計算科学概論
— 高性能プログラミングと性能測定 (6/22 更新)

田浦健次朗

工学部 電子情報工学科
情報理工学系研究科 電子情報学専攻
情報基盤センター

6/24 の過ごし方 (1) — Jupyter Notebook での演習

- このスライドをダウンロードし, 以下の URL (Google のスプレッドシート) へアクセスし, 自分の演習用 URL とパスワードを使って演習ページへ飛んで下さい
- `https://tinyurl.com/y58d8bm6`
- ほとんどは説明を読んでその場で実行して結果を確認してもらうだけのものです

6/24 の過ごし方 (2) — Reedbush での演習

1. Reedbush へログイン

```
$ ssh -X t23xxx@reedbush-u1.cc.u-tokyo.ac.jp
```

2. Lustre フォルダへ移動

```
$ cdw
Changing current directory to /lustre/gt23/t23xxx
```

3. git レポジトリを clone (タイプ数・ミスを減らすために以下で実行)

```
reedbush-u1:t23xxx$ ../t23001/clone
```

4. おきることは以下です

```
+ git clone https://gitlab.eidos.ic.i.u-tokyo.ac.jp/tau/computational-science
          -ex.git
Cloning into 'computational-science-ex'...
remote: Enumerating objects: 493, done.
  ...
Checking out files: 100% (446/446), done.
```

5. `computational-science-ex` というフォルダで README.md を見つつ進めて下さい

目標

- 計算 & 計算機の「性能」について理解して
- 高性能プログラミングを実践 (次回 Reedbush で演習)
- 第 2 回 (4/08)「高性能計算機のアーキテクチャ」と, その他の計算科学の題材の間をつなぐ

Contents

ロードマップ

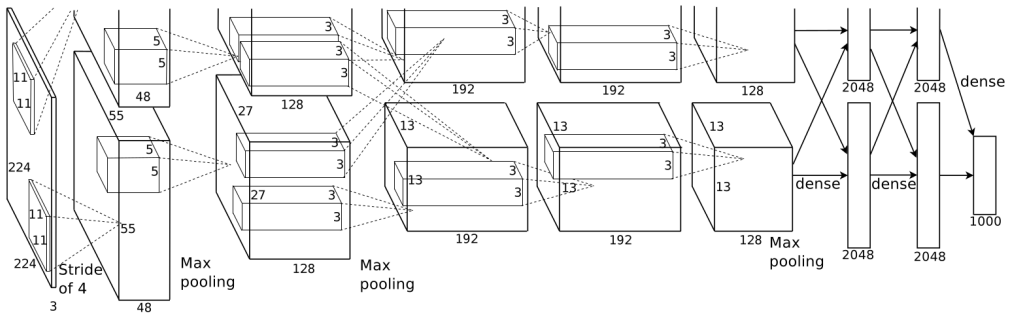
計算科学で代表的なワークロード

計算機の「性能」

高性能プログラミングの実践
- ワークロード
- SIMD 化
  - SIMD 命令
  - GCC のベクトル型拡張
  - ベクタ intrinsics 関数
  - 例題の SIMD 化
- マルチコア並列化
  - `parallel` pragma
  - Work sharing 構文
  - データ共有

計算科学で代表的なワークロード (1) — 密行列

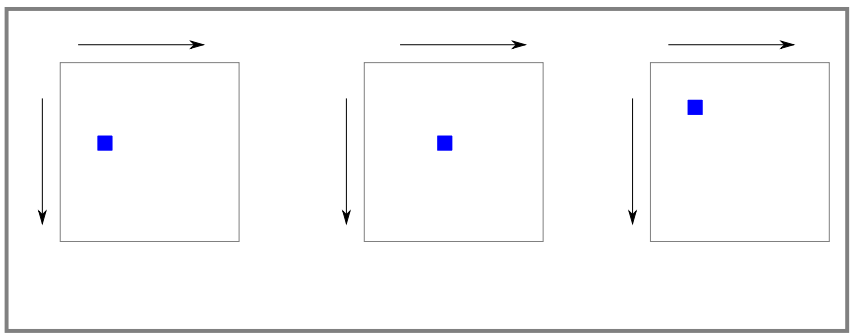
- 例: 境界要素法, 深層学習



Krizhevsky et al. “ImageNet Classification with Deep Convolutional Neural Networks”

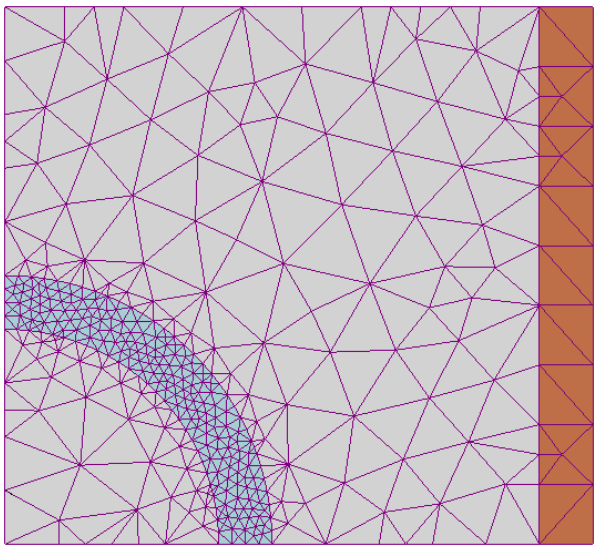
- 計算カーネル: 行列・行列積

```
1  for (i = 0; i < M; i++)
2    for (j = 0; j < N; j++)
3      for (k = 0; k < K; k++)
4        C(i,j) += A(i,k) * B(k,j);
```



計算科学で代表的なワークロード (2) — 疎行列

- 例: 不規則格子での差分法や有限要素法



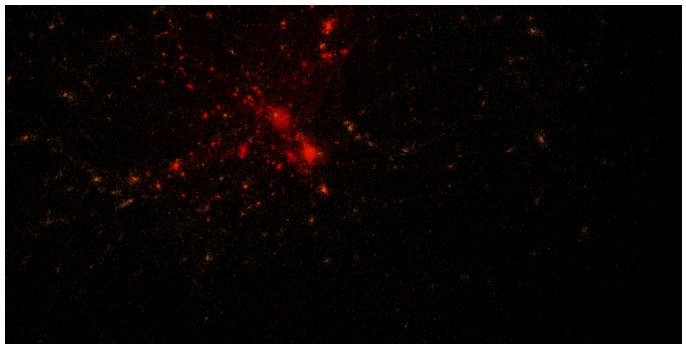
www.wikipedia.org

- 計算カーネル: 疎行列ベクトル積

```
1  for (i = 0; i < M; i++) {
2    y[i] = 0;
3    for (idx = start[i]; idx < start[i+1]; idx++) {
4      y[i] += A[idx] * x[J[idx]];
5    }
6  }
```

計算科学で代表的なワークロード (3) — N 体問題

- 例: 分子動力学, 天文, 流体, ...



www.wikipedia.org

- 計算カーネル: 直接計算, 多重極展開
- 重力の直接計算カーネル

```
1  for (i = 0; i < n; i++) {
2    for (j = 0; j < n; j++) {
3      if (i != j) {
4        dx = p[j].pos - p[i].pos;
5        r = |dx|;
6        p[i].acc += p[j].m * dx / (r * r * r);
7      }
8    }
9  }
```

計算科学で代表的なワークロード (4) — モンテカルロ法

- 例: 統計物理, 機械学習, あらゆる期待値計算, ...

ロードマップ

計算機の「性能」

- プロセッサ (CPU や GPU) の性能は普通「○○ FLOPS」と表現され, 1 秒に実行可能な最大の浮動小数点演算数を表す

計算機の「性能」

- プロセッサ (CPU や GPU) の性能は普通「○○ FLOPS」と表現され, 1 秒に実行可能な最大の浮動小数点演算数を表す
  - FLOPS = FLoating point Operations Per Second

計算機の「性能」

- プロセッサ (CPU や GPU) の性能は普通「○○ FLOPS」と表現され, 1 秒に実行可能な最大の浮動小数点演算数を表す
  - FLOPS = FLoating point Operations Per Second
  - 例えば 10 GFLOPS = 1×10^{10} FLOPS
  - K = 10^3, M = 10^6, G = 10^9, T = 10^{12}, P = 10^{15}, E = 10^{18}, ...

計算機の「性能」

- プロセッサ (CPU や GPU) の性能は普通「○○ FLOPS」と表現され, 1 秒に実行可能な最大の浮動小数点演算数を表す
  - FLOPS = FLoating point Operations Per Second
  - 例えば 10 GFLOPS = 1 $\times 10^{10}$ FLOPS
  - K = 10^3, M = 10^6, G = 10^9, T = 10^{12}, P = 10^{15}, E = 10^{18}, ...
- ちなみに flop (小文字) は floating point operation の意味で使われる事が多い
  - 100 flops = 100 回の浮動小数点演算
  - 100 FLOPS = 毎秒 100 回の浮動小数点演算

今時の CPU/GPU の「性能」

- 単位: TFLOPS

	倍精度	単精度	消費電力
NVIDIA Tesla P100	5.0	10.0	$\approx$ 300W
NVIDIA Tesla V100	7.5	15.0	$\approx$ 300W
Intel Knights Landing	3.0	6.0	$\approx$ 250W
Intel Broadwell E5-2695 v4	0.6	1.2	$\approx$ 120W
Intel Cascade Lake Platinum 8280	2.4	4.8	$\approx$ 205W

「性能」について知っておかねばならない事

- この CPU/GPU の「性能」が○○ FLOPS というのは, 割と多くのアプリケーションで自然にそのくらい (例: 6〜7 割) の性能が出る, ということを意味していない

「性能」について知っておかねばならない事

- この CPU/GPU の「性能」が○○ FLOPS というのは, 割と多くのアプリケーションで自然にそのくらい (例: 6〜7 割) の性能が出る, ということを意味していない
- 常に最大性能が出るわけじゃないのは当たり前, と思うでしょうが, そのギャップの大きさには驚き, 怒りすら覚えるかも知れません

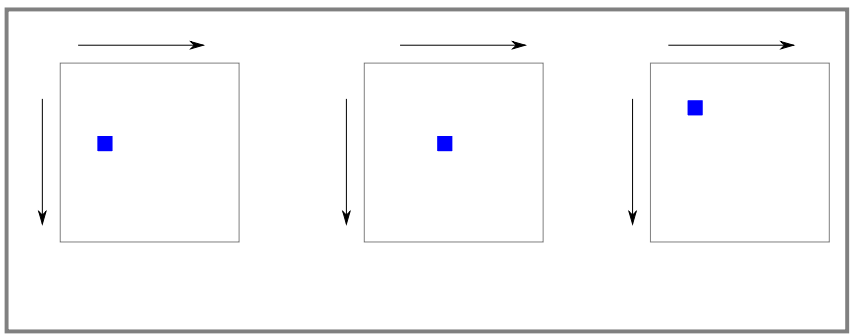
「性能」について知っておかねばならない事

- この CPU/GPU の「性能」が○○ FLOPS というのは, 割と多くのアプリケーションで自然にそのくらい (例: 6〜7 割) の性能が出る, ということを意味していない
- 常に最大性能が出るわけじゃないのは当たり前, と思うでしょうが, そのギャップの大きさには驚き, 怒りすら覚えるかも知れません
- 不都合な真実?

