Utilize your CPU power

Cache optimizations and SIMD instructions

Mario Mulansky

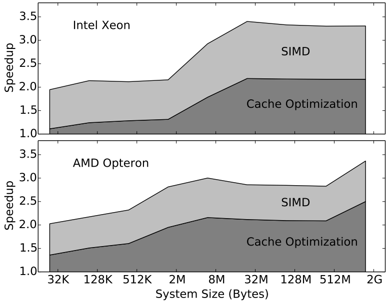
ISC-CNR, Institute for Complex Systems, Florence, Italy

Utilize your CPU power

Cache optimizations and SIMD instructions

Mario Mulansky

ISC-CNR, Institute for Complex Systems, Florence, Italy



Performance

Numerical Simulations:

- Flops Flops Flops
- Actually: Simulation runtime
- Not: DB queries / Requests per second, Response rate, ...

Performance

Numerical Simulations:

- Flops Flops Flops
- Actually: Simulation runtime
- Not: DB queries / Requests per second, Response rate, ...

Performance bounds:

- CPU power – Floating point operations per second: Flops/s
- Memory bandwidth

Performance

Numerical Simulations:

- Flops Flops Flops
- Actually: Simulation runtime
- Not: DB queries / Requests per second, Response rate, ...

Performance bounds:

- CPU power – Floating point operations per second: Flops/s
- Memory bandwidth

This talk:

- Bandwidth bottleneck
- Example: Numerical algorithm (ODE solver)
- Bandwidth bound $\rightarrow$ Flops bound
- More speed: Boost.SIMD

Data bandwidth and latency limitation

Modern CPUs: ~ 3 GHz, 1 Op/cycle $\rightarrow$ 3 GFlops/s

Data bandwidth and latency limitation

Modern CPUs: ~ 3 GHz, 1 Op/cycle $\rightarrow$ 3 GFlops/s

Problem: Data transfer from memory to the CPU $\sim$: 16 GB/s

Example $\mathbf{x} = \mathbf{a} + \mathbf{b}$: 24 Bytes/Operation

3 GFlops/s $\leftrightarrow$ 72 GByte/s   or   16 GByte/s $\leftrightarrow$ 0.7 GFlop/s

Data bandwidth and latency limitation

Modern CPUs: $\sim$ 3 GHz, 1 Op/cycle $\rightarrow$ 3 GFlops/s

Problem: Data transfer from memory to the CPU $\sim$: 16 GB/s

Example $\mathbf{x} = \mathbf{a} + \mathbf{b}$: 24 Bytes/Operation

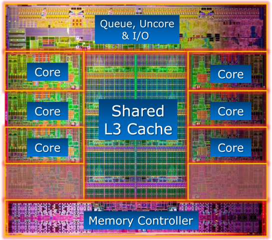
3 GFlops/s $\leftrightarrow$ 72 GByte/s or 16 GByte/s $\leftrightarrow$ 0.7 GFlop/s

Cache

Intel Sandy Bridge

Cache	Size	GB/s	Cycles
L1	32K	350	4
L2	256K	250	12
L3	3-6M	100	12-30
RAM	4-32G	16	20-100

Intel® Core™ i7-3960X Processor Die Detail



Cache effect: Ordinary Differential Equations

Solve a system of N small, independent ODEs: $\dot{r}_i = f(r_i, t)$

(Parameter study, Monte-Carlo)

Cache effect: Ordinary Differential Equations

Solve a system of N small, independent ODEs: $\dot{r}_i = f(r_i, t)$

(Parameter study, Monte-Carlo)

Two approaches:

- time first: $r_1(t) \to r_1(t + \Delta t) \to r_1(t + 2\Delta t) \ldots$, then r_2 ...
- vector first: $\mathbf{r}(t) \to \mathbf{r}(t + \Delta t) \to \mathbf{r}(t + 2\Delta t) \ldots$

