Optimization

Giannis Koutsou

Computational based Science and Technology Research Center (CaSToRC)
The Cyprus Institute

Lattice Practices 2012, 10th October 2012, Zeuthen

THE CYPRUS INSTITUTE

CaSToRC

Objectives

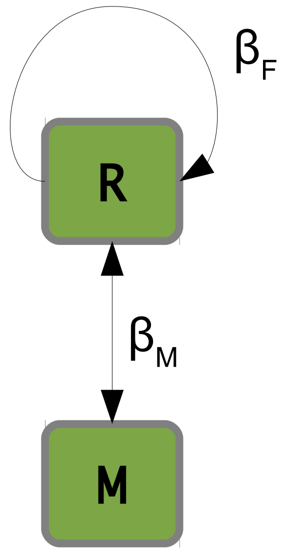
- **Understand/Expect performance**
  - Performance models
  - Connection to hardware specifications
- **Measure performance**
- **Improve performance (optimization)**
  - Single-core optimizations
    - Vectorization
  - Parallelization
    - Shared-memory (i.e. thread-level)
    - Distributed memory (i.e. message passing)

Understanding performance

- **Machine characteristics which influence performance**
  - Memory hierarchy
  - Floating-point rate
  - Bandwidths

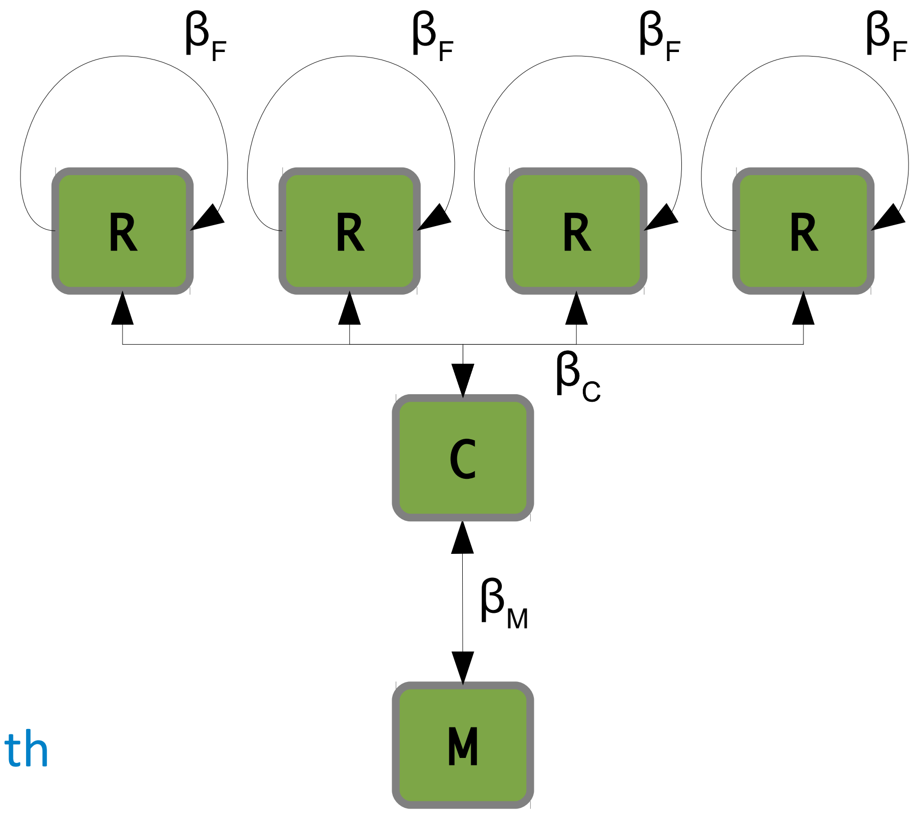
Understanding performance

- **Machine characteristics which influence performance**
  - Memory hierarchy
  - Floating-point rate
  - Bandwidths
- **Simple model**
  - R: register file
  - M: memory
  - β_F: Floating point rate
  - β_M: Memory bandwidth