不都合な真実 — 密行列積の場合

- 以下の密行列積コード:

```
1 for (long i = 0; i < M; i++) {
2   for (long j = 0; j < N; j++) {
3     for (long k = 0; k < K; k++) {
4       C(i,j) += A(i,k) * B(k,j);
5     }
6   }
7 }
```



- を Reedbush の, 「性能」 1.2 TFLOPS のプロセッサ Intel Broadwell E5-2695 v4 (Reedbush の計算ノード) で実行したときの性能は?

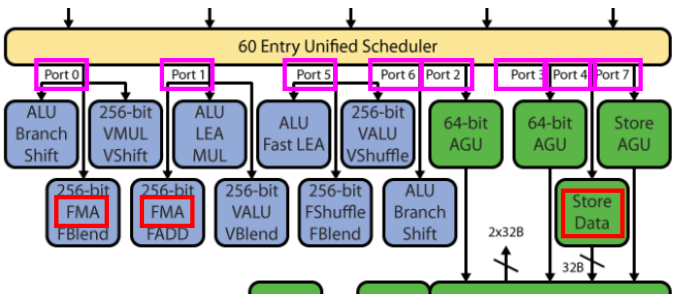
実際の結果

```
1  A = 1000 x 1000 (4000000 bytes)
2  B = 1000 x 1000 (4000000 bytes)
3  C = 1000 x 1000 (4000000 bytes)
4  2000000000 flops, total 12000000 bytes
5  2000000000 flops in 3942476919 clocks = 0.507295 flops/clock
6  frequency 2.099996 GHz
7  1.07 GFLOPS
8  OK: max relative error = 0.000000
```

- CPU の最大性能 1.2 TFLOPS と比べると約 1/1000 (!)
- ちなみに計算ノードには同 CPU が 2 個積まれているので, ノードの最大性能は 2.4 TFLOPS

最大「性能」の真実

- プロセッサの性能を決めているのは, 1 クロックに実行できる命令数



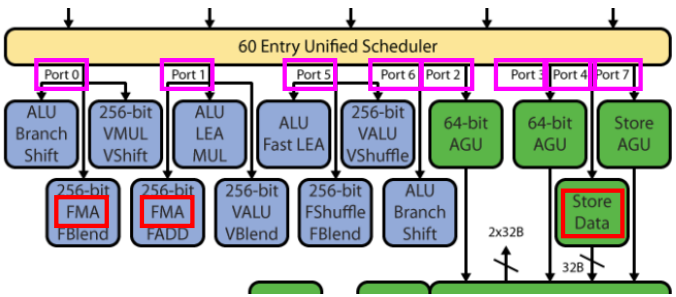
source:

http://www.realworldtech.com/haswell-cpu/

最大「性能」の真実

- プロセッサの性能を決めているのは, 1 クロックに実行できる命令数

- ざっくり言えば, 1 クロックに,
  - 浮動小数点命令が○個
  - ＋ 整数演算が○個
  - ＋ load 命令が○個
  - ＋ store 命令が○個,
  - . . . みたいな



source:

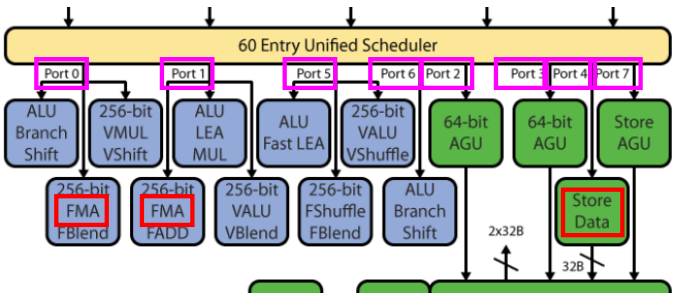
http://www.realworldtech.com/haswell-cpu/

最大「性能」の真実

- プロセッサの性能を決めているのは, 1 クロックに実行できる命令数

- ざっくり言えば, 1 クロックに,
  - 浮動小数点命令が○個
  - ＋ 整数演算が○個
  - ＋ load 命令が○個
  - ＋ store 命令が○個,
  - . . . みたいな
- 最大「性能」は以下の掛け算の結果に過ぎない:

  1 浮動小数点命令あたりの演算数
  × 1 クロックに実行できる浮動小数点命令数
  × クロック周波数 (秒あたりのクロック数)
  × コア数

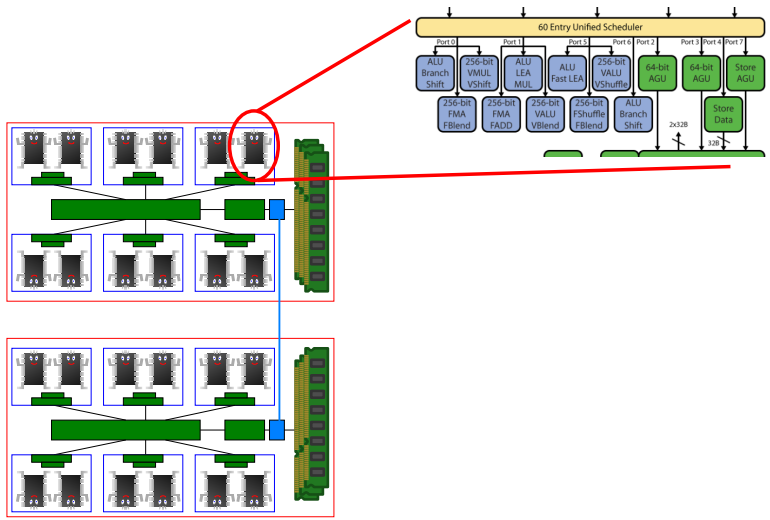


source:

http://www.realworldtech.com/haswell-cpu/

例: Intel Broadwell (E5-2695 v4 2.10 GHz) の場合

- 単精度 (32 bit) 浮動小数点数の場合:

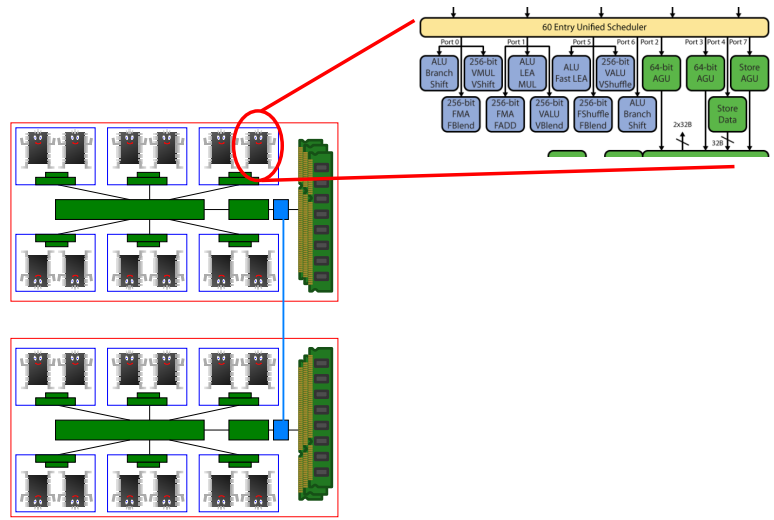


例: Intel Broadwell (E5-2695 v4 2.10 GHz) の場合

- 単精度 (32 bit) 浮動小数点数の場合:

1 浮動小数点命令あたりの演算数

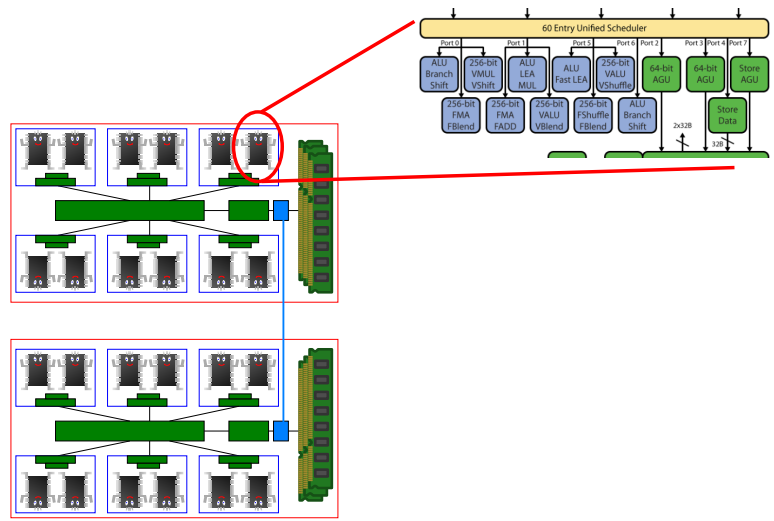
$=$ 16 flops



例: Intel Broadwell (E5-2695 v4 2.10 GHz) の場合

- 単精度 (32 bit) 浮動小数点数の場合:

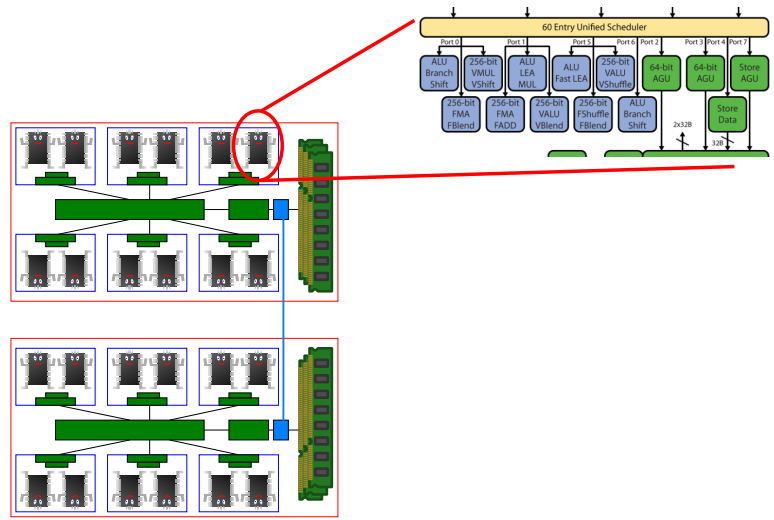
$$
\begin{aligned}
& 1\text{ 浮動小数点命令あたりの演算数} \\
\times\quad & 1\text{ クロックに実行できる浮動小数点命令数} \\
\\
= \quad & 16\text{ flops} \times 2
\end{aligned}
$$



例: Intel Broadwell (E5-2695 v4 2.10 GHz) の場合

- 単精度 (32 bit) 浮動小数点数の場合:

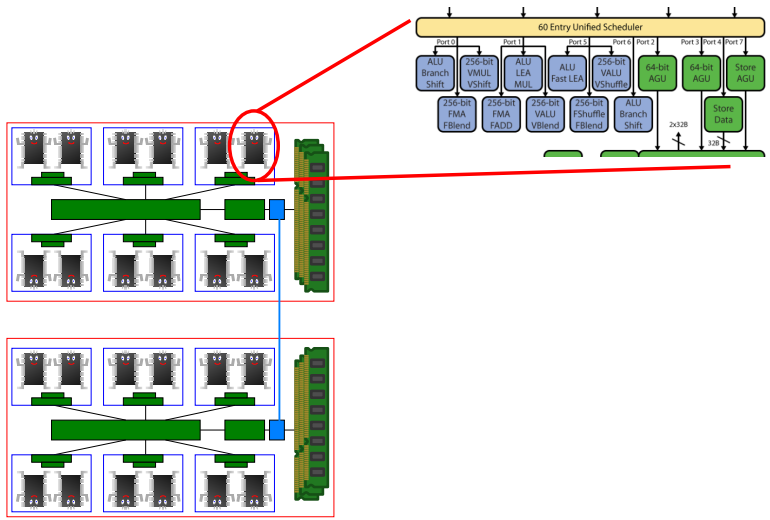
$$
\begin{aligned}
& 1\text{ 浮動小数点命令あたりの演算数} \\
\times\quad & 1\text{ クロックに実行できる浮動小数点命令数} \\
\times\quad & \text{クロック周波数 (秒あたりのクロック数)} \\
\\
=\quad & 16\text{ flops} \times 2 \times 2.1\text{ GHz (1/sec)}
\end{aligned}
$$



