 2x 64bit

- Example: Single precision FP packed vector addition

  - x_i and y_i are single precision FP numbers (4 per SSE register)
  - Four single precision FP additions are done in one single instruction
  - Pentium 4: 2-cycle latency & 2-cycle throughput for double precision SSE2 MULT & ADD leading to a peak performance of 2 (DP) FLOPs/cycle
In addition to packed SIMD operations, scalar operations are also possible:
- Operate on lowest item only!
- Alternative for non-vectorizable codes (still better than x87)

Caveat: SSE2 operations are carried out with 64-bit accuracy, whereas the x87 FP instructions use 80 bits:
- Slight numerical differences between x87 and SSE mode possible

Significant performance boost achievable by using SSE(2)!
- In cache
- Hardly any gain for out of cache performance expected
- See below for compiler options concerning SSE

IA-32 Architecture Basics:
Triads Performance: A(·) = B(·) + C(·) * D(·)
When given correct options, compiler will automatically try to vectorize simple loops
- Rules for vectorizability similar as for SMP parallelization or "real" vector machines
- See compiler documentation

SIMD can also be used directly by the programmer if compiler fails

Several alternatives:
- Assembly language
  For experts only
- Compiler Intrinsics
  Map closely to assembler instructions, programmer is relieved of working with registers directly.
- C++ class data types and operators
  High-level interface to SIMD operations. Easy to use, but restricted to C++.

Special C++ data types map to SSE registers

FP types:
- F32vec4: 4 single-precision FP numbers
- F32vec1: 1 single-precision FP number
- F64vec2: 2 double-precision FP numbers

Integer types: Is32vec4, Is64vec2, etc.

C++ operator+ and operator* are overloaded to accommodate operations on those types
- programmer must take care of remainder loops manually

Alignment issues arise when using SSE data types
- compiler intrinsics and command line options control alignment
- uncontrolled unaligned access to SSE data will induce runtime exceptions!
IA-32 Architecture Basics: Programming With SIMD Extensions

- A simple example: vectorized array summation
  - Original code (compiler-vectorizable):
    ```c
    double asum(double *x, int n) {
        int i;
        double s = 0.0;
        for(i=0; i<n; i++)
            s += x[i];
        return s;
    }
    ```
  - Hand-vectorized code:
    ```c
    #include <dvec.h>
    double asum_simd(double *x, int n) {
        int i; double s = 0.0;
        F64vec2 *vbuf = (F64vec2*)x;
        F64vec2 accum(0.,0.);
        for(i=0; i<n/2; i++)
            accum += vbuf[i];
        for(i=n&(-2); i<n; i++)
            s += x[i];
        return accum[0] + accum[1] + s;
    }
    ```
  - Compilation vectorized version is still faster than hand-vectorized; WHY?

- Alignment issues
  - alignment of arrays in SSE calculations should be on 16-byte boundaries
  - other alternatives: use explicit unaligned load operations (not covered here)
  - How is manual alignment accomplished?
  - 2 alternatives
    - manual alignment of structures and arrays with
      ```c
      __declspec(align(16)) <declaration>;
      ```
    - dynamic allocation of aligned memory (align=alignment boundary)
      ```c
      void* _mm_malloc (int size, int align);
      void _mm_free (void *p);
      ```
IA-32 Architecture Basics: Hyperthreading

- Hyper-Threading Technology enables multi-threaded software to execute tasks in parallel within each processor
- Duplicates architectural state allowing 1 physical processor to appear as 2 “logical” processors to software (operating system and applications)
- One set of shared execution resources (caches, FP, ALU, dispatch, etc.)
  - only registers and a few other things are duplicated

IA-32 Architecture Basics: Hyperthreading

- Idea behind HT: usually a large fraction of a CPU's resources are unused during execution
IA-32 Architecture Basics: Hyperthreading

- By duplicating architectural state, two threads or processes can share the resources of one CPU and put better use to them:

  ![Diagram of IA-32 Architecture Basics: Hyperthreading]