Understanding performance

- **Machine characteristics which influence performance**
  - Memory hierarchy
  - Floating-point rate
  - Bandwidths
- **Simple model**
  - R: register file
  - C: cache
  - M: memory
  - β_F: Floating point rate
  - β_M: Memory-to-cache bandwidth
  - β_C: Cache-to-register file bandwidth



Understanding performance

- **Simple model for completion of a kernel *k* with on a hardware sub module *x*, with through-put β_x**

$$t_x^k = \frac{I_x^k}{\beta_x}$$

- **Example, double precision** $y[:] \leftarrow a * x[:] + y[:]$
  - Floating-point: 2 per element
  - Memory transfers: 2 in + 1 out = 3*8 = 24 bytes per element
  - Assume hardware with β_F = 12 Gflop/s and β_M = 24 Gbyte/s
  - t_{FP}= 1/6 ns per element
  - t_{MT}= 1 ns per element

Understanding performance

- Simple model for completion of a kernel *k* with on a hardware sub module *x*, with through-put β_x

$$t_x^k = \frac{I_x^k}{\beta_x}$$

- Example, double precision $y[:] \leftarrow a * x[:] + y[:]$
  - t_{FP}= 1/6 ns per element
  - t_{MT}= 1 ns per element
  - Assuming perfect overlap, execution time is the largest, i.e. 1 ns per element

Measuring performance

- For the examples here:
  - We know *I_x* for given kernel
  - We only measure *wall clock time*

- Alternatively
  - Read performance counter registers directly (e.g. via the PAPI library), not covered here

Measuring performance

- Reproducibility
  - Typically performance measurements “jitter”
  - Strictly speaking, one should repeat performance measurement to perform a statistical analysis
  - And perform many iterations of the same kernel and obtain an average time

Measuring performance

- Typical time-scales
  - Typical clock-rate O(1) GHz, or one cycle per ns
  - Typical bandwidths O(10) Gbyte/s
  - Clock granularity
    - Here we'll use `gettimeofday()`
    - Returns seconds and microseconds since fixed time (the Epoc)
    - Granularity of microseconds → O(seconds) benchmark runs for reliable measurements

Optimization

- Here we will talk about optimization on x86 architectures, though most of the items can be generalized to other architectures
- We will cover:
  - Single-core optimizations (vectorization)
  - Multi-core parallelization (OpenMP)
  - Some general info on the Message Passing Interface

Optimization

- Vectorization
  - Most processor architectures have some vector extensions for vectorized math operations
  - E.g. SSE (Intel), Altivec (PowerPC), QPX (BlueGene/Q) etc.
  - Single Instruction Multiple Data (SIMD) operations
    - One instruction is performed on vectors of data
  - SSE3,4 supports 128-bit wide vectors
    - 2 double-precision numbers
    - 4 single-precision numbers
  - SSE3,4 through-put: 1 multiply-then-add (madd) per cycle, $\beta_{FP} = 4$ DP flops per cycle or 8 SP flops per cycle

Optimization

- "Auto-vectorization" of compilers usually inadequate for complex numbers
- Assume double precision complex multiplication:

```
                      c.re = a.re*b.re − a.im*b.im
c = a*b ⇒
                      c.im = a.re*b.im + a.im*b.re
```

- If `a` and `b` are stored as `[re, im]` in DP SSE registers vectorization can become rather non-trivial
- In such cases it is useful to perform schedule analysis

Schedule analysis: SSE vectorization

```
c.re = a.re*b.re – a.im*b.im

c.im = a.re*b.im + a.im*b.re
```

- 3 load/stores
- 5 integer ops (move/shuffle)
- 3 floating point ops

`ra ← ld a`	`a.re`	`a.im`
`rb ← ld b`	`b.re`	`b.im`
`r3 ← movdup ra`	`a.im`	`a.im`
`r4 ← movd ra`	`a.re`	`a.im`
`r4 ← shuf r4`	`a.re`	`a.re`
`r3 ← mul r3, rb`	`a.im*b.re`	`a.im*b.im`
`r4 ← mul r4, rb`	`a.re*b.re`	`a.re*b.im`
`r3 ← shuf r3`	`a.im*b.im`	`a.im*b.re`
`r3 ← xor r3, sign`	`-a.im*b.im`	`a.im*b.re`
`r3 ← add r3, r4`	`(a*b).re`	`(a*b).im`
`c ← store r3`	`(a*b).re`	`(a*b).im`