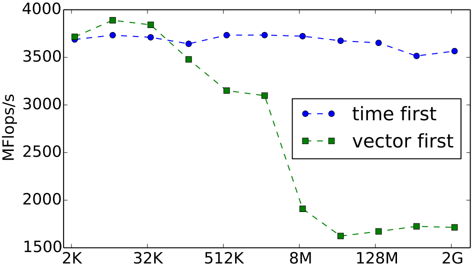
Cache effect: Ordinary Differential Equations

Solve a system of N small, independent ODEs: $\dot{r}_i = f(r_i, t)$

(Parameter study, Monte-Carlo)

Two approaches:

- time first: $r_1(t) \to r_1(t+\Delta t) \to r_1(t+2\Delta t)\ldots$, then r_2 ...
- vector first: $\mathbf{r}(t) \to \mathbf{r}(t+\Delta t) \to \mathbf{r}(t+2\Delta t)\ldots$



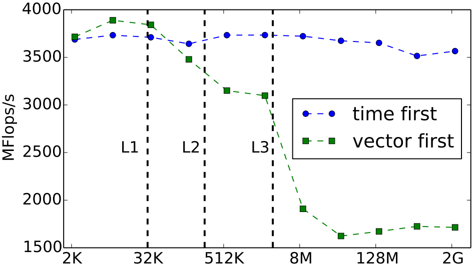
Cache effect: Ordinary Differential Equations

Solve a system of N small, independent ODEs: $\dot{r}_i = f(r_i, t)$

(Parameter study, Monte-Carlo)

Two approaches:

- time first: $r_1(t) \to r_1(t + \Delta t) \to r_1(t + 2\Delta t) \ldots$, then r_2 ...
- vector first: $\mathbf{r}(t) \to \mathbf{r}(t + \Delta t) \to \mathbf{r}(t + 2\Delta t) \ldots$



ODE with nearest neighbor coupling

$$\begin{aligned}\dot{r}_i &= f_i(\mathbf{r}, t) \\ &= \underbrace{h_i(r_i, t)}_{\text{local}} + \underbrace{g_i(r_i, r_{i-1}, r_{i+1}, t)}_{\text{n. n. coupling}}\end{aligned}$$

ODE with nearest neighbor coupling

$$\begin{aligned} \dot{r}_i &= f_i(\mathbf{r}, t) \\ &= \underbrace{h_i(r_i, t)}_{\text{local}} + \underbrace{g_i(r_i, r_{i-1}, r_{i+1}, t)}_{\text{n. n. coupling}} \end{aligned}$$

Time discretization $\mathbf{r}_t$ with Δt.

R-K algorithms: $\mathbf{r}_t \rightarrow \mathbf{r}_{t+\Delta t}$
with s stages $j = 1 \dots s$.

at each stage:

$\mathbf{k}^j = \mathbf{f}(\mathbf{r}', t')$
$\mathbf{r}' = \mathbf{r}_t + \sum_{n<j} a_{j,n} \mathbf{k}^n \Delta t$

ODE with nearest neighbor coupling

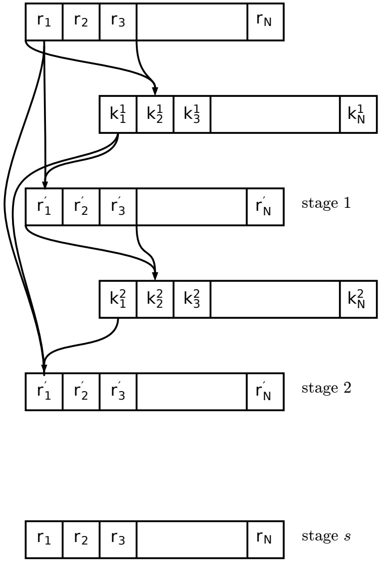
$$\dot{r}_i = f_i(\mathbf{r}, t)$$
$$= \underbrace{h_i(r_i, t)}_{\text{local}} + \underbrace{g_i(r_i, r_{i-1}, r_{i+1}, t)}_{\text{n. n. coupling}}$$

Time discretization $\mathbf{r}_t$ with Δt.