- What is the advantage of HT?
  - puts better use to one CPU's resources
  - CPU is faster under high workload (many processes/threads)
  - helps throughput, not performance

- What does HT not do?
  - can not speed up one single process/thread (can even slow it down)
  - does not give you more resources per CPU

- Who can benefit from HT?
  - workloads in which different threads use different functional units (e.g. integer & fp operations, respectively)

- Where is HT useless?
  - purely floating-point code that uses the FP units continuously
  - code which is very sensitive to cache size
  - spin waits do not free resources

- With suitable applications, speedups of 30% per node are possible
IA-32 System Architecture

- CPU comes in many flavours
  - Pentium 4
  - Xeon DP for dual-CPU systems with larger L2 cache (512 kB)
  - Xeon MP for multi-CPU systems, up to 16 today with L3 cache (up to 1 Mbyte)

- Memory Interface
  - currently: FSB800 (theoretical memory bandwidth of 6.4 Gbytes/s)
  - Common chipsets cover mainly single and dual CPU configurations
  - There is always only one path to memory in 2- and 4-CPU systems!

- A multitude of PCI, AGP, I/O configurations possible

IA-32 Clustering

- Due to its unrivaled price/performance ratio the Pentium 4 is very suitable for cluster computing on any scale
- "Poor man's supercomputer": Go to ALDI and buy a bunch of boxes and a Fast Ethernet switch (100 Mbit)
  - might be perfectly well suited for many applications
- Other end: Quadrics Elan3 interconnect
  - expensive, but at least 30 times more communication bandwidth than Fast Ethernet
  - far more than $1000 per node "just for the network"

- Common setups (compromise between speed and purse)
  - Myrinet (LRZ Linux Cluster)
  - Gbit Ethernet (RRZE Linux cluster)
IA-32 Cluster at RRZE

- 73 (soon 86) nodes (146/172 CPUs)
  - dual Xeon 2.66 GHz
  - 2GB RAM
  - Gbit Ethernet
  - 80 GByte local disk
- **Peak performance: 776 GFlops/s**
- 2 frontend nodes with 4 GB RAM
- NFS fileserver (approx. 700GB)
  - server traffic shares communication network
- Queueing System: **OpenPBS**
  - *serial* queue with 48h runtime
    (max. 1 node)
  - *parallel* queue with 24h runtime
    (max. 32 nodes)
  - *special* queue for high-priority projects
    (max. 73 nodes, unlimited runtime)
Itanium: Intel's new 64 Bit Architecture: What's so new about it?

- **It's not RISC:**
  Compiler generates **bundles** with three instructions each

  - ![Image of a 3-bit bundle]

  Length: 41 bits each = 123 bits

- **It's not VLIW:**
  Compiler generates a 5-Bit **template field** containing information about instruction level parallelism

  - ![Image of a 5-bit template field]

  - **This is called** **Explicitly Parallel Instruction Computing (EPIC)**

---

**Itanium: Intel’s new 64 Bit Architecture**

**Principles of EPIC**

- **Template field** provides information about
  - the instruction type / execution unit of each instruction
  - independent instruction groups (instruction level parallelism)

- **Template field** allows only a limited number of instruction combinations (12) in each bundle, e.g.:
  - a maximum of one Floating-Point-Instruction per bundle
  - a maximum of two Memory-Instruction per bundle

- **Template field** may include up to two stop operations (;;) to mark the end of an independent instruction group:

<table>
<thead>
<tr>
<th>Template</th>
<th>I0</th>
<th>I1</th>
<th>I2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0E</td>
<td>M</td>
<td>M</td>
<td>F</td>
</tr>
<tr>
<td>0F</td>
<td>M</td>
<td>M</td>
<td>F ;</td>
</tr>
<tr>
<td>0B</td>
<td>M ;</td>
<td>M</td>
<td>I ;</td>
</tr>
</tbody>
</table>
Itanium: Intel’s new 64 Bit Architecture
Principles of EPIC

