計算科学概論
— 高性能プログラミングと性能測定

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

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
  - Jupyter 環境で学習
  - Wisteria で本格演習

Contents

ロードマップ

計算科学で代表的なワークロード

OpenMP によるマルチコア CPU 並列化
- `parallel` pragma
- Work sharing 構文
- データ共有

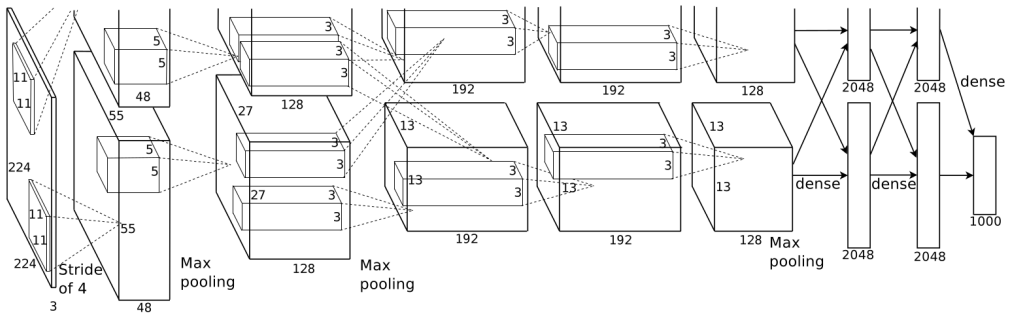
OpenMP による GPU 並列化

CPU の性能

SIMD 命令
- SIMD 命令
- ベクトル型拡張
- ベクタ intrinsics 関数

計算科学で代表的なワークロード (1) — 密行列

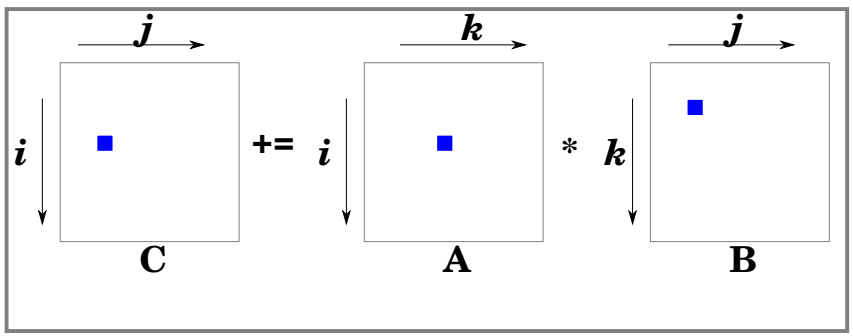
- 例: 境界要素法, 深層学習



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

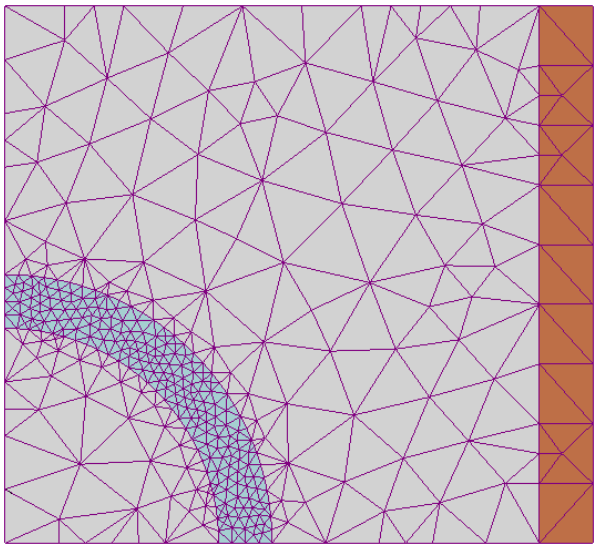
- 計算カーネル: 行列・行列積

```
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



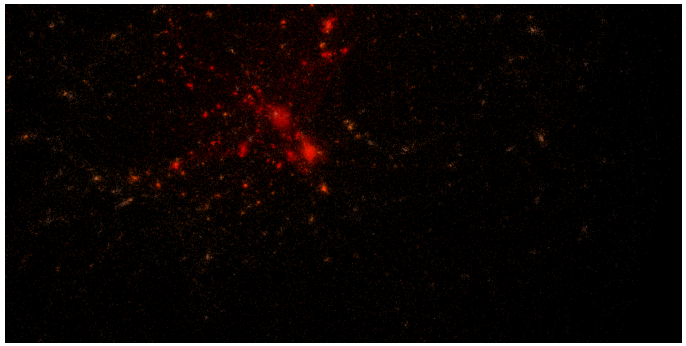
www.wikipedia.org

- 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

計算科学で代表的なワークロード

OpenMP によるマルチコア CPU 並列化
- `parallel` pragma
- Work sharing 構文
- データ共有

OpenMP による GPU 並列化

CPU の性能

SIMD 命令
- SIMD 命令
- ベクトル型拡張
- ベクタ intrinsics 関数