R-K algorithms: $\mathbf{r}_t \to \mathbf{r}_{t+\Delta t}$
with s stages $j = 1 \dots s$.

at each stage:

$\mathbf{k}^j = \mathbf{f}(\mathbf{r}', t')$
$\mathbf{r}' = \mathbf{r}_t + \sum_{n<j} a_{j,n} \mathbf{k}^n \Delta t$



ODE with nearest neighbor coupling

$$\dot{r}_i = f_i(\mathbf{r}, t)$$
$$= \underbrace{h_i(r_i, t)}_{\text{local}} + \underbrace{g_i(r_i, r_{i-1}, r_{i+1}, t)}_{\text{n. n. coupling}}$$

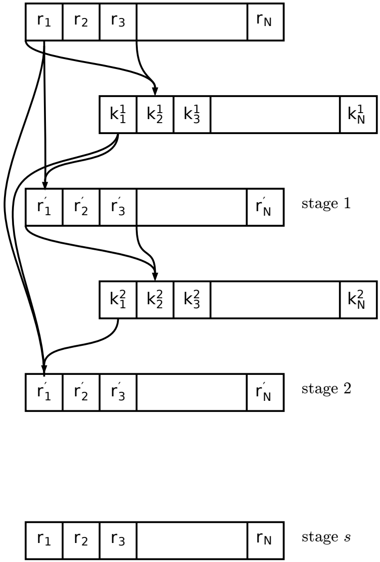
Time discretization $\mathbf{r}_t$ with Δt.

R-K algorithms: $\mathbf{r}_t \to \mathbf{r}_{t+\Delta t}$
with s stages $j = 1 \dots s$.

at each stage:

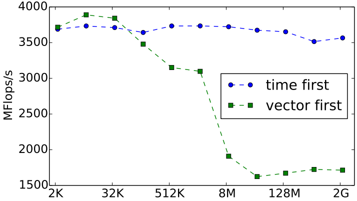
$\mathbf{k}^j = \mathbf{f}(\mathbf{r}', t')$
$\mathbf{r}' = \mathbf{r}_t + \sum_{n<j} a_{j,n} \mathbf{k}^n \Delta t$

Problem: Coupling prevents
"time first" approach



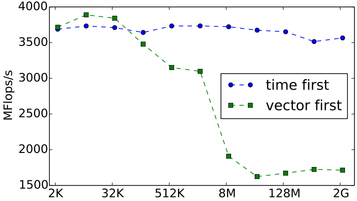
Improve Cache Usage: Granularity

Coupling: only "vector first"?



Improve Cache Usage: Granularity

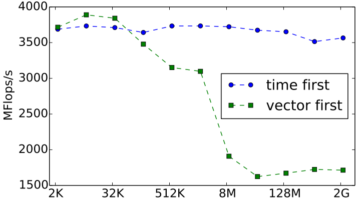
Coupling: only "vector first"?

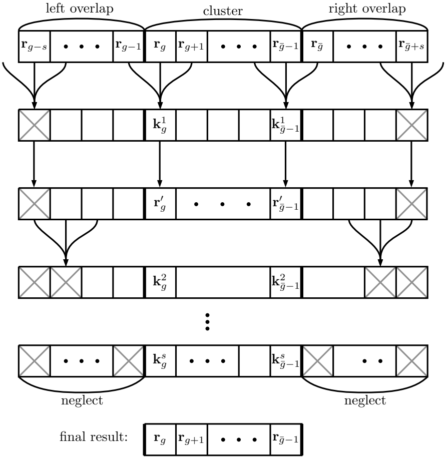


Clustering!

Improve Cache Usage: Granularity

Coupling: only "vector first"?

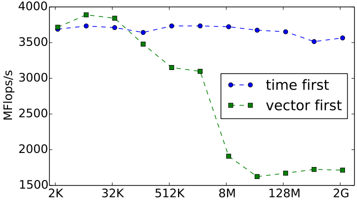




Clustering!