Optimization

- Vectorization: Intel with gcc
  - Compiler provides “Intrinsics”, i.e. functions and types used to manipulate vector registers and issue vector instructions
  - Avoids the need to write inline assembly
  - Allows compiler certain chances of optimization
  - More portable than inline assembly

Optimization

- Vectorization: Intel with gcc `#include <xmmintrin.h>`
  - Intrinsic types, can be initialized like structures

Single precision	Double precision
`__m128 x = {a, b, c, d};`	`__m128d x = {a, b};`

  - Memory loaded in SSE registers must be 16-byte aligned

Static	Dynamic
`double x __attribute__((aligned(16))) = 22;`	`posix_memalign(&ptr, 16, size);`

Optimization

- **Vectorization**
  - Loading, storing, multiply, add

Single precision	Double precision
`__m128 _mm_load_ps(float *);`	`__m128d _mm_load_pd(double *);`
`_mm_store_ps(float *, __m128);`	`_mm_store_pd(double *, __m128d);`
`c = _mm_mul_ps(a, b);`	`c = _mm_mul_pd(a, b);`
`c = _mm_add_ps(a, b);`	`c = _mm_add_pd(a, b);`

  - Note: there is no explicit “madd” intrinsic (or assembly op-code)
  - You can hint for a “madd” by interchanging “mul” and “add”

Optimization

- **Vectorization**
  - Shuffle operations

Single precision	Double precision
`c = _mm_shuffle_ps(a, b, mask);`	`c = _mm_shuffle_pd(a, b, mask);`
`mask: _MM_SHUFFLE([0-3],[0-3],[0-3],[0-3])`	`mask: _MM_SHUFFLE2([0-1],[0-1])`

  - Instead of a mask, you can use the available macros "_MM_SHUFFLE" and "_MM_SHUFFLE2" readily

Optimization

c = a * b (DP, complex)

```
__m128d register ra = {a.re, a.im};
__m128d register rb = {b.re, b.im};
__m128d register si = {-1, 1};
__m128d register r3 = _mm_shuffle_pd(ra, ra, _MM_SHUFFLE2(0,0));
__m128d register r4 = _mm_shuffle_pd(ra, ra, _MM_SHUFFLE2(1,1));
r3 = _mm_mul_pd(r3, rb);
r4 = _mm_mul_pd(r4, rb);
r3 = _mm_shuffle_pd(r3, r3, _MM_SHUFFLE2(0,1));
r3 = _mm_mul_pd(r3, si);
r3 = _mm_add_pd(r3, r4);
_mm_store_pd(&c, r3);
```

Parallelization

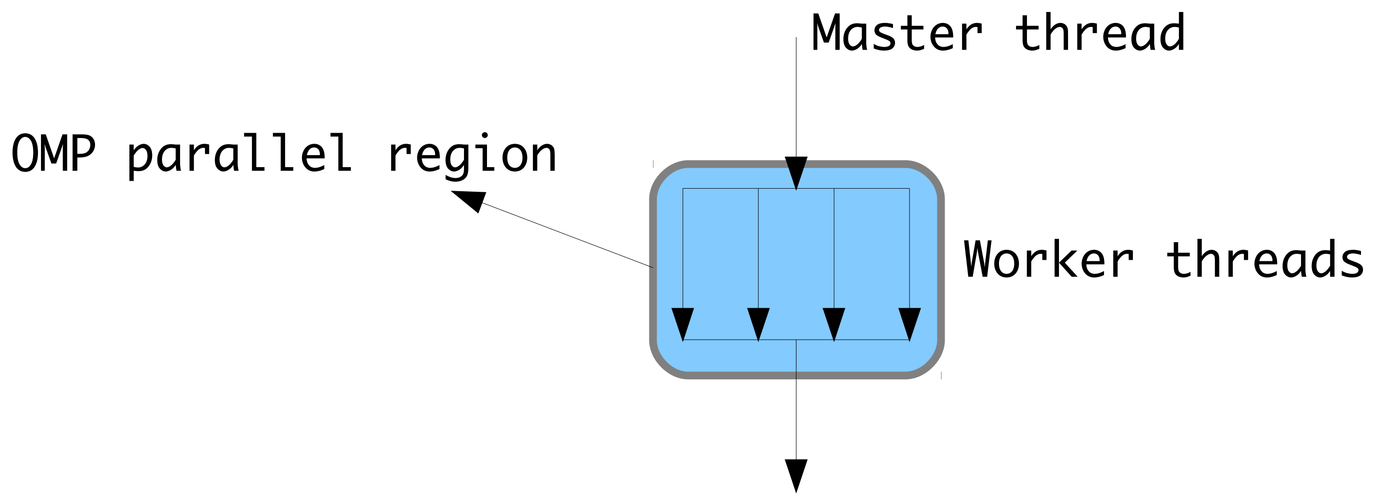
- Essential to take advantage of today's multi-core systems
- Two main distinctions:
  - Shared memory model
    - Processing elements share a common memory address space
    - E.g. multiple cores sharing the same RAM
  - Distributed memory model
    - Memory is distributed and sharing of data is done via communication
    - E.g. nodes in a cluster