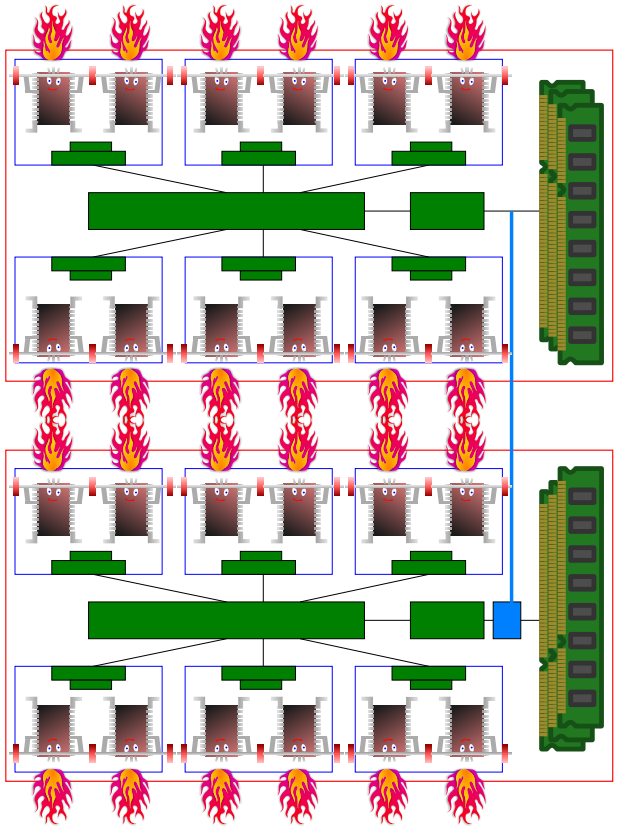
マルチコア並列化の選択肢

- 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
- 最近は GPU もサポートしている
  - 本演習ではそのこともあり OpenMP で CPU, GPU プログラミングを行う
- 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, Clang (LLVM), NVIDIA はじめ多くのコンパイラでサポート
  - 本演習では GPU もサポートしている NVIDIA コンパイラを使ってみる (自分の好きなのを使って良い)



OpenMP 参考文献

- 公式 HP `http://openmp.org/`
- 仕様 `https://www.openmp.org/specifications/`
- 最新の仕様 5.1 (pdf, html)

OpenMP プログラムのコンパイルと実行 (GCC)

- GCC: コンパイル時に`-fopenmp` オプション

```
$ gcc -Wall -fopenmp program.c   # C++ は g++
```

- NVIDIA HPC SDK (nvc, nvc++): コンパイル時に `-mp=multicore` オプション

```
$ nvc -mp=multicore program.c    # C++ は nvc++
```

- 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

```
$ OMP_NUM_THREADS=1 ./a.out  # use 1 thread
$ 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, Oakbridge CX 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

```
#PJM --omp thread=x
```