Improve Cache Usage: Granularity

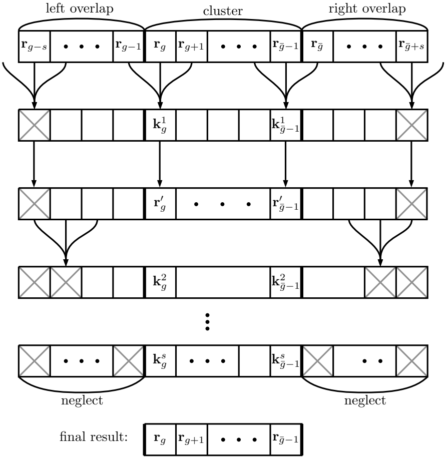
Coupling: only "vector first"?



Clustering!

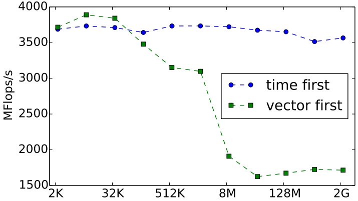
s iterations for each cluster at once → better cache usage

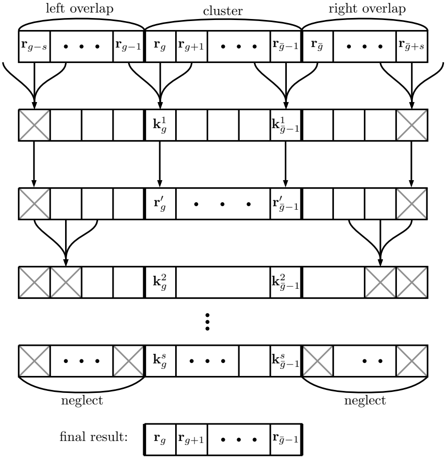
Price: additional overlap computations



Improve Cache Usage: Granularity

Coupling: only "vector first"?





Clustering!

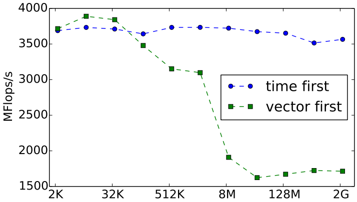
s iterations for each cluster at once $\rightarrow$ better cache usage

Price: additional overlap computations

Optimal granularity?

Improve Cache Usage: Granularity

Coupling: only "vector first"?

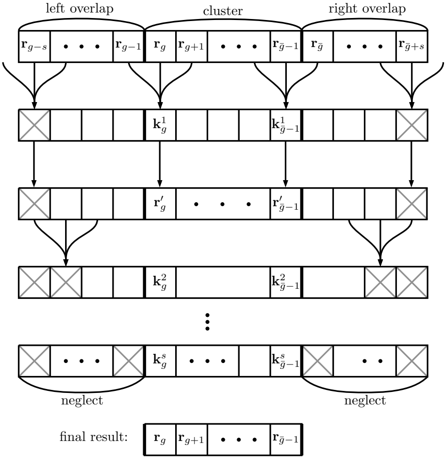


Clustering!

s iterations for each cluster at once $\rightarrow$ better cache usage

Price: additional overlap computations

Optimal granularity?
$\sim$ Cache size... Measure!



Performance Results

Coupled Rössler systems, $N = 2^{20} \approx 10^6$ (24 MB)

Intel Xeon E5-2690 @ 3.8GHz, Intel Compiler 15.0.0

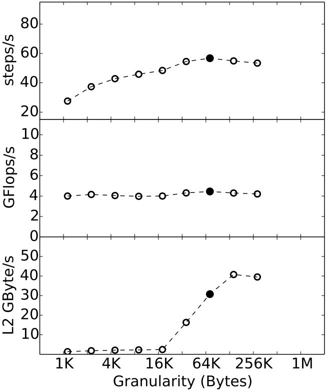
Performance measurements with `likwid` framework