- If instruction sequence does not fit one of the templates the compiler will pack one or two `nop` instructions into the bundle
- No correlation between (independent) instruction groups and beginning or end of bundles

---

1. Bundle1
2. Bundle2
3. Bundle3
4. Bundle4

Instruction group A Instruction group B IG C

---

Itanium: Intel’s new 64 Bit Architecture
Principles of EPIC

- Itanium1/Itanium2 can execute two bundles at each processor cycle
  - maximum of 6 instructions per cycle
  - 6 way "superscalar"
- No out-of-order execution
- Ability of compiler to determine (independent) instruction groups is crucial for Itanium processors
  - Use appropriate algorithms
  - Do not hide independence of operations
  - Use compiler directives (CDIR$ IVDEP)
Itanium2: Intel's current 64 Bit processor
Processor Specifications

- Processor frequencies (today): 1 GHz ... 1.5 GHz
- Functional units:
  - 6 INTEGER
  - 3 BRANCH
  - 2 FLOATING POINT (FMA) max. 4 Flop/Cycle
- Huge set of registers:
  - 128 Floating Point (82 Bit)
  - 128 Integer (64-Bit)
  - 64 Branch
  - 64 Predicate
  - Rotating Register / Register Stack

Itanium2: Intel's current 64 Bit processor
Processor Specifications

- L1 cache (instruction)
  - 16 kB; 4-way set associative; 64 byte cache line

- L1 cache (data)
  - 16 kB; 4-way set associative; 64 byte cache line
  - no floating point data (L1-cache bypass)
  - Latency: 1 cycle / write through
  - 4 references per cycle

- L2 cache (unified)
  - 256 kB; 8-way set associative; 128 byte cache line
  - Latency: 5 cycles (LD) ; 7 cycles (ST) / write back
  - 2 LD & 2 LD/ST per cycle
Itanium2: Intel’s current 64 Bit processor

Processor Specifications

- **L3 cache (unified)**
  - 1.5 MB ... 6 MB on die!
  - 6-way or 12-way set associative; 128 byte Cache line
  - Latency: 7 Cycles (ST) ; 12 cycles (LD) / write back

- **System/memory bus**
  - 128 Bits wide, running at 400 MHz ("double-pumped" 200 MHz)
  - Bandwidth: 6.4 GB/s (of which you can get 6.4)
  - Maximum of 18 outstanding Bus requests per CPU

- **Addressing**
  - 50 Bit physical / 64 Bit virtual
  - Max. page size: 4 GB (current Linux Kernel limit: max. 64 kB)

---

**Intel Itanium 2 Microarchitecture Block diagram**

H. Strauss, HP
Itanium2: Intel’s current 64 Bit processor
Processor Die (McKinley, 3MB L3)

Itanium Systems in use by RRZE

- **Itanium1:** 1 x SGI750
  - dual Itanium1 - 733MHz - 2MB L3
  - 2 GB RAM
  - Intel "white box"
  - test and compilation machine (newest compilers are always tested here first)

- **Itanium2:** 2 x HP zx6000
  - dual Itanium2 - 900 MHz – 1.5 MB L3
  - 10 GB RAM; HP zx1 Chipset; 3*73 GB SCSI
  - GB Ethernet

- Soon (end of October): SGI Altix 3700 with 28 CPUs (1.3 GHz) and 112 Gbyte shared memory
SGI Altix: ccNUMA Distributed Shared Memory

- **System Architecture**
  - building block: SC-Brick (4 CPUs) with two 2-CPU nodes

---

**NUMAlink3 2x1.6**

---

**System Architecture cont’d**
- NUMAlink3 network (like Origin 3000, but Fat Tree topology)
- Layout for 32-CPU system:

---

**NUMA placing** is important issue due to larger bandwidth imbalance as compared to Origin 3000!
Some dos and don’ts for Itanium2