ロードマップ

計算科学で代表的なワークロード

OpenMP によるマルチコア CPU 並列化
- `parallel` pragma
- Work sharing 構文
- データ共有

OpenMP による GPU 並列化

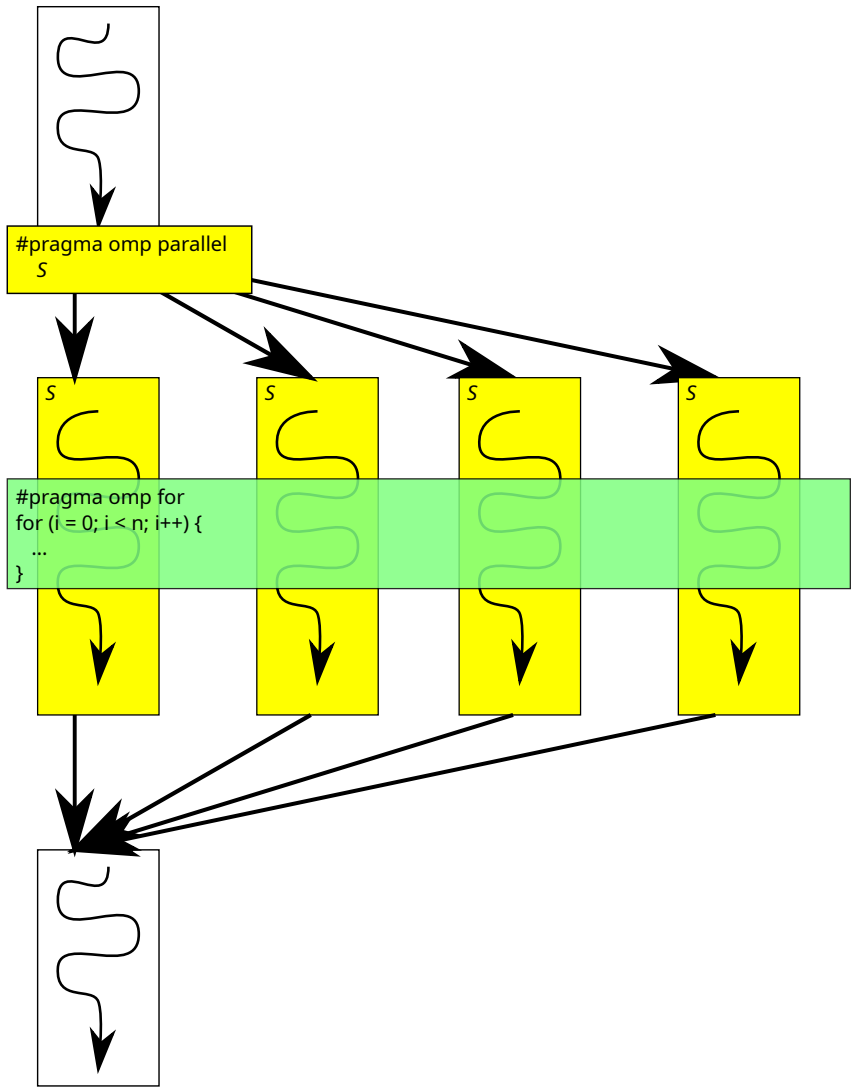
CPU の性能

SIMD 命令
- SIMD 命令
- ベクトル型拡張
- ベクタ intrinsics 関数

基本中の基本の二つ

- `#pragma omp parallel` によってスレッドを生成 (2.6)
- その後 `#pragma omp for` で for 文の繰り返しをスレッドで分担 (2.11.4)