Performance Results

Coupled Rössler systems, $N = 2^{20} \approx 10^6$ (24 MB)

Intel Xeon E5-2690 @ 3.8GHz, Intel Compiler 15.0.0

Performance measurements with `likwid` framework

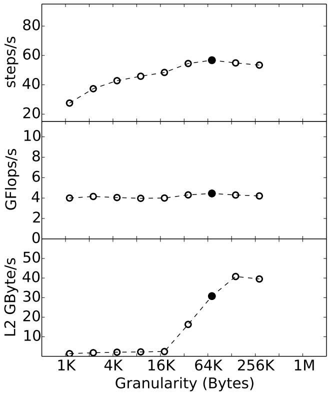


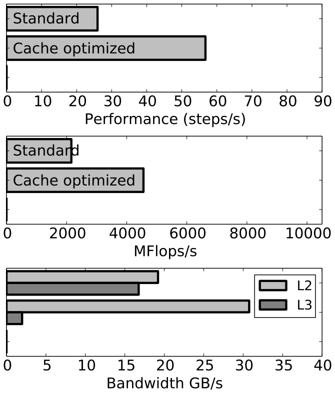
Performance Results

Coupled Rössler systems, $N = 2^{20} \approx 10^6$ (24 MB)

Intel Xeon E5-2690 @ 3.8GHz, Intel Compiler 15.0.0

Performance measurements with `likwid` framework



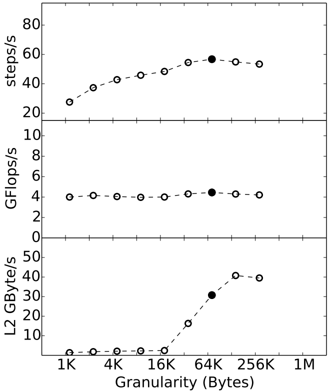


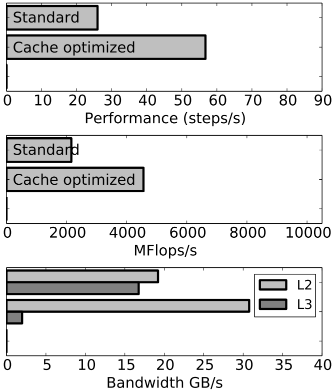
Performance Results

Coupled Rössler systems, $N = 2^{20} \approx 10^6$ (24 MB)

Intel Xeon E5-2690 @ 3.8GHz, Intel Compiler 15.0.0

Performance measurements with `likwid` framework

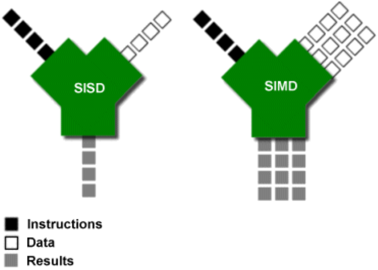




Bandwidth bound $\longrightarrow$ Flops/s bound

Increase Flops/s: SIMD instructions

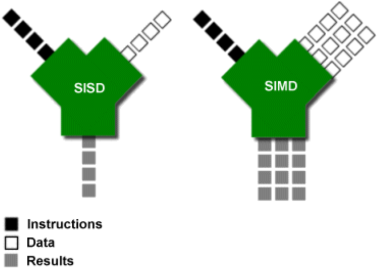
SIMD: Single Instruction Multiple Data



Additional registers and instructions in modern CPUs

Increase Flops/s: SIMD instructions

SIMD: Single Instruction Multiple Data



Additional registers and instructions in modern CPUs

Compilers try to use those automatically

Often, explicit SIMD code improve Flops/s significantly

Only helpful, if algorithm is Flops bound!

Boost.SIMD (not yet official Boost library)

- Abstraction of SIMD instructions:
  (SSE3, SSE4.1, SSE4.2, AVX, FMA4, AltiVec, Intel MIC).
- Fundamental ingredient: SIMD registers $\rightarrow$ SIMD pack.
- Expression template for optimization possibilities.
- Keep container–iterator–algorithm abstraction.