例: Intel Broadwell (E5-2695 v4 2.10 GHz) の場合

- 単精度 (32 bit) 浮動小数点数の場合:

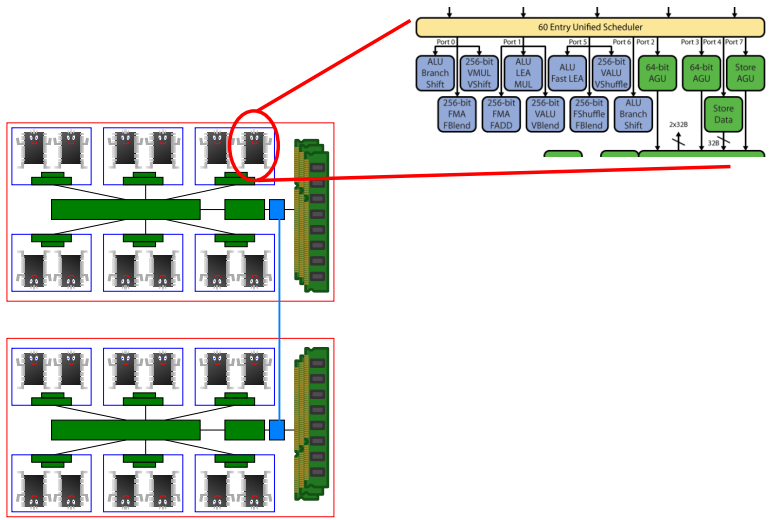
$$
\begin{aligned}
& 1\text{ 浮動小数点命令あたりの演算数} \\
\times\ & 1\text{ クロックに実行できる浮動小数点命令数} \\
\times\ & \text{クロック周波数 (秒あたりのクロック数)} \\
\times\ & \text{コア数} \\
=\ & 16\text{ flops} \times 2 \times 2.1\text{ GHz (1/sec)} \times 18
\end{aligned}
$$



例: Intel Broadwell (E5-2695 v4 2.10 GHz) の場合

- 単精度 (32 bit) 浮動小数点数の場合:

$$
\begin{aligned}
& 1 \text{ 浮動小数点命令あたりの演算数} \\
\times \quad & 1 \text{ クロックに実行できる浮動小数点命令数} \\
\times \quad & \text{クロック周波数 (秒あたりのクロック数)} \\
\times \quad & \text{コア数} \\
= \quad & 16 \text{ flops} \times 2 \times 2.1 \text{ GHz (1/sec)} \times 18 \\
= \quad & \text{1209.6 GFLOPS (1.2 TFLOPS)}
\end{aligned}
$$

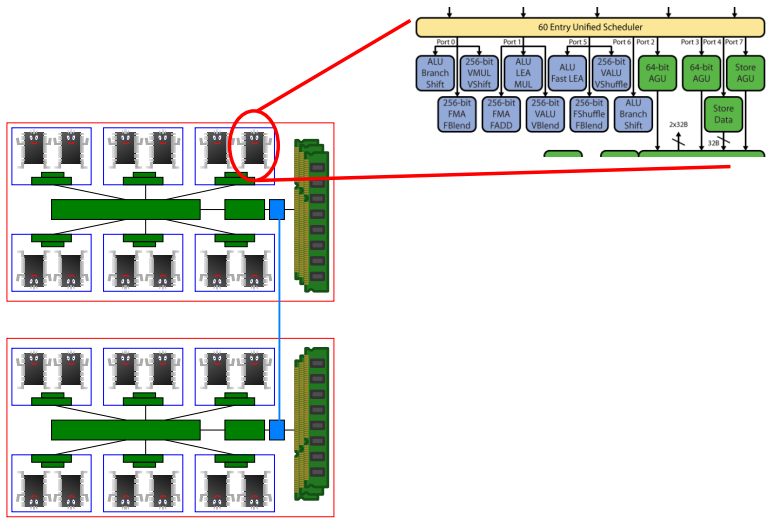


例: Intel Broadwell (E5-2695 v4 2.10 GHz) の場合

- 単精度 (32 bit) 浮動小数点数の場合:

$$
\begin{aligned}
& 1\text{ 浮動小数点命令あたりの演算数} \\
\times\ & 1\text{ クロックに実行できる浮動小数点命令数} \\
\times\ & \text{クロック周波数 (秒あたりのクロック数)} \\
\times\ & \text{コア数} \\
=\ & 16\text{ flops} \times 2 \times 2.1\text{ GHz (1/sec)} \times 18 \\
=\ & 1209.6\text{ GFLOPS (1.2 TFLOPS)}
\end{aligned}
$$

- 注: 倍精度 (64 bit) 浮動小数点数の場合: 1 クロックに実行できる浮動小数点命令数 = 8 となる以外は同じ

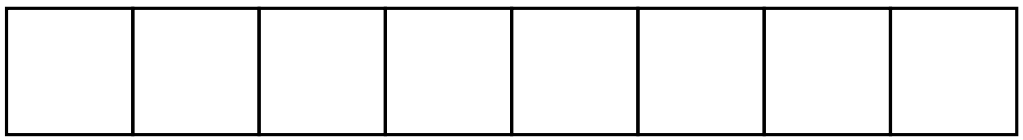


1 浮動小数点命令あたりの演算数について

- Intel Broadwell は 256 bit 幅のレジスタ (SIMD レジスタ) を持つ

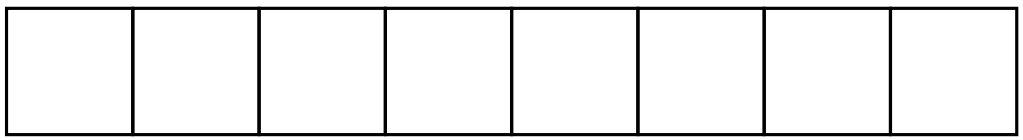
1 浮動小数点命令あたりの演算数について

- Intel Broadwell は 256 bit 幅のレジスタ (SIMD レジスタ) を持つ
- 256 bit (32 bytes) = 単精度 (32 bit / 4 bytes)) 浮動小数点数が 8 つ



1 浮動小数点命令あたりの演算数について

- Intel Broadwell は 256 bit 幅のレジスタ (SIMD レジスタ) を持つ
- 256 bit (32 bytes) = 単精度 (32 bit / 4 bytes)) 浮動小数点数が 8 つ

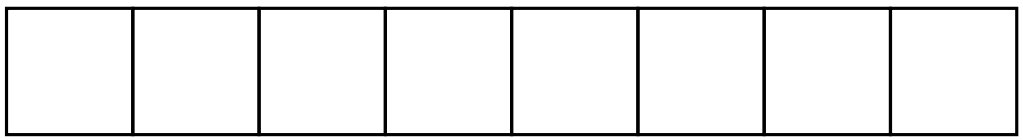


- 乗算と加算を同時に行う命令 (fmadd) があり, それを使うと 1 命令で $8 \times 2 = 16$ 演算ということになる

$$c = a * b + c$$

1 浮動小数点命令あたりの演算数について

- Intel Broadwell は 256 bit 幅のレジスタ (SIMD レジスタ) を持つ
- 256 bit (32 bytes) = 単精度 (32 bit / 4 bytes)) 浮動小数点数が 8 つ



- 乗算と加算を同時に行う命令 (fmadd) があり, それを使うと 1 命令で $8 \times 2 = 16$ 演算ということになる

$$c = a * b + c$$

- 同じ SIMD レジスタで倍精度 (64 bit / 8 bytes) 数 4 つを保持することもできる (したがって fmadd の演算数 =8)

知ってしまった真実

fmadd 命令を使わ (え) ない	→	1/2	(50.0%)
SIMD 命令を使わ (え) ない	→	1/8	(12.5%)
並列化して (でき) いない	→	1/18	(5.5%)
どちらも使ってない	→	1/144	(0.7%)

- ⇒ 最大性能に近い性能は, 相当限られた状況でのみ目にできる
- これらを意識せずにプログラムを書いたら言語処理系がそれらを勝手に使ってくれる, というのが理想だがなかなかそうはなっていない

ロードマップ

高性能「プログラミング」とは

1. ワークロードを見て,

高性能「プログラミング」とは

1. ワークロードを見て,
2. 「出せるはずの (近似) 性能」を理解して,

高性能「プログラミング」とは

1. ワークロードを見て,
2. 「出せるはずの (近似) 性能」を理解して,
3. 正しく道具 (SIMD, 並列) を取り出し,

高性能「プログラミング」とは

1. ワークロードを見て,
2. 「出せるはずの (近似) 性能」を理解して,
3. 正しく道具 (SIMD, 並列) を取り出し,
4. それに近い性能を出す

以降のロードマップ

- 実例に沿って各要素を説明
- ワークロード
  - 密行列積
  - N 体問題の直接法カーネル部分
- それぞれのツールを使うための「プログラミング」
  - SIMD 化 — ベクタ型, intrinsics
  - マルチコア並列化 — OpenMP parallel/for pragma
  - 複数ノード並列化 — MPI (時間切れの予定)
- 思想・哲学
  - 速くなったという結果より, 何が起きているかをしっかり理解するのが大事
  - つまり, 何が起きているかを調べる方法を覚えるのが大事
  - コンパイラが出したコードを見る, 測定する

ロードマップ

計算科学で代表的なワークロード

計算機の「性能」

高性能プログラミングの実践

- ワークロード
- SIMD 化
  - SIMD 命令
  - GCC のベクトル型拡張
  - ベクタ intrinsics 関数
  - 例題の SIMD 化
- マルチコア並列化
  - `parallel` pragma
  - Work sharing 構文
  - データ共有

想定ワークロード (1) — 密行列積

```
1 for (i = 0; i < M; i++) {
2  for (j = 0; j < N; j++) {
3   real c = 0.0;
4   for (k = 0; k < K; k++) {
5    c += A(i,k) * B(k,j);
6   }
7   C(i,j) += c;
8  }
9 }
```

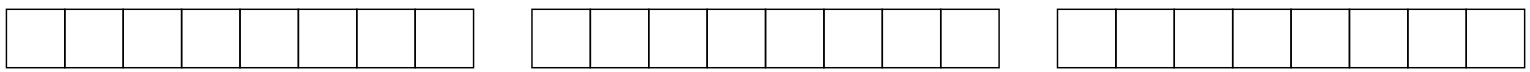
想定ワークロード (2) — N 体問題

```
1  real interact_all(long n, particle * p) {
2    for (i = 0; i < n; i++) {
3      for (j = 0; j < n; j++) {
4        if (i != j) {
5          dx = p[j].pos - p[i].pos; // (x,y,z)のベクトル
6          r = |dx|; // (x*x+y*y+z*z)^{1/2}
7          p[i].acc += p[j].m * dx / (r * r * r);
8        }
9      }
10   }
11 }
```

ロードマップ

SIMD 化基礎

- SIMD 命令 = Single Instruction Multiple Data 命令
- ひとつの命令で複数のデータに同時に (同じ) 演算



  - CPU にとっては, 命令デコードやディスパッチのオーバーヘッドを減らし, 低消費電力で最大性能を稼ぐ手段
- 注: SIMD 命令 (化) のことをしばしばベクトル命令 (化) と言う
- 昔のベクトル計算機におけるベクトル命令とは違うが, 考え方は似ているので, もはや区別しなくても平気 (?)