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

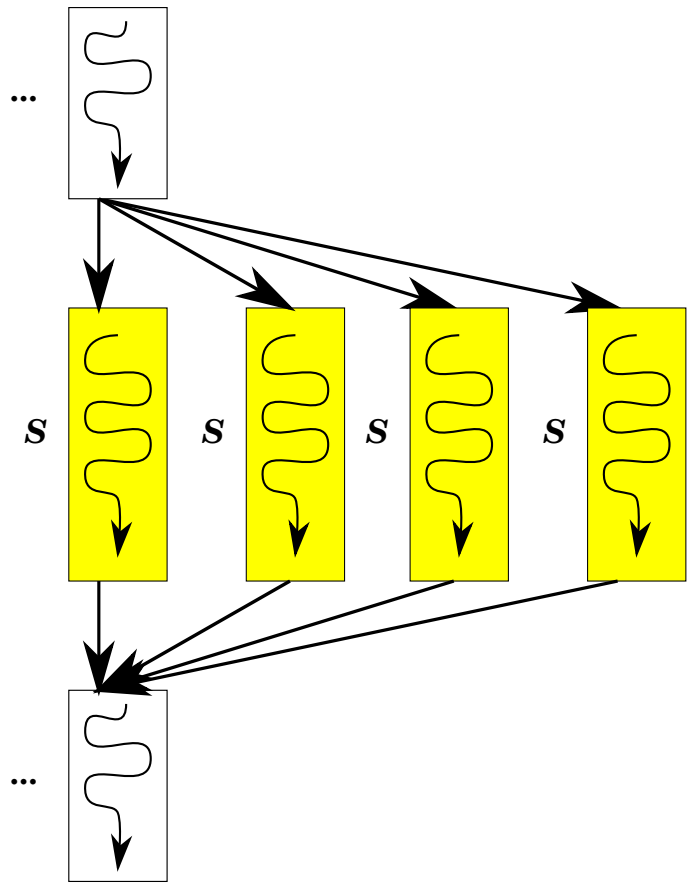


#pragma parallel

- 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

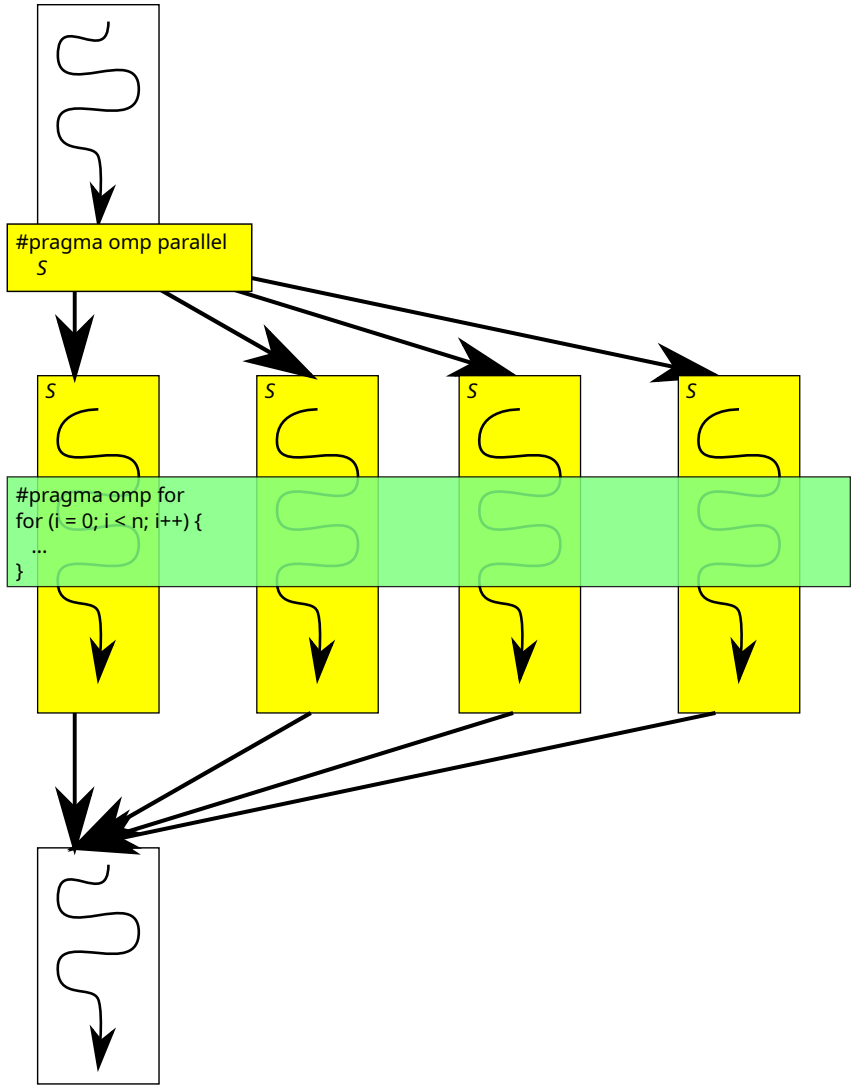
- 理論的には `parallel`, `omp_get_num_threads()`, `omp_get_thread_num()` だけで生きていくことが可能
- だが不便すぎる
- OpenMP は仕事をスレッド間で分割する手段 *(work sharing* 構文*)* も提供している
  - for
  - task (省略)
  - section (省略)