Boost.SIMD (not yet official Boost library)

- Abstraction of SIMD instructions:
  (SSE3, SSE4.1, SSE4.2, AVX, FMA4, AltiVec, Intel MIC).
- Fundamental ingredient: SIMD registers → SIMD pack.
- Expression template for optimization possibilities.
- Keep container–iterator–algorithm abstraction.

`simd::pack<T>`

- `pack<T,N>` SIMD register with `N` elements of `T`.
- `pack<T>` **automatically chooses** `N` **from available hardware.**
- `T` must be integral, e.g. `int`,`float`,`double`...
- `N` must be power of 2.

Boost.SIMD (not yet official Boost library)

- Abstraction of SIMD instructions:
  (SSE3, SSE4.1, SSE4.2, AVX, FMA4, AltiVec, Intel MIC).
- Fundamental ingredient: SIMD registers → SIMD pack.
- Expression template for optimization possibilities.
- Keep container–iterator–algorithm abstraction.

`simd::pack<T>`

- `pack<T,N>` SIMD register with `N` elements of `T`.
- `pack<T>` **automatically chooses `N` from available hardware.**
- `T` must be integral, e.g. `int`,`float`,`double`...
- `N` must be power of 2.

- Operations available `+`,`-`,`*`,`/`
- Math functions available: `pow`,`abs`,`sqrt`,`log10`,...

Quick Example: vector addition $\mathbf{x} = \alpha \cdot \mathbf{a} + \mathbf{b}$

```
typedef vector<double> vec;

double alpha;
vec x(N), a(N), b(N);

for(int n=0; n<x.size(); ++n)
  x[n] = alpha * a[n] + b[n];
```

Quick Example: vector addition $\mathbf{x} = \alpha \cdot \mathbf{a} + \mathbf{b}$

```
typedef vector<double> vec;

double alpha;
vec x(N), a(N), b(N);

for(int n=0; n<x.size(); ++n)
  x[n] = alpha * a[n] + b[n];
```

```
typedef simd::pack<double> pack;  // automatic size
typedef vector<pack, simd::allocator<pack> > vec;
static const size_t pack_size = pack::static_size;
static const int M = N/pack_size;

double alpha;
vec x(M), a(M), b(M);

for(int n=0; n<x.size(); ++n)
  x[n] = alpha * a[n] + b[n];
```

Algorithm does not change

```
for(int n=0; n<x.size(); ++n)
  x[n] = alpha * a[n] + b[n];
```

even more clear:

```
transform(a, b, x, [alpha](double a_n, double b_n)
    {return alpha * a_n + b_n;}
```

Algorithm does not change

```
for(int n=0; n<x.size(); ++n)
  x[n] = alpha * a[n] + b[n];
```

even more clear:

```
transform(a, b, x, [alpha](double a_n, double b_n)
    {return alpha * a_n + b_n;}
```

With Boost.SIMD: change typdefs, don't touch algorithm.

SIMD for ODE Simulation

ODE iteration $\mathbf{r}_t \to \mathbf{r}_{t+\Delta t}$: some sort of `transform`

Abstraction Boost.odeint provides:

- generic algorithms
- container independent implementation
- exchangeable backends

```
typedef vector<double> state_type;

state_type x(N);
odeint::runge_kutta4<state_type> rk4;
odeint::integrate_const(rk4, roessler, x, 0.0, T, dt);
```

SIMD for ODE Simulation

ODE iteration $\mathbf{r}_t \to \mathbf{r}_{t+\Delta t}$: some sort of `transform`

Abstraction Boost.odeint provides:

- generic algorithms
- container independent implementation
- exchangeable backends