ロードマップ

Intel の SIMD 命令概要

- Intel は長きに渡り, 命令セットを拡張して SIMD 命令を増やしてきた (今も増えている)
- ひとつの SIMD 命令で扱える bit 幅も増え続けている
- 命令セット名 (括弧内は SIMD の bit 幅): MMX (64) → SSE3 (128) → SSE4 (128) → AVX (256) → AVX2 (256) → AVX-512 (512)
- Reedbush の CPU は Broadwell というマイクロアーキテクチャで AVX2 までをサポート
- 自分で書ける必要はないが「どんなコードが SIMD 化できるのか」を知る, 無事 SIMD 化できたかを判断できるようになる必要がある

いくつかの AVX 命令 (アセンブリ言語)

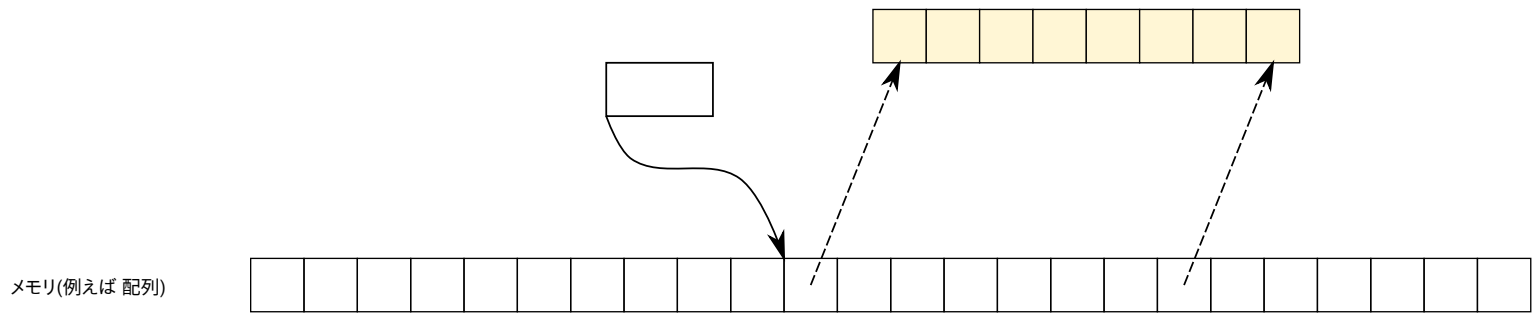
演算	表記	意味 (C 風表記で)
乗算	`vmulps %ymm0,%ymm1,%ymm2`	`ymm2 = ymm1 * ymm0`
加算	`vaddps %ymm0,%ymm1,%ymm2`	`ymm2 = ymm1 + ymm0`
乗加算	`vfmadd132ps %ymm0,%ymm1,%ymm2`	`ymm2 = ymm0*ymm2+ymm1`
ロード	`vmovaps 400(%rax),%ymm0`	`ymm0 = *(rax+400)`
ストア	`vmovaps %ymm0,400(%rax)`	`*(rax+400) = ymm0`

- `ymm0`, `ymm1` ... SIMD レジスタ (通常 ymm レジスタ)
  - 8 個の単精度浮動小数点数 (C の `float`), 4 個の倍精度浮動小数点数 (C の `double`) を保持できる
- `ymm0`, ..., `ymm15` まで 16 個ある
- `XXXXps` は *packed single precision* の略で, 単精度用の SIMD 命令であることを示す

ロード・ストア命令について

ロード	`vmovaps 400(%rax),%ymm0`	`ymm0 = *(rax+400)`
ストア	`vmovaps %ymm0,400(%rax)`	`*(rax+400) = ymm0`

- `%rax` は汎用レジスタで 64 bit の整数 (アドレス) を 1 つ保持できる
- `400(%rax)` は (`%rax` + 400) 番地を指定する記法
- `vmovaps` (`vmovups` もある) は, 指定したアドレスから連続した 256 bit (32 バイト) をアクセスする



- AVX2 以降は, バラバラのアドレスをアクセスする命令 (gather, scatter) もあるが, 省略

SIMD 命令と非 SIMD (スカラ) 命令

- 似て非なる命令たち
- SIMD 版 (`...p.`) vs. スカラ版 (`...s.`)
- 同じ SIMD でも 256 bit 版 (ymm) と 128 bit 版 (xmm)

	演算対象	SIMD /スカラ?	幅 (bits)	ISA
`vmulps %ymm0,%ymm1,%ymm2`	8 SPs	vector	256	AVX
`vmulpd %ymm0,%ymm1,%ymm2`	4 DPs	vector	256	AVX
`mulps %xmm0,%xmm1`	4 SPs	vector	128	SSE
`mulpd %xmm0,%xmm1`	2 DPs	vector	128	SSE
`mulss %xmm0,%xmm1`	1 SP	scalar	(32)	SSE
`mulsd %xmm0,%xmm1`	1 DP	scalar	(64)	SSE
`vfmadd132ss %ymm0,%ymm1,%ymm2`	1 SP	scalar	(32)	AVX

- ...ps : *p*acked *s*ingle precision
- ...pd : *p*acked *d*ouble precision
- xmm0, ..., xmm15 : 128 bit SSE registers (xmm$i$ は, 実は ymm$i$ の下半分)

SIMD の適用範囲と限界

- 命令を見ればわかるとおり, SIMD は, 複数のデータに対し, ほぼ同じ演算を適用する場合にしか適用できない
- また, それらのデータがメモリ上連続していないと (ロード・ストアに大きなオーバーヘッドがかかるため) 適用しにくい
- $\Rightarrow$ 主なターゲットは, 隣接した繰り返しが隣接した要素にアクセスする, 分岐 (if, switch, etc.) がほとんどない単純なループ

SIMD 化容易・困難なループ

易

```
1  for (i = 0; i < n; i++)
2    c[i] = a[i] + b[i];
```

難 (要素がバラバラ)

```
1  for (i = 0; i < n; i++)
2    c[i] = a[3 * i] + b[4 * i];
```

難 (分岐が多数)

```
1  for (i = 0; i < n; i++) {
2    if (...) {
3      ...
4    } else if (...) {
5      ...
6    } else if (...) {
7      ...
8    } else if (...) {
9      ...
10   }
11 }
```

SIMD を使う方法の選択肢

1. SIMD 化されたライブラリ関数の呼び出し (BLAS など)
2. コンパイラによるループの自動 SIMD(ベクトル) 化
3. SIMD 用言語拡張
   - OpenMP/Cilk Plus の SIMD 指示構文
   - Cilk Plus の配列構文
4. GCC/ICC ベクタ拡張
5. intrinsics 関数
6. アセンブリ

どのSIMD化手法を選択すべきか?

- コンパイラによるループの自動SIMD(ベクトル)化や, SIMD指示構文が十分強力ならばそれが理想だが, 実際には制限が多い
- それらが成功するようにするには結局, CPUの特性に合わせたプログラムやデータの書き換えが必要なことも多い
- この授業では目的に鑑みて, 低水準だが直截な方法を中心に扱う
  - GCC/ICC ベクタ拡張
  - intrinsics 関数
- それをマスターした後, 高水準な方法でのプログラミングをマスターするのは (多分) 易しい

ロードマップ

- 計算科学で代表的なワークロード
- 計算機の「性能」
- 高性能プログラミングの実践
  - ワークロード
  - SIMD 化
    - SIMD 命令
    - GCC のベクトル型拡張
    - ベクタ intrinsics 関数
    - 例題の SIMD 化
  - マルチコア並列化
    - `parallel` pragma
    - Work sharing 構文
    - データ共有

GCC のベクトル型

- GCC では以下のようにして (SIMD) ベクトル型を定義可能

```
1 typedef float floatv __attribute__((vector_size(32),aligned(sizeof(float))));
```

- `floatv` という新しい型が定義され, `float` を 8 つ (32 バイト分) 並べたもの, という意味になる
- 注: Intel C Compiler でも使えます
- 通常の四則演算がベクトル型に対しても可能 (そして, 当然 SIMD 命令が使われるだろうと確信できる)

```
1 floatv x, y, z;
2 z += x * y;
```

- 最近の GCC はスカラーとベクトルの混合も許す (ICC はダメ)

```
1 floatv x, y, z;
2 z = 3.5 * x + y;
```

実際に生成されたコードを見てみる

- コンパイラの-S オプション (アセンブリコード (.s) 生成) とお友達になる
- -S オプションで学習
  - 小さな関数を書いて生成されたコードを見る

```
1 float  plus(float x, float y) { return x + y; }
2 floatv plusv(floatv x, floatv y) { return x + y; }
```

  - 注目部分 (e.g., 最内ループ) に以下を埋め込みアセンブリ言語の中で検索 (asm volatile("...") は"..."をアセンブリに埋め込む構文).

```
1 asm volatile("# start my loop");
2 for (...) {
3     ...注目部分...
4 }
5 asm volatile("# end my loop");
```

- 以下などでコンパイル (g++, icc, icpc も同様)

```
1 gcc -S -O3 -mavx2 file.c
```

- -mavx2 は AVX2 を使うことを指示

スカラ型データからベクトル型データを作る方法あれこれ(1)

1. 変数宣言時の初期化子

```
floatv v = { 0,10,20,30,40,50,60,70 };
```

2. 8 要素を明示的に指定する関数

```
floatv v = _mm256_set_ps(0,10,20,30,40,50,60,70);/* これで v =
           0,10,20,30,40,50,60,70 */
```

   方法 1 と異なり初期化以外の場所でも使える.