- Do try to use FMA where possible; formulate your inner loops in an FMA-friendly way
- Avoid very short, tight inner loops; if a short trip count cannot be avoided, try to unroll outer loops to make the body fatter
- When working in L2 cache, try different array paddings; due to banked L2 layout, significant performance boosts can be achieved by not hitting the same bank in every loop iteration
- With current Intel compilers, try to avoid too many synchronization points in OpenMP programs – locks and barriers tend to be slow
- Use !DIR$ IVDEP when applicable (indirect array access)
High-Performance Libraries

- Important functionality for every architecture: optimized dense linear algebra (BLAS, LAPACK) and FFT libs
  - "vanilla code" from http://www.netlib.org/ is unsuitable performancewise
  - optimized versions available from Intel and other sources
- Intel's High Performance LAPACK/BLAS/FFT package: Math Kernel Library (MKL)
  - complete BLAS 1/2/3 and LAPACK3 implementation
  - FFT functions
  - commercial, but free (beer) for personal use
- Alternative: Goto's High Performance BLAS
  - approx. 10% faster than MKL for matrix-matrix operations
  - http://www.cs.utexas.edu/users/flame/goto/
- Intel's Integrated Performance Primitives (IPP)
  - special subroutines esp. for multimedia processing
Using Intel MKL

- Usually installed in `/opt/intel/mkl`
  - `-L/opt/intel/mkl/lib/{32,64}` required

- **dynamic linking on IA-32 or IA-64:**
  - `-lmkl_lapack64 -lmkl_lapack32 -lmkl -lguide -lpthread`

- **static linking on IA-32:**
  - `-lmkl_lapack -lmkl_ia32 -lguide -lpthread`

- **static linking on IA-64:**
  - `-lmkl_lapack -lmkl_ipf -lguide -lpthread`

- Watch for correct `LD_LIBRARY_PATH` (or equivalent compiler options) when using dynamic libs
- C headers for BLAS functions available: `mkl.h`

---

Using Intel MKL

- **MKL** is shared-memory parallelized

- Default: No parallel execution (starting with V6.0)
- `OMP_NUM_THREADS` environment variable determines number of threads at runtime
- Force serial mode (regardless of `OMP_NUM_THREADS`):
  - `MKL_SERIAL=YES`

- Calling MKL from OpenMP regions is ok – MKL will use serial mode.

- Calling parallel MKL from concurrent POSIX (OS) threads is not safe – use serial mode in this case
Intel Compilers

- Fortran95 and C++ compilers available from Intel
  - commercial, but free (beer) for personal use (yet)
- Nearly identical interface (command line options) for IA-32 and IA-64
- Fortran compiler can cope with little- and big-endian UNFORMATTED files
- GNU compiler compatibility and interoperability
- C++ compiler is up to modern standards
- Unrivaled Fortran performance, C++ has some room for improvements
- Included: Short Vector Math Library
  - automatically used by compiler to speed up vector calculations with transcendental and other functions
- Included: Command line debugger with parallel debugging support

Intel Compilers:
Basics of Usage

- Compiler executable
  - IA-32: ifc, icc  IA-64: efcc, ecc
- To set up correct paths and manpaths, source
  
  /opt/intel/compiler71/ia{32,64}/bin/{e,i}fcvars.[c]sh

  first

- Important environment variables at program runtime
  - TMP
determines directory for Fortran temporary files (scratch files)
  - F_UFMTENDIAN
  allows unit-specific runtime conversion of Fortran unformatted binary files
Endianness conversion for Fortran UNFORMATTED files

\[ F_{\text{UFMTENDIAN}} = \text{MODE} | [\text{MODE;} \text{ EXCEPTION}] \]

where:

\[ \text{MODE} = \text{big} | \text{little} \]

\[ \text{EXCEPTION} = \text{big:ULIST} | \text{little:ULIST} | \text{ULIST} \]

\[ \text{ULIST} = \text{U} | \text{ULIST,U} \]

\[ \text{U} = \text{decimal} | \text{decimal-decimal} \]