#pragma omp for (2.11.4 Worksharing-loop)

- 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 (2.11.1)

```
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 (2.11.4)

- `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 }
```

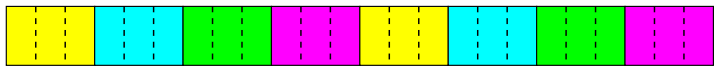
- 3 つの選択肢 (`static, dynamic, guided`)

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

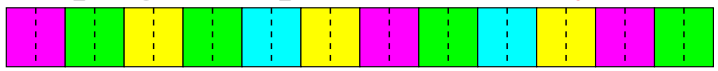
{\tt\#pragma omp for schedule(static)}

{\tt\#pragma omp for schedule(static,3)}

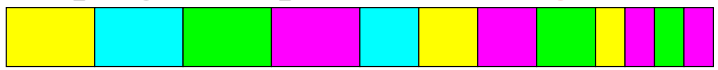


{\tt\#pragma omp for schedule(dynamic)}

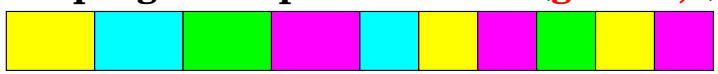
{\tt\#pragma omp for schedule(dynamic,2)}



{\tt\#pragma omp for schedule(guided)}



{\tt\#pragma omp for schedule(guided,2)}



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

計算科学で代表的なワークロード

OpenMP によるマルチコア CPU 並列化

- `parallel` pragma
- Work sharing 構文
- データ共有

OpenMP による GPU 並列化

CPU の性能

SIMD 命令

- SIMD 命令
- ベクトル型拡張
- ベクタ intrinsics 関数

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?