3. スカラ一個から全要素同一のベクトルを作る

```
floatv v = _mm256_set1_ps(30);/* これで v = 30,30,30,30,30,...  */
```

スカラ型データからベクトル型データを作る方法あれこれ(2)

4. スカラ型の配列から連続 8 要素をロードする

```
1 float * a;
2       ...
3 floatv v = *((floatv *)(&a[i]));
4 /* これで v = {a[i],a[i+1], ...,a[i+7]} */
```

5. 同じことを関数で

```
1 float * a;
2       ...
3 floatv v = _mm256_load_ps(&a[i]);
4 floatv v = _mm256_loadu_ps(&a[i]);
5 /* これで v = {a[i],a[i+1], ...,a[i+7]} */
```

注意

- 方法 2 `_mm256_set_ps(0,10,20,30,40,50,60,70)` は, そのような命令があるわけではなく, 遅い場合が多い. 全てをメモリに一度ストアして, 方法 4 でロードしていたりする
- 方法 5 の `_mm256_load_ps(&a[i])` はアドレスが 32 の倍数で (32 バイト境界に align されてい) なくてはならない, という制限があり, それを破ると Segmentation Fault になる. 以降使わない.
- 方法 4 や方法 5 の `_mm256_loadu_ps` にはそのような制限はなく, 性能のペナルティもないのでこれを使うのを推奨
- 方法 4 で `_mm256_loadu_ps` が使われるのは, `floatv` を定義する時に `aligned(sizeof(float))` としたため. これを省略すると, `_mm256_load_ps` を使われてやっかいなことになる

ベクトル型データからスカラ型データを取り出す方法あれこれ

- 配列風の構文

```
floatv v;
float x = v[3];
```

- スカラ型配列の連続 8 要素にストアする

```
float * a;
floatv v;
     ...
*((floatv *)(&a[i])) = v;
/* これで a[i],a[i+1], ...,a[i+7] <- {v[0],v[1],...,v[7]} */
```

- 同じことを関数で

```
float * a;
     ...
_mm256_store_ps(&a[i], v);
_mm256_storeu_ps(&a[i], v);
/* これで a[i],a[i+1], ...,a[i+7] <- {v[0],v[1],...,v[7]} */
```

注意

- 方法 1 (`float x = v[3]`) は, そのような命令があるわけではなく, 遅い場合が多い. 全てをメモリに一度ストアしてからロードしていたりする
- 方法 3 の`_mm256_store_ps(&a[i], v)` では, アドレスが 32 の倍数で (32 バイト境界に align されてい) なくてはならない, という制限があり, それを破ると Segmentation Fault になる. 以降使わない.
- 方法 2 や, 方法 3 の`_mm256_storeu_ps` にはそのような制限はなく, 性能のペナルティもないのでこれを使うのを推奨
- 方法 2 で`_mm256_storeu_ps` が使われるのは, `floatv` を定義する時に `aligned(sizeof(float))` としたため. これを省略すると, `_mm256_store_ps` を使われてやっかいなことになる

ロードマップ

ベクタ intrinsics

- Intrinsics 関数: コンパイラによって特別処理される関数
- 「ベクタ」intrinsics は, SIMD 命令とほぼ 1 対 1 に対応
- ⇒ どんな命令を出してほしいかをかなり詳細に制御できる
- 使い方: まず以下の include 指示を書く

```
1 #include <x86intrin.h>
```

  すると以下が使えるようになる
  - いくつかのベクトル型 (floatv に相当するもの)
  - ベクトル型に対する多数の演算
- “Intel Intrinsics Guide” (`https://software.intel.com/sites/landingpage/IntrinsicsGuide/`) をブックマークしよう

ベクタ intrinsics

- ベクタ型:
  - `__m128` (128 bit 浮動小数点数),
  - `__m256` (256 bit 浮動小数点数),
  - ...
- 関数群
  - `_mm_xxx` (128 bit),
  - `_mm256_xxx` (256 bit),
  - ...
- ほぼすべての関数は特定の SIMD 命令に対応
  - `_mm_add_ps`, `_mm_mul_ps`, etc.
  - `_mm256_loadu_ps`
  - `_mm256_storeu_ps`
  - `_mm256_add_ps`, `_mm256_mul_ps`, `_mm256_fmadd_ps`, ...
- もっとも四則演算は intrinsics を使うまでもない

ロードマップ

SIMD 化の実践 (前提)

- Reedbush の CPU 用に AVX2 命令セット (256 bit = 32 byte)
- 単精度浮動小数点数 (32 bit = 4 byte)

```
1 typedef float real;
```

- つまりベクトル型 (`realv`) は `real` を 8 個保持

```
1 typedef real realv __attribute__((vector_size(32),aligned(sizeof(real))));
```

- ベクトル型の個々の要素をしばしば「レーン」と呼ぶ

```
1 enum { n_lanes = sizeof(realv) / sizeof(real) /* 8 */ };
```

密行列積の SIMD 化

- 元コード:

```
1  void mm(matrix& A, matrix& B, matrix& C) {
2    long M = A.n_rows, K = A.n_cols, N = B.n_cols;
3    for (long i = 0; i < M; i++) {
4      for (long j = 0; j < N; j++) {
5        real c = 0.0;
6        for (long k = 0; k < K; k++) {
7          c += A(i,k) * B(k,j);
8        }
9        C(i,j) += c;
10     }
11   }
12 }
```

- 注: C++の機能を使い `matrix` 型と, その要素アクセス演算 (`A(i,j)`) を定義している

matrix クラス

```
1  struct matrix {
2    long n_rows;                    // 行数
3    long n_cols;                    // 列数
4    long ld;                        // 行間の要素数 (通常 = n_cols)
5    real * a;                       // (n_rows x ld)個の要素を持つ配列
6    real& operator()(long i, long j) {
7      return a[i * ld + j];
8    }
9  };
```

SIMD 化

- 作戦: j ループのベクトル化
  `c += A(i,k) * B(k,j)`
  の $B(k,j), B(k,j+1), \ldots, B(k,j+7)$ に対する計算を SIMD 命令で実行
- 問い: なぜ i, k ではなく j なのか?

```
1 for (i = 0; i < M; i++) {
2  for (j = 0; j < N; j++) {
3   real c = 0.0;
4   for (k = 0; k < K; k++) {
5    c += A(i,k) * B(k,j);
6   }
7   C(i,j) += c;
8  }
9 }
```

$\Rightarrow$

```
1 for (i = 0; i < M; i++) {
2  for (j = 0; j < N; j+=8) {
3   real c = 0.0;
4   for (k = 0; k < K; k++) {
5    c += A(i,k) * B(k,j:j+7);
6   }
7   C(i,j:j+7) += c;
8  }
9 }
```

注: `B(k,j:j+7)` などという記法はないです

ベクトル要素のアクセス

- B(k,j), B(k,j+1), ..., B(k,j+7) をひとつのベクトル型データとして返す関数を定義するのがよい

```
1 realv loadv(long i, long j) {
2   return *(realv*)(&a[i*ld+j]); // _mm256_loadu_ps(&a[i*ld+j]);
3 }
```

- C(i,j), C(i,j+1), ..., C(i,j+7) にベクトル型データをストアする部分も同様

```
1 void storev(long i, long j, realv v) {
2   *(realv*)(&a[i*ld+j]) = v; // _mm256_storeu_ps(&a[i*ld+j], v);
3 }
```

```
1 for (i = 0; i < M; i++) {
2  for (j = 0; j < N; j+=n_lanes) {
3   realv c = {0,0,0,0,0,0,0,0}; // または _mm256_set1_ps(0);
4   for (k = 0; k < K; k++) {
5    c += A(i,k) * B.loadv(k,j);
6   }
7   C.storev(i,j,c);
8  }
9 }
```

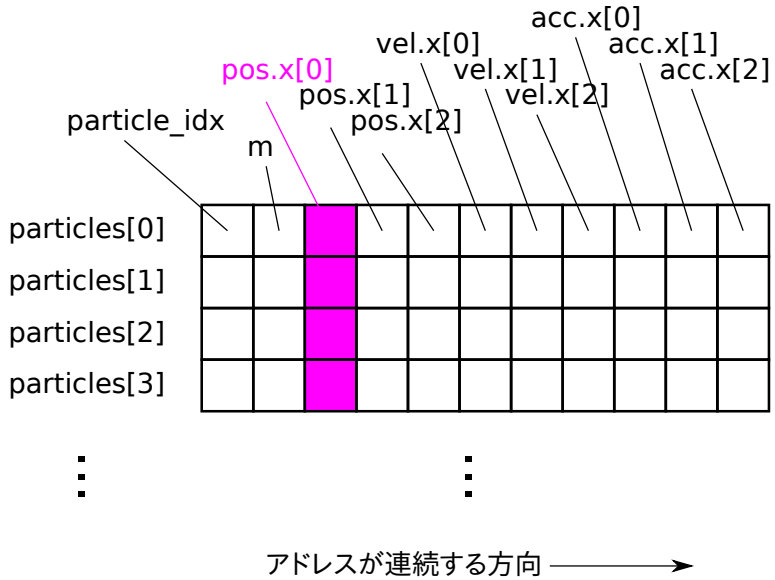
N 体問題の SIMD 化

- 作戦: 8 個の i 粒子 - 1 個の j 粒子の計算を SIMD 命令で
- 問題点 1: i 粒子の x (y, z) 座標 8 つを束ねたベクトルデータをどう作るか
  - ベクトル用のロード命令は, 連続した 8 つの要素しか取り出せない
- 問題点 2: 分岐命令 (`if (i != j)`) の処理
  - SIMD の 8 要素中の 1 要素でのみ `i == j` となる

```
1  real interact_all(long n, particle * p) {
2    for (i = 0; i < n; i++) {
3      for (j = 0; j < n; j++) {
4        if (i != j) {
5          dx = p[j].pos - p[i].pos; // (x,y,z)のベクトル
6          r = |dx|; // (x*x+y*y+z*z)^{1/2}
7          p[i].acc += p[j].m * dx / (r * r * r);
8        }
9      }
10   }
11 }
```

問題点 1: 粒子集合の (自然な) データ構造

```
1  struct vec {
2    real x[3]; // 3次元座標
3  };
4  typedef struct {
5    particle_idx idx; // 通し番号
6    real m;           // 質量
7    vec pos;          // 位置
8    vec vel;          // 速度
9    vec acc;          // 加速度
10 } particle;
```



- 連続する粒子の x (y, z) 座標がバラバラに (おそらく 44 バイトずつ離れて) 配置されている
- 構造体の配列 (Array of Structure; AoS)
- SIMD load 命令ひとつで取り出すのが困難
- 注: AVX2 の gather 命令ならば可能. だが性能は?

問題点 1 の解決策

- 策 1: 同じ要素だけをまとめた配列をたくさん作る
  - 配列の構造体 (Structure of Array; SoA)

```
particle_idx * idx; // 通し番号
real * m;           // 質量
real * pos[3];      // 位置 (x,y,z)
real * vel[3];      // 速度 (x,y,z)
real * acc[3];      // 加速度 (x,y,z)
```

- 策 2: 粒子 8 個 (SIMD レーン数) 分まとめた構造体を作る

```
struct vecv {
  realv x[3]; // 3次元座標
};
typedef struct {
  particle_idxv idx; // 通し番号
  realv m;           // 質量
  vecv pos;          // 位置
  vecv vel;          // 速度
  vecv acc;          // 加速度
} particlev;
```

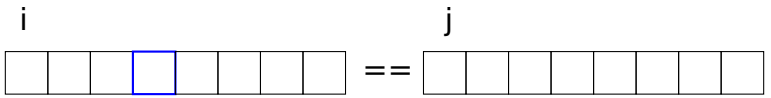
問題 2: 分岐の処理

元のコード

```
interact_all(long n, particle * p) {
 for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
   if (i != j) {
    dx = p[j].pos - p[i].pos;
    r = |dx|;
    p[i].acc += p[j].m * dx / (r*r*r);
} } } }
```

(問題 1 をなんとか解決して) SIMD 化された (擬似) コード:

```
interact_all(long n, particle * p) {
 // i を 8 つまとめて処理
 for (i = 0; i < n; i += 8) {
  for (j = 0; j < n; j++) {
   if (i:i+7 != j) {
    dx = p[j].pos - p[i:i+7].pos;
    r = |dx|;
    p[i:i+7].acc += p[j].m * dx / (r*r*r);
} } } }
```