Examples:

\[ F_{\text{UFMTENDIAN}}=\text{big} \quad \text{file format is big-endian for all units} \]

\[ F_{\text{UFMTENDIAN}}=\text{big:9,12} \quad \text{big-endian for units 9 and 12, little-endian for others} \]

\[ F_{\text{UFMTENDIAN}}=\text{"big;little:8"} \quad \text{big-endian for all except unit 8} \]

Intel Compilers: Important Options

- Not processor-specific

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-g</td>
<td>include debugging information in binary; can be combined with optimization options</td>
</tr>
<tr>
<td>-qp, -p</td>
<td>compile for profiling with gprof</td>
</tr>
<tr>
<td>-f[no-]alias</td>
<td>assume there is [no] aliasing in program; esp. suitable for C(++) and F90</td>
</tr>
<tr>
<td>-openmp</td>
<td>enable OpenMP directives</td>
</tr>
<tr>
<td>-openmpS</td>
<td>compile OpenMP program as serial program; use stub OpenMP library</td>
</tr>
<tr>
<td>-syntax</td>
<td>check program syntax only; do not generate code</td>
</tr>
<tr>
<td>-ipo</td>
<td>enable interprocedural optimizations across files</td>
</tr>
</tbody>
</table>
### IA-32/IA-64 Introduction

#### Intel Compilers: Important Options

### Not processor-specific

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o3</td>
<td>high level optimizations (loop nest, prefetching, unrolling,…)</td>
</tr>
<tr>
<td>-opt_report</td>
<td>print optimization report to stderr</td>
</tr>
<tr>
<td>-opt_report_level [min</td>
<td>med</td>
</tr>
<tr>
<td>-opt_report_phase</td>
<td>report only for certain optimization phases</td>
</tr>
<tr>
<td>-opt_report_help</td>
<td>print available optimization phases to report on</td>
</tr>
<tr>
<td>-parallel</td>
<td>enables auto-parallelization of loops</td>
</tr>
<tr>
<td>-par_report_level</td>
<td>report on parallelization success with different verbosity (0..3, default: 1)</td>
</tr>
</tbody>
</table>

### IA-32 specific/IA-64 specific

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-tpp7</td>
<td>optimize for Pentium 4 and Xeon</td>
</tr>
<tr>
<td>-xW</td>
<td>use SSE2 extensions when possible; code will only run on SSE2 capable architectures</td>
</tr>
<tr>
<td>-vec[-]</td>
<td>enable [disable] vectorizer</td>
</tr>
<tr>
<td>-vec_reportn</td>
<td>print diagnostic information about vectorization; levels 0..5, default is 1</td>
</tr>
<tr>
<td>-rcd</td>
<td>use rounding instead of truncation for float-to-int conversions in C++; faster, but not standard-conforming</td>
</tr>
<tr>
<td>-ftz</td>
<td>flush denormals to zero; faster, but not IEEE compliant</td>
</tr>
<tr>
<td>-ivdep_parallel</td>
<td>when a loop is marked by !DIR$ IVDEP, assume there is no loop-carried dependency</td>
</tr>
</tbody>
</table>
Intel VTune Performance Analyzer

- For performance analysis, gprof will suffice in many cases
- For more advanced performance metrics than "time per function" other tools are required; some of them are free:
  - HW counter analysis from system level down to line level with Oprofile (http://oprofile.sourceforge.net/)
  - GNU coverage tool: gcov
  - Performance Counter Library (PCL): very flexible, but manual instrumentation required
- ...
- Intel VTune: Commercial tool (free trial versions available, time-limited)
  - Windows (GUI)
  - Linux (command line), GUI coming next year
  - remote sampling of Linux applications from Windows GUI
  - will be covered in separate tutorial (Thursday)

References

  - good introduction, must be complemented with compiler and architecture documentation
  - extremely detailed, suitable for assembler programmers
  - slightly outdated
  - tutorials, manuals, white papers, discussion forums etc.
- c't Magazine 13/2003, several articles (25th birthday of Intel's x86 architecture)