Parallelization

- OpenMP
  - Shared-memory model
  - Allows for fast parallelization on-node
  - Can define private and shared data
  - Need to be careful when more than one thread accesses (writes) to same location

Parallelization

- OpenMP - parallelization
  - Simple pragma-based parallelization
  - Fork / Join model



Parallelization

- **OpenMP - parallelization**
  - **Works well for simple loops**
  -

	`#pragma omp parallel for`
`for(i=0; i<N; i++) {`	`for(i=0; i<N; i++) {`
`...`	`...`
`}`	`}`

- **With GCC, add `-fopenmp` to compiler arguments**
- **Control of number of threads**
  - **Run-time env. variable:** `OMP_NUM_THREADS`

Parallelization

- OpenMP - functions
  - Two important OpenMP functions

Function	Description
`int size = omp_get_num_threads();`	Returns number of threads
`int id = omp_get_thread_num();`	Unique id for each thread

- More at `openmp.org`

Parallelization

- **The Message Passing Interface**
  - MPI: An Application Programmer Interface (API)
  - A *de facto* standard for programming distributed memory systems
  - Current specification is version 2 (MPI-2)
  - Several free (open) implementations, e.g.:
    - Mvapich(2)
    - OpenMPI

The Library

- **Includes**
  - Function definitions, types, constants and macros for the MPI library are included in a single include file:

Fortran	C
`include “mpif.h”`	`#include <mpi.h>`

The Library

- **Compiling**
  - Compiling and linking is made easy with a wrapper-compiler which most implementations provide. Invocation is usually via:

Fortran	C

`mpif77 hello_world.f`
`mpif90 hello_world.f90`
	`mpicc hello_world.c`

Runtime

- **Running**
  - Running an MPI program is usually done via the `mpirun` or `mpiexec` wrapper scripts, which take care of initializing the appropriate environment for the parallel run:

Fortran and C
`mpirun -np 2 ./a.out`

Basic concepts

- **The distributed memory model**
  - Invocation of `mpirun` will run multiple instances of the *same* program in parallel
  - Without calls to MPI, all parallel instances will, ideally, run and terminate identically
  - With calls to MPI, one can:
    - Differentiate between parallel instances (i.e., give each instance, or *process,* a unique ID)
    - Synchronize processes
    - Send messages between processes

Basics

- **Initialization**
  - All MPI programs must begin with a call to `MPI_Init()` and close with a call to `MPI_Finalize()`.

Fortran	C

`CALL MPI_INIT(IERROR)`
`CALL MPI_FINALIZE(IERROR)`
	`ierror = MPI_Init(&argc, &argv);`
	`ierror = MPI_Finalize();`

  - In C the return value is always an integer error-code
  - Not invoking `MPI_Finalize()` at the end may raise an error
  - In C, the command line arguments must be passed to `MPI_Init()`.

Basics

- **Size and rank**
  - Get how many processes are running in a given *communicator*, and the rank of the calling process within that communicator.