- → SIMD 処理される一部のレーンでのみ実行される仕組みが必要

問題 2: SIMD と分岐

- 分岐:

```
1 if (E) {
2   A;
3 }
```

- ある程度汎用的な方法: predicate 付き命令
- よくやる方法: E が成り立たない lane で A を実行しても何も起きないような処理 (マスク) を考える (詳しくは演習で)

ロードマップ

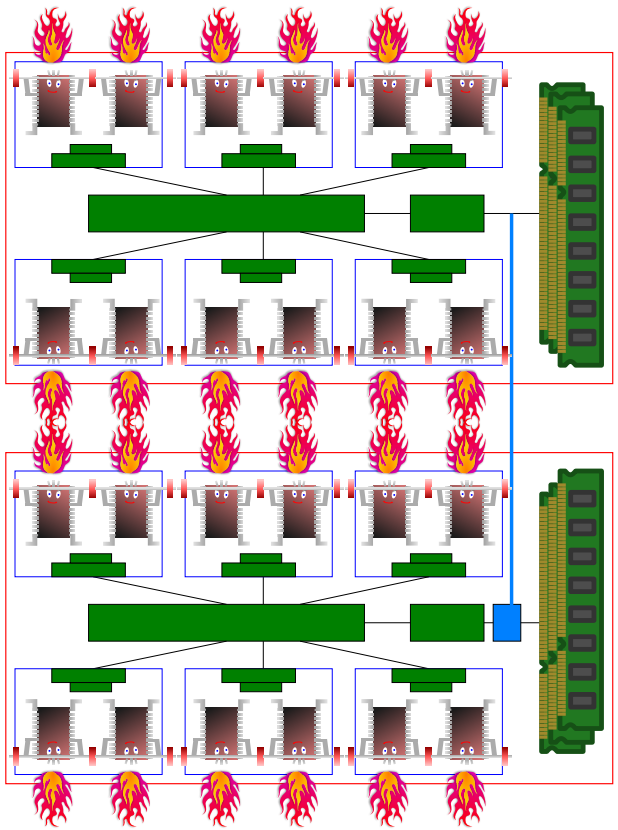
マルチコア並列化の選択肢

- OS が提供するスレッド (Pthread) を直接使う
- OpenMP
- Intel Thread Building Blocks, Cilk Plus

今日は OpenMP の, それもごく基本的な機能だけを使う

OpenMP

- ノード内 (共有メモリマルチコア) 上での並列プログラミング言語の, 事実上の標準
- C/C++/Fortran + 並列指示構文 (directive) + APIs
  - C/C++では`#pragma`,
  - Fortran ではコメントで
- GCC, ICC はじめ多くのコンパイラでサポート



OpenMP 参考文献

- 公式 HP `http://openmp.org/`
- 仕様 `http://openmp.org/wp/openmp-specifications/`
- 最新の仕様 4.5
  (`http://www.openmp.org/mp-documents/openmp-4.5.pdf`)
- 以下で節番号は仕様書 4.0 に基づく
  (`http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`)

OpenMP プログラムのコンパイルと実行

- コンパイル時に-fopenmp オプション

```
1 $ gcc -Wall -fopenmp program.c
```

- 実行時に, OMP_NUM_THREADS 環境変数で, 使うスレッド数 (コア数と思ってよい) を指定

```
1 $ OMP_NUM_THREADS=1 ./a.out  # use 1 thread
2 $ OMP_NUM_THREADS=4 ./a.out  # use 4 threads
```

- 指定しなければノードのコア数. ただし, Reedbush のジョブスクリプトでは, 以下の節の x 部分でそれを指定できる

```
1 #PBS -l select=1:ncpus=1:mpiprocs=1:ompthreads=x
```

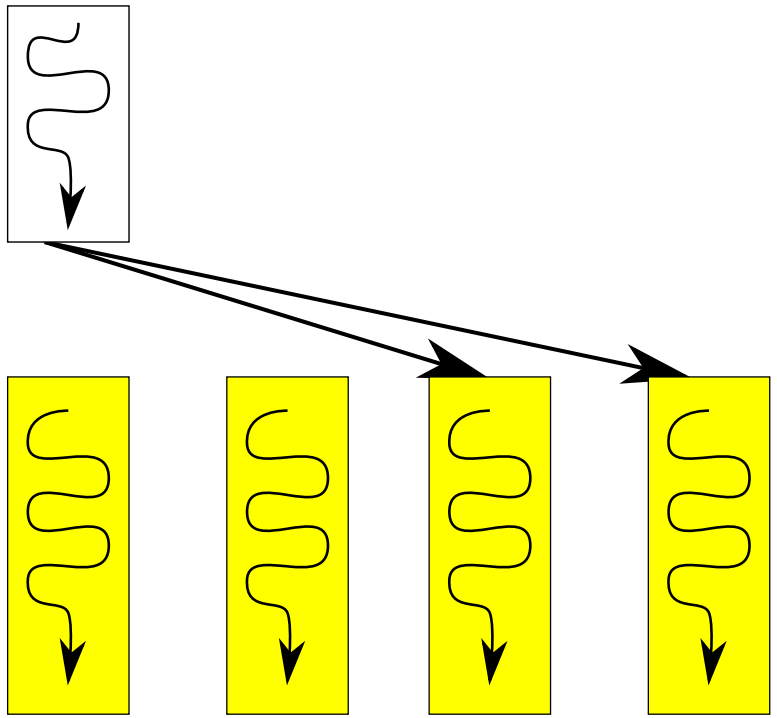
ロードマップ

- 計算科学で代表的なワークロード
- 計算機の「性能」
- 高性能プログラミングの実践
  - ワークロード
  - SIMD 化
    - SIMD 命令
    - GCC のベクトル型拡張
    - ベクタ intrinsics 関数
    - 例題の SIMD 化
  - マルチコア並列化
    - `parallel` pragma
    - Work sharing 構文
    - データ共有

基本中の基本の二つ

- `#pragma omp parallel` によってスレッドを生成 (2.5)
- その後 `#pragma omp for` で for 文の繰り返しをスレッドで分担 (2.7.1)

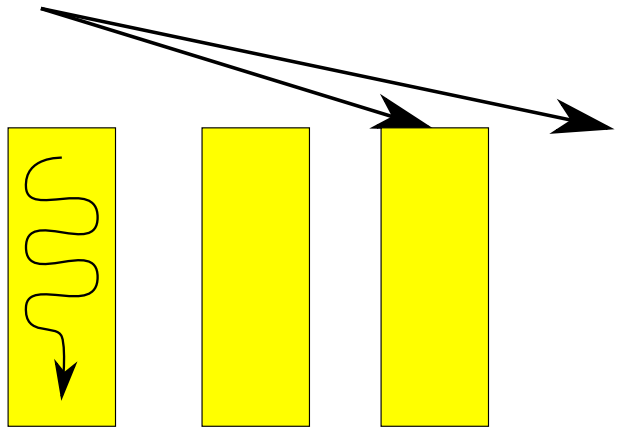
注: すべての OpenMP プラグマは以下の形`#pragma omp ...`



#pragma parallel

- 文法:

```
1   ...
2 #pragma omp parallel
3   S
4   ...
```



- 意味 (動作):
  - OMP_NUM_THREADS 個のスレッドのチームができる
  - 最初からいたスレッドがチームの *master* になる
  - チームの各スレッドが S を実行
  - マスターは全員が S を終了するのを待ち, 終了したら続きを (自分だけで) 実行

簡単なparallel pragmaの例

```
1  #include <stdio.h>
2  int main() {
3    printf("hello\n");
4  #pragma omp parallel
5    printf("world\n");
6    printf("good bye\n");
7    return 0;
8  }
```

```
1  $ OMP_NUM_THREADS=1 ./a.out
2  hello
3  world
4  $ OMP_NUM_THREADS=4 ./a.out
5  hello
6  world
7  world
8  world
9  world
10 good bye
```

細かい注

- 注: 複数の文を { ... } で囲めば一つの文になり, 結果として複数の文を parallel で実行可能

```
1  #include <stdio.h>
2  int main() {
3    printf("hello\n");
4  #pragma omp parallel
5    { printf("world\n");
6      printf("good bye\n");  }
7    return 0;
8  }
```

```
1  $ OMP_NUM_THREADS=4 ./a.out
2  hello
3  world
4  world
5  good bye
6  world
7  good bye
8  world
9  good bye
10 good bye
```

parallel の実際の動作

- ここでのスレッド ≈ OS がサポートするスレッド (e.g., Pthread) と思っていてよい
- 以下を書いて

```
1 int main() {
2 #pragma omp parallel
3   worker();
4 }
```

以下のように実行すれば,

```
1 $ OMP_NUM_THREADS=50 ./a.out
```

50 の OS レベルのスレッドができ, それぞれが関数 worker() を実行する

- ⇒ 実践的には, 使いたいコア数だけのスレッドを作ると思っておけば良い

スレッド間で仕事を「分割」するには?

- `#pragma omp parallel` はスレッドを作り, 全員が同じ文を実行する
- つまりそれ自体がいわゆる「並列化 (仕事を分け合って高速化)」の手段ではない
- スレッド間で仕事を分け合う手段が必要
  1. 自前でやる方法
  2. *work sharing* 構文を使う方法

自分でやるためのプリミティブ

- omp_get_num_threads() (3.2.2) : チーム内のスレッド数
- omp_get_thread_num() (3.2.4) : チーム内の自分の ID (0, 1, . . .)
- これさえあれば原理的には自分で好きなように仕事を分割できる
- e.g.,

```
1 #pragma omp parallel
2 {
3   int t  = omp_get_thread_num();
4   int nt = omp_get_num_threads();
5   /* n 回の繰り返しを nt スレッドで均等分割 */
6   for (i = t * n / nt; i < (t + 1) * n / nt; i++) {
7     ...
8   }
9 }
```

ロードマップ

Work sharing 構文

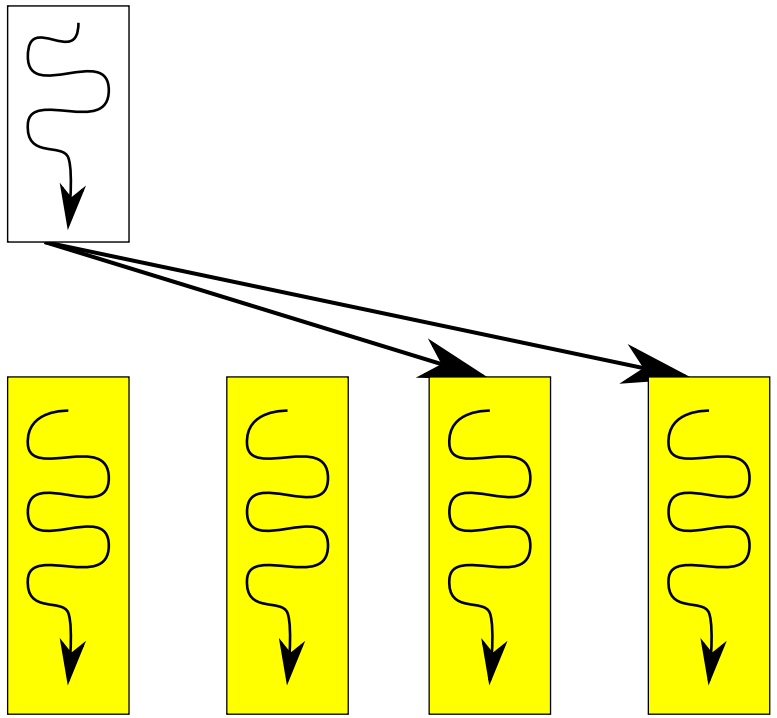
- 理論的には `parallel`, `omp_get_num_threads()`, `omp_get_thread_num()` だけで生きていくことが可能
- だが不便すぎる
- OpenMP は仕事をスレッド間で分割する手段 *(work sharing* 構文*)* も提供している
  - for
  - task (省略)
  - section (省略)