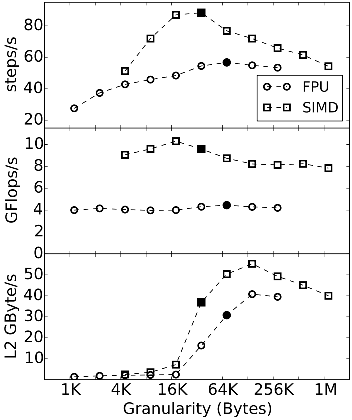
```
typedef simd::pack<double> simd_pack;
typedef vector<simd_pack,
               simd::allocator<simd_pack> > state_type;
static const size_t pack_size = simd_pack::static_size;
static const int M = N/pack_size;

state_type x(M);
odeint::runge_kutta4<state_type> rk4;
odeint::integrate_const(rk4, roessler, x, 0.0, T, dt);
```

SIMD Performance Results

Boost.odeint + Boost.SIMD

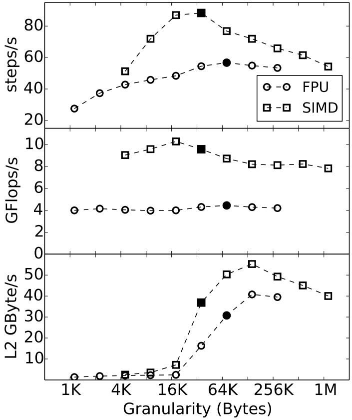
SIMD Performance Results

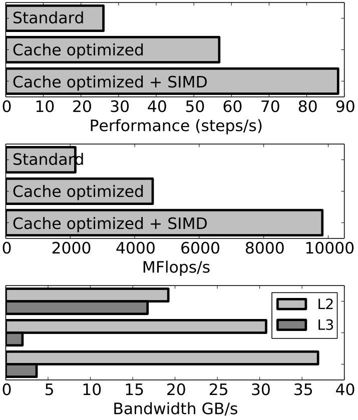
Boost.odeint + Boost.SIMD



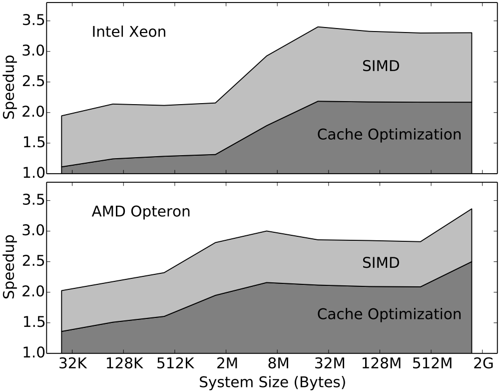
SIMD Performance Results

Boost.odeint + Boost.SIMD





Size Dependence



Summary + Conclusions

- Granularity $\rightarrow$ data transfer $\searrow$, BW bound $\rightarrow$ Flops/s bound.
- Increase Op/cycle via SIMD $\rightarrow$ total performance gain 3x.
- Known for stencil computations: space- and time-blocking.

Summary + Conclusions

- Granularity $\rightarrow$ data transfer $\searrow$, BW bound $\rightarrow$ Flops/s bound.
- Increase Op/cycle via SIMD $\rightarrow$ total performance gain 3x.
- Known for stencil computations: space- and time-blocking.

Take-home-message:

- Data size $>$ L2 cache size $\longrightarrow$ **introduce granularity**.
- Write generic algorithms, substitute Boost.SIMD.
- Source code: `https://github.com/mariomulansky/olsos`
- C++ with Boost.odeint and Boost.SIMD $\sim$ 200 lines of code.

Summary + Conclusions

- Granularity $\rightarrow$ data transfer $\searrow$, BW bound $\rightarrow$ Flops/s bound.
- Increase Op/cycle via SIMD $\rightarrow$ total performance gain 3x.
- Known for stencil computations: space- and time-blocking.

Take-home-message:

- Data size $>$ L2 cache size $\longrightarrow$ **introduce granularity**.
- Write generic algorithms, substitute Boost.SIMD.
- Source code: `https://github.com/mariomulansky/olsos`
- C++ with Boost.odeint and Boost.SIMD $\sim$ 200 lines of code.

odeint solving ODEs in C++

`www.odeint.com`