Fortran	C

`CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROC, IERR)`
`CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)`
	`ierr = MPI_Comm_size(MPI_COMM_WORLD, &nproc);`
	`ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);`

  - The communicator `MPI_COMM_WORLD` is set to contain all processes available, after invocation of `MPI_Init()`
  - The integer `nproc` will be the number of processes within the communicator (should be the same as what was specified with `mpirun`)
  - Here, `MPI_Comm_rank()` is our first example where an MPI function gives a different result depending on the calling process. `rank` will be the rank of the calling process within the communicator: a number from 0 to `nproc`-1.

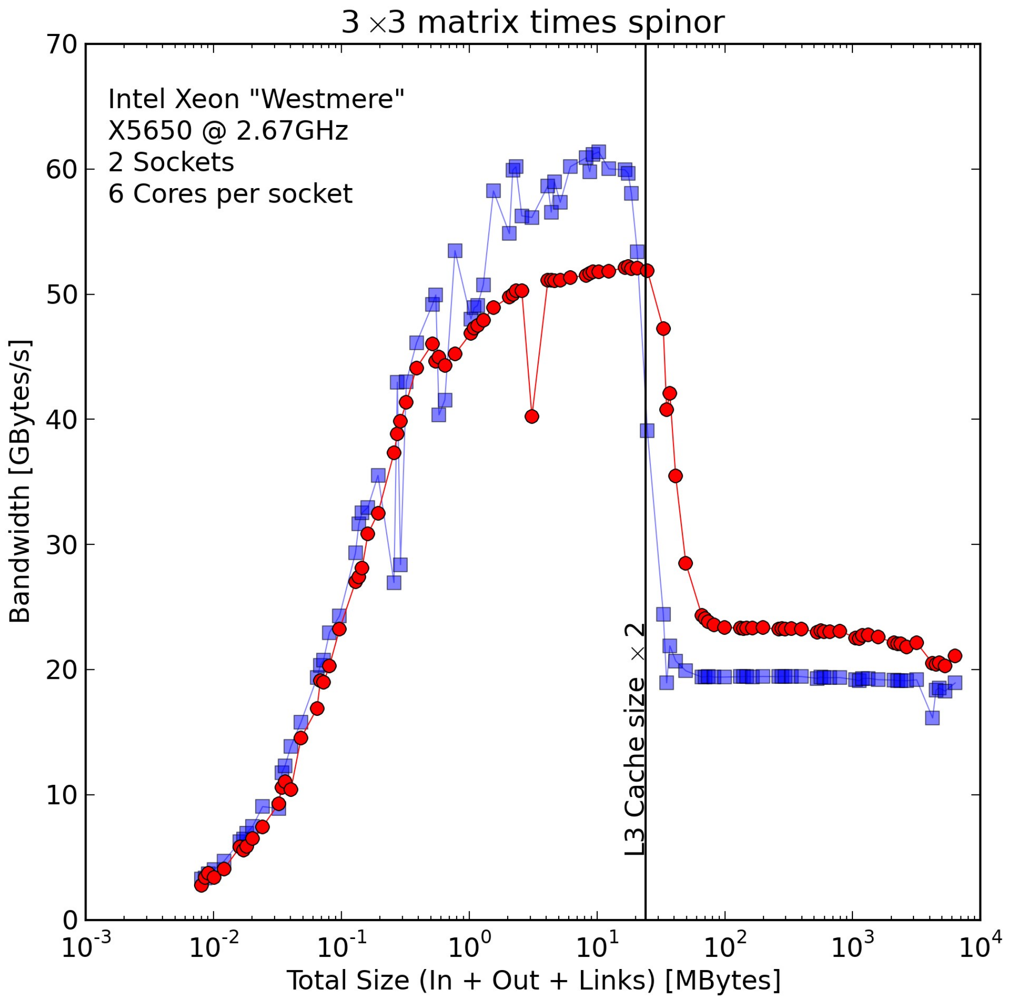
Basics

- **Reduction**
  - Perform an operation over data on all processes and store the result in one process

Fortran	`CALL MPI_REDUCE(A, B, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, MPI_COMM_WORLD, IERR)`
C	`ierr = MPI_Reduce(&a, &b, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)`

  - Perform `a` sum over the double-precision variable a over all processes and place the result into `b` on process `0`.
  - The fifth argument (`MPI_SUM`) is an MPI handle to the operation (can e.g. be sum, prod, sub, or, etc.)