#pragma omp for (work-sharing for)

- 構文

```
1 #pragma omp for
2 for(i=...; i...; i+=...){
3   S
4 }
```

- 意味 (動作)
  チーム内のスレッドがループの
  繰り返しを「分け合って」実行
- 具体的にはどう「分け合う」？
  ⇒ scheduling



#pragma omp for の制限

- for 文ならなんでも並列実行できるわけではない
- 文法上の厳しい制約がある．繰り返しの回数が *for* 文開始時に簡単にわかることを保証するため
- 簡単に言えば以下のような形のものだけ

```
1  #pragma omp for
2  for(i = init; i < limit; i += incr)
3    S
```

(< や += は他の同種の演算も可<=や-=)

- *init*, *limit*, *incr* はループ中一定

Scheduling (2.7.1)

- `schedule` という節でループの繰り返しがどうスレッド間で分割されるかを指定できる

```
1 #pragma for schedule(...)
2 for (i = 0; i < n; i++) {
3   ...
4 }
```

- 3 つの選択肢 (`static, dynamic, guided`)

static, dynamic, guided

- `schedule(static[,`*chunk*`])`: *chunk* 回ずつ巡ぐり (round-robin)
- `schedule(dynamic[,`*chunk*`])`: 各スレッドが *chunk* 回分取っては終わったら次の *chunk* 回を取る, ... を繰り返す
- `schedule(guided[,`*chunk*`])`: dynamic と似ているが, 前半はいっぱい取り, 後半は少く取る
- *chunk* は一回に取る回数の最小値を与える

他の選択肢

- `schedule(runtime)` とすると実行時に環境変数 `OMP_SCHEDULE` で指定可能になる

```
1 $ OMP_SCHEDULE=dynamic,2 ./a.out
```

- `schedule(auto)` または何も指定しないと実装依存のデフォルトになる
- 多分 static で, chunk が ≈ 繰り返し数/スレッド数であるようなものが使われていると思う (要確認)

OpenMP プログラミングの頻出パターン

- 並列化できる・したいループを見つける

```
1 for (i = 0; i < HUGE; i++) {
2     ...
3 }
```

- そこに `parallel` と `for` を挿入

```
1 #pragma omp parallel
2 #pragma omp for
3 for (i = 0; i < HUGE; i++) {
4     ...
5 }
```

- 実際その二つを一緒にした構文もある

```
1 #pragma omp parallel for
2 for (i = 0; i < HUGE; i++) {
3     ...
4 }
```

ロードマップ

OpenMP は「共有メモリ」モデル

- `# pragma omp parallel` で作られたスレッドは, メモリ (アドレス空間) を共有する

OpenMP は「共有メモリ」モデル

- `# pragma omp parallel` で作られたスレッドは, メモリ (アドレス空間) を共有する
- メモリ? アドレス空間? 共有? どういう意味?

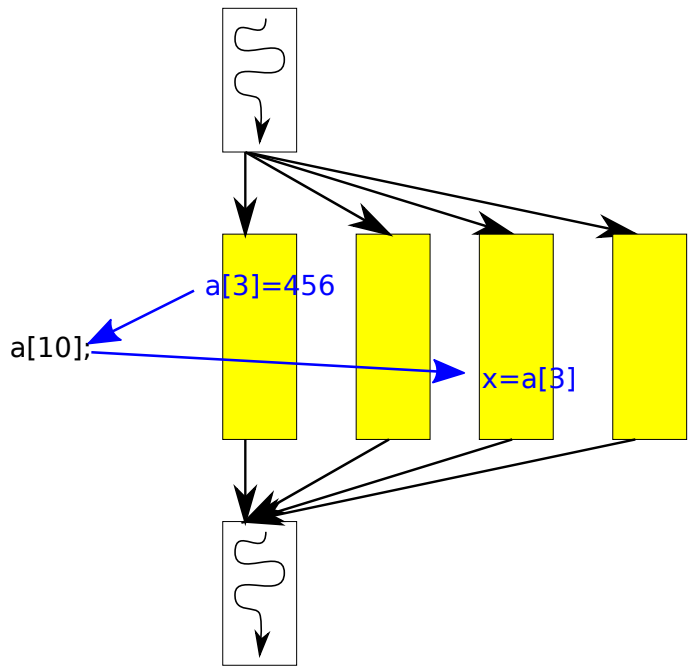
OpenMP は「共有メモリ」モデル

- `# pragma omp parallel` で作られたスレッドは, メモリ (アドレス空間) を共有する
- メモリ? アドレス空間? 共有? どういう意味?
  - ≈ プログラム言語の言葉で言えば, たいがいの「変数や配列を共有する」という意味です

OpenMP は「共有メモリ」モデル

- `# pragma omp parallel` で作られたスレッドは, メモリ (アドレス空間) を共有する

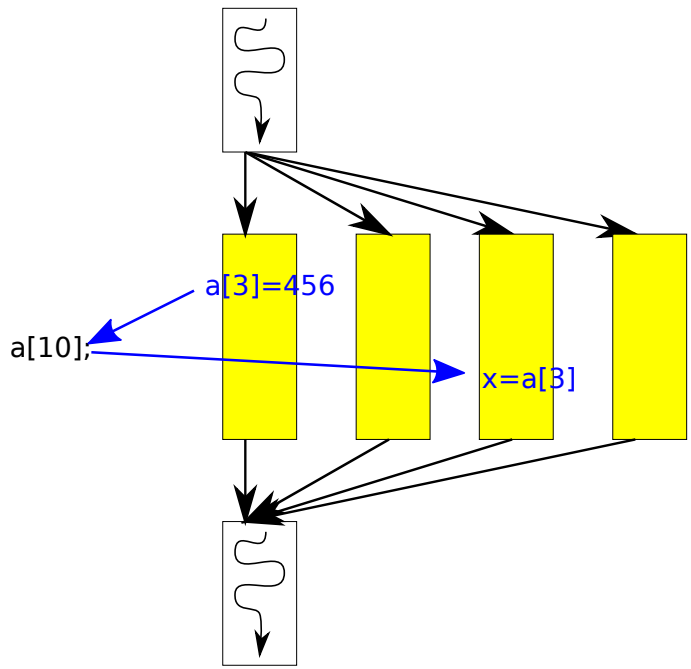
- メモリ? アドレス空間? 共有? どういう意味?
  - ≈ プログラム言語の言葉で言えば, たいがいの「変数や配列を共有する」という意味です
  - ⇒ つまり, あるスレッドが変数や配列の値を更新して, 他のスレッドが読んだら, それはちゃんと伝わる, という意味



OpenMP は「共有メモリ」モデル

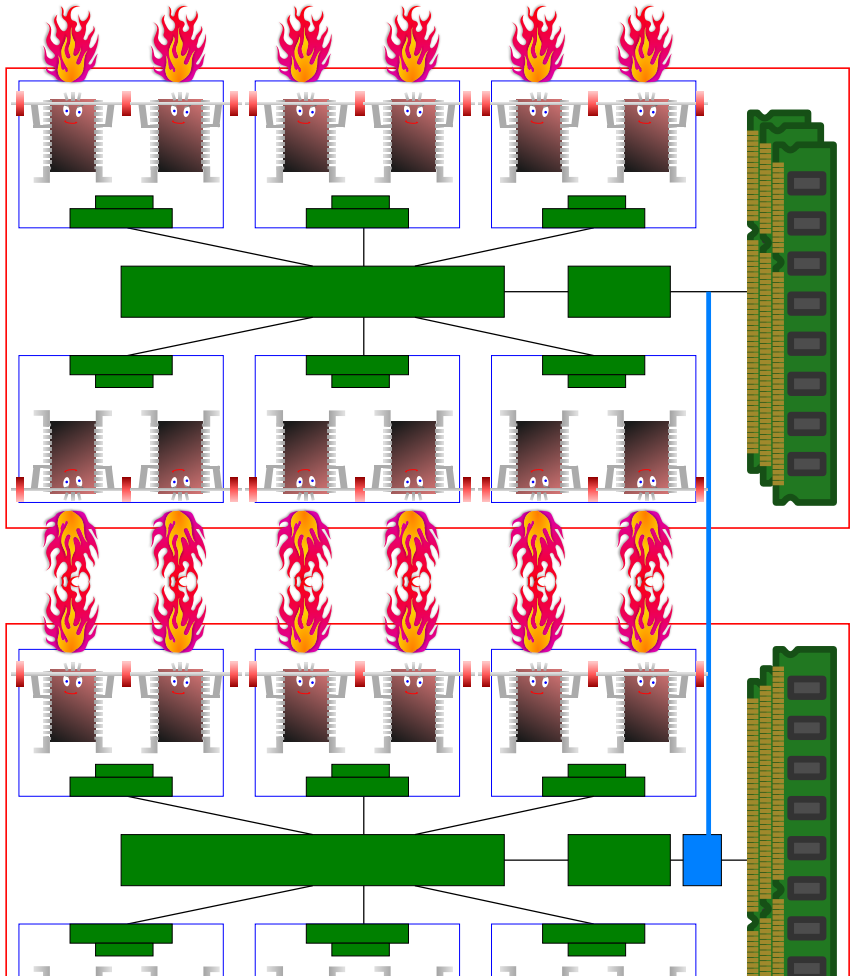
- `# pragma omp parallel` で作られたスレッドは, メモリ (アドレス空間) を共有する

- メモリ? アドレス空間? 共有? どういう意味?
  - ≈ プログラム言語の言葉で言えば, たいがいの「変数や配列を共有する」という意味です
  - ⇒ つまり, あるスレッドが変数や配列の値を更新して, 他のスレッドが読んだら, それはちゃんと伝わる, という意味
  - 参考: MPI はそうではない



注: 共有メモリが実現されているレイヤ

- マルチコア CPU 自身が共有メモリの機能を (ソフトウェアの介在なく) 持っている
- あるコアがアドレス a に 456 を store 命令で書き (`movq $456,(`a`)`), 別のコアが (後で) 同じアドレス a を load 命令で読めば (`movq (`a`),%rbx`), 456 が出てくる, というのは CPU の機能



OpenMP でのデータ共有に関して知っておくべき点

- 共有されている場合の注意点 (競合状態)
- 共有されるもの (shared) とされないもの (private) の区別
- 競合状態の調停法

共有されたデータで計算をする場合の注意点

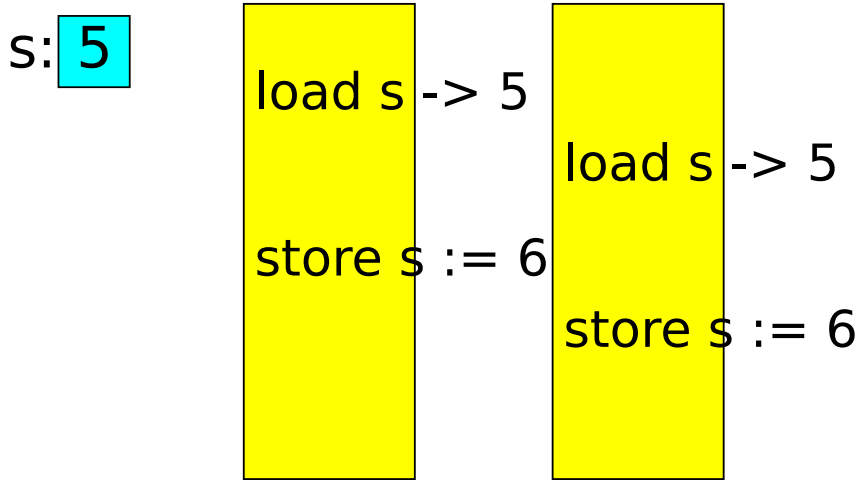
```
1  s = 0;
2  for (i = 0; i < n; i++) {
3    s += f(i);
4  }
```

の f(0), f(1), ...の計算を並列化したかったので,

```
1  s = 0;
2  #pragma omp parallel for
3  for (i = 0; i < n; i++) {
4    s += f(i);
5  }
```

とした.