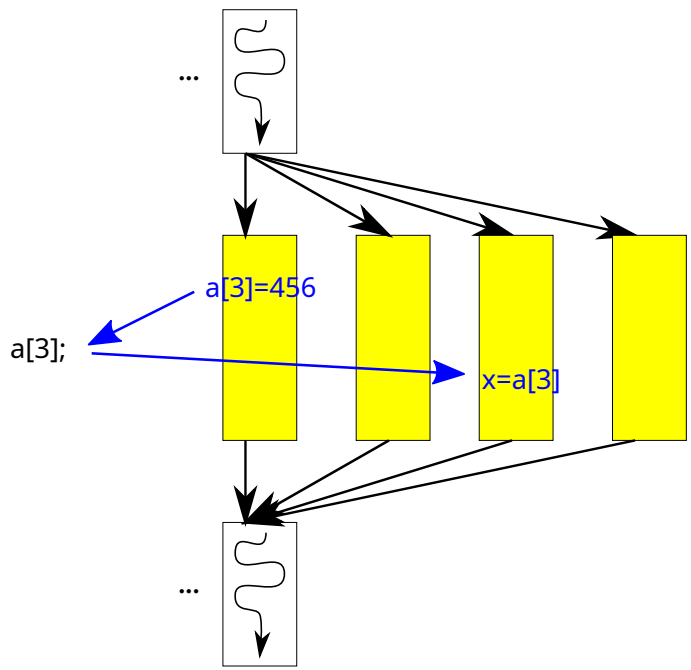
OpenMP は「共有メモリ」モデル

- `# 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

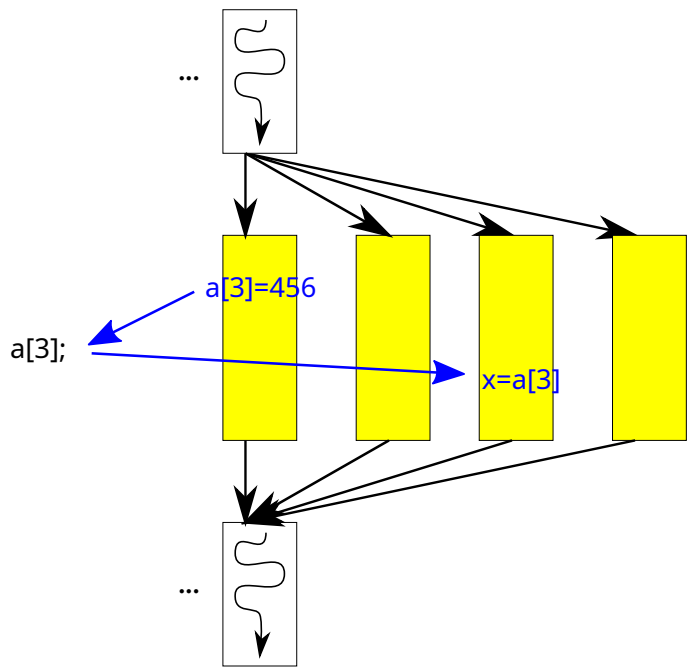
- メモリ? アドレス空間? 共有? どういう意味?
  - ≈ プログラム言語の言葉で言えば, たいがいの「変数や配列を共有する」という意味です
  - ⇒ つまり, あるスレッドが変数や配列の値を更新して, 他のスレッドが読んだら, それはちゃんと伝わる, という意味



OpenMP は「共有メモリ」モデル

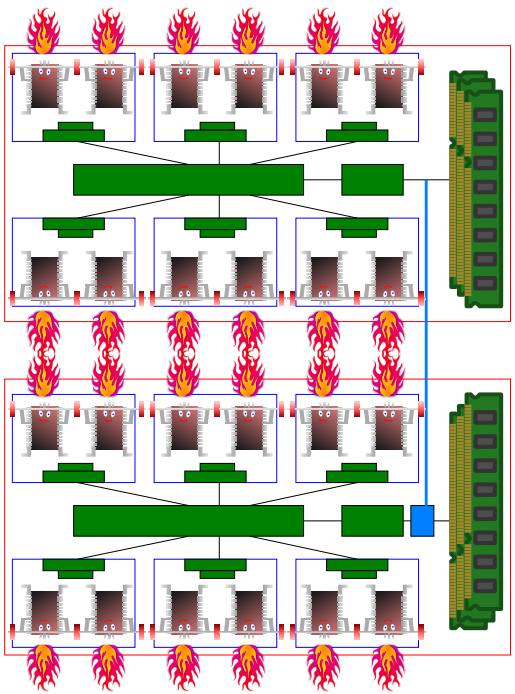
- `# pragma omp parallel` で作られたスレッドは, メモリ (アドレス空間) を共有する

- 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



- ありがたいことに, プログラミング言語処理系がやることは並列プログラムであろうとなかろうとほぼ同じ
- ただしこれは, 一つのノード (≈ 箱) 内での話
- 今日, 複数のノード間では, そのような機能はない

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