Putting it all together



- Kernel: array of 3x3 complex times array of 3x4 complex
- Kernel is 0.5 Flops/Byte, meaning a BW-bound on a single node
- Kernel is OpenMP parallelized
- Blue for non-vectorized, red for vectorized
- Better saturation of BW (and thus FP) with vectorized instructions

Exercises

pax8

Pax8 nodes: Intel X5560	
Clock rate:	2.8 GHz
Peak DP:	1 SSE madd per cycle => 11.2 Gflop/s [core]
Cores per socket:	4
Sockets per node:	2
Bandwidth to memory:	32 Gbyte/s [shared between sockets]
L3 Cache size:	8 Mbytes [per socket, shared between cores]

Exercises

- There are 6 exercises, under directories: Ex1, Ex2, …, Ex6
- The source files for each exercise are incomplete. Look for "`TODO`" tags in the comments for instructions on what to do
- Ex1 – Ex3 deals with a complex zaxpy operation
- Ex4 – Ex6 deals with a gauge-times-spinor operation

Exercise 1

- **Ex1 has an (unoptimized) zaxpy operation set-up**
  - zaxpy: complex y ← a*x+y
- **The makefile will make both a double precision and single precision binary for you**
  - `main`: double precision binary
  - `mainf`: single precision binary
- **You need to:**
  - Place calls to `stop_watch()`, defined in `utils.[ch]` to time the `spinor_zaxpy()` function
  - Report the right bandwidth sustained in the `printf` statement

Exercise 1

- **If you believe you have corrected the code**
  - Compile by invoking `` `make` ``
  - Run the script: `./run.sh`
- **The script runs the benchmark for various combinations of array sizes and repetitions, both for SP and DP**
- **The results are stored in** `zaxpy.dat` **and can be plotted using gnuplot and the file** `zaxpy.gpl`

Exercise 2

- **Ex2 contains the same files as Ex1**
- `spinor_zaxpy.c` **has been modified**
  - Read the comments carefully
  - You need to write the main loop-body for vectorizing the single precision and double precision zaxpy operation
  - You may follow the schedule given to you in the comments
- **Once done, compile and run as before**

Exercise 3

- Ex3 follows Ex2
- Here, you need to modify `spinor_zaxpy.c` to parallelize the loop with OpenMP
- You also need to modify the Makefile to add the appropriate compiler flags
- You can run again as before. You may also change the number of OMP threads from inside the script

Exercise 4

- Ex4 is similar to the previous exercises in structure
- The kernel being measured now is the multiplication of an array of 3x3 matrices times a spinor:

$$\psi_{\mu}^{a}(x) \leftarrow u^{ab}(x)\,\chi_{\mu}^{b}(x)$$

- In Ex4, an unoptimized version of this kernal is set-up (only double precision)
- You need to count the bytes read/written per site and the floating point operations per site and report these in the `printf` statement in `main.c`
- Again, you can run and plot as in the previous exercises

Exercise 5

- **Ex5 is an extension of Ex4**
- **OMP pragmas have been added to parallelize the loop over the vector length**
- **A function `mul_su3_spinor_intrins()` has been added to `mul_su3_spinor.c`**
  - This new function contains an incomplete vectorized version of the matrix-vector multiplication
  - You need to understand how the vectorization is being done and complete the function

Exercise 5

- The main program compares the result of the vectorized function with the non-optimized one
- If correct, you should see diffs not larger than 10^{-32} when running the program
- As before, you can run and plot the timings as a function of the vector length

Exercise 6

- Ex6 builds upon Ex5
- The code here demonstrates a minimal MPI-parallelized program
- You need to calculate the bandwidth and flop-rate to be reported in `printf`
- You need to also sum over MPI processes the kernel timings to obtain an average
- You can run and plot the data as before
- Note how `run.sh` is set-up. You can experiment with different numbers of omp threads-per-process