- 右のようなタイミングで実行されると?



競合状態 (Race Condition)

- 競合状態: 並列に動いているスレッドが, 同じデータをアクセスしており, 誰か一人でも書き込みを行っている状態
- たいがいの場合はうまく動かない
- 対策:
  1. 競合状態を作らない (あるスレッドが書き込むなら, そのスレッド以外は触らない)
  2. 読み出しから書き込みまでの間に, 他のスレッドに書かれないようにする (atomic, critical 指示) (省略)
  3. 足し算のような, 演算順序が関係ない集計操作であれば, 各スレッドごとに計算した結果を後で一度だけ足す (reduction) (後述)

データ共有 (shared)・非共有 (private) 指示

- `parallel` pragma で作られたスレッド間で, データは
  - 文内で宣言された変数や配列は private (各スレッドが持つ)
  - それ以外は共有が基本
- それを逸脱したい場合の指示ができる
- 2.14 “Data Environments”
  - `private`
  - `firstprivate`
  - `shared`
  - `reduction` (only for `parallel` and `for`)
  - `copyin`

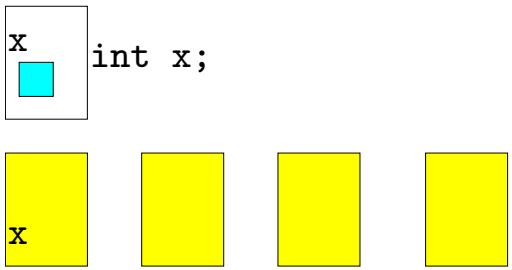
例

```
1  int main() {
2    int S;  /* shared */
3    int P;  /* made private below */
4  #pragma omp parallel private(P) shared(S)
5    {
6      int L; /* automatically private */
7      printf("S at %p, P at %p, L at %p\n",
8             &S, &P, &L);
9    }
10   return 0;
11 }
```

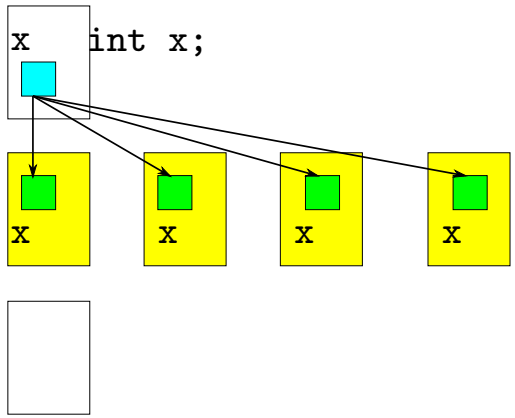
```
1  $ OMP_NUM_THREADS=2 ./a.out
2  S at 0x..777f494, P at 0x..80d0e28, L at 0x..80d0e2c
3  S at 0x..777f494, P at 0x..777f468, L at 0x..777f46c
```

挙動の図解

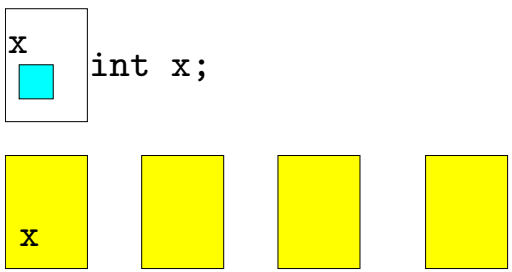
shared



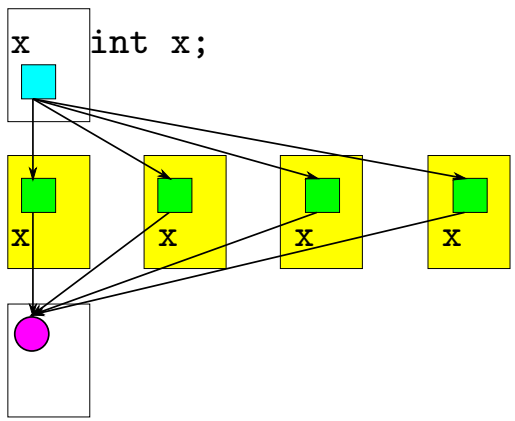
firstprivate



private



reduction

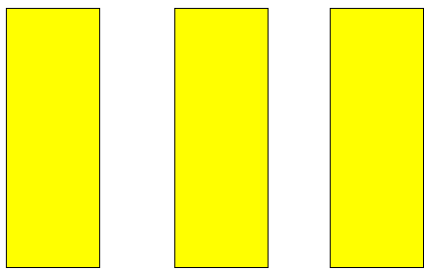


Reduction (集約操作)

- “reduction”：多くのデータを一つのデータに集約する操作
  - $v = v_1 + \cdots + v_n$
  - $v = \max(v_1, \cdots, v_n)$
  - ...
- 直接共有された変数を更新すると，競合状態

```
1  v = 0.0;
2  for (i = 0; i < n; i++) {
3    v += f(a + i * dt) * dt;
4  }
```

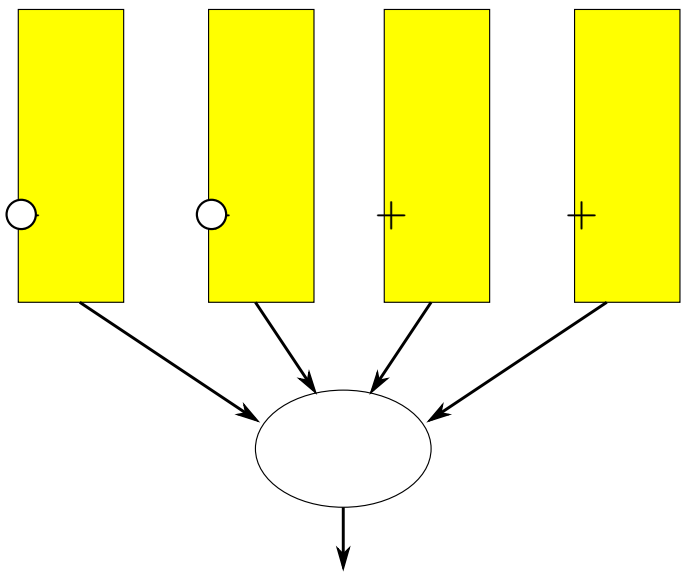
```
1  v = 0.0;
2  #pragma omp parallel shared(v)
3  #pragma omp for
4  for (i = 0; i < n; i++) {
5  #pragma omp atomic
6    v += f(a + i * dt) * dt;
7  }
```



OpenMP の Reduction 節

```
1  v = 0.0;
2  #pragma omp parallel shared(v)
3  #pragma omp for reduce(+:v)
4  for (i = 0; i < n; i++) {
5    v += f(a + i * dt) * dt;
6  }
```

- 以下を一言でやってくれる
  - 各スレッドが private な変数に集約
  - 全スレッドが終わったら一つの値に reduction



Simple reduction and user-defined reduction (2.7.1)

- reduction syntax:

```
1 #pragma omp parallel reduction(op:var,var,...)
2     S
```

  - builtin reductions
    - *op* is one of `+`, `*`, `-`, `&`, `^`, `|`, `&&`, and `||`
    - (Since 3.1) `min` or `max`
  - (Since 4.0) a user-defined reduction name
- user-defined reduction syntax:

```
1 #pragma omp declare reduction (name : type : combine statement)
```

User-defined reduction example

```
1  typedef struct {
2    int x; int y;
3  } point;
4  point add_point(point p, point q) {
5    point r = { p.x + q.x, p.y + q.y };
6    return r;
7  }
8  // declare reduction "ap" to add two points
9  #pragma omp declare reduction(ap:  point:  omp_out=add_point(omp_out, omp_in))
10
11 int main(int argc, char ** argv) {
12   int n = atoi(argv[1]);
13   point p = { 0.0, 0.0 };
14   int i;
15 #pragma omp parallel for reduction(ap : p)
16   for (i = 0; i < n; i++) {
17     point q = { i, i };
18     p = add_point(p, q);
19   }
20   printf("%d %d\n", p.x, p.y);
21   return 0;
22 }
```

密行列のマルチコア並列化

- 簡単な作戦: i ループを分割

```
1  for (i = 0; i < M; i++) {
2   for (j = 0; j < N; j++) {
3    real c = 0.0;
4    for (k = 0; k < K; k++) {
5     c += A(i,k) * B(k,j);
6    }
7    C(i,j) += c;
8   }
9  }
```

$\Rightarrow$

```
1  #pragma omp parallel for
2  for (i = 0; i < M; i++) {
3   for (j = 0; j < N; j++) {
4    real c = 0.0;
5    for (k = 0; k < K; k++) {
6     c += A(i,k) * B(k,j);
7    }
8    C(i,j) += c;
9   }
10 }
```

注: すでに行った SIMD 化と組み合わせるのも同じくらい簡単

N 体問題のマルチコア並列化

- 簡単な作戦: i ループの分割

```
1  real
2  interact_all(n, particle *p) {
3    real U = 0.0;
4
5    for (i = 0; i < n; i++) {
6      p[i].acc = vec(0,0,0);
7      for (j = 0; j < n; j++) {
8        U += interact2(p+i, p+j);
9      }
0    }
1    return 0.5 * U;
2  }
```

$\Rightarrow$

```
1  real
2  interact_all(n, particle *p) {
3    real U = 0.0;
4  #pragma omp parallel for reduction(+:U)
5    for (i = 0; i < n; i++) {
6      p[i].acc = vec(0,0,0);
7      for (j = 0; j < n; j++) {
8        U += interact2(p+i, p+j);
9      }
10   }
11   return 0.5 * U;
12 }
```

注: SIMD 化と組み合わせるのも簡単

まとめ

- 最大性能 = SIMD 並列性 × スーパスカラ並列性 × マルチコア並列性 × マルチノード並列性
- 現状 それぞれのマスターが必要:
  - SIMD : ベクトル型, intrinsics
  - マルチコア : OpenMP `#pragma omp parallel` (+ `for`)
  - マルチノード : MPI (先週)
  - スーパスカラ並列性 : 意識しなくてもある程度は勝手に. 時として意識必要 (説明せず)
- 普遍的に重要なこと: 何が起きているかの理解
- 「期待できる性能」の正しい理解と実際の性能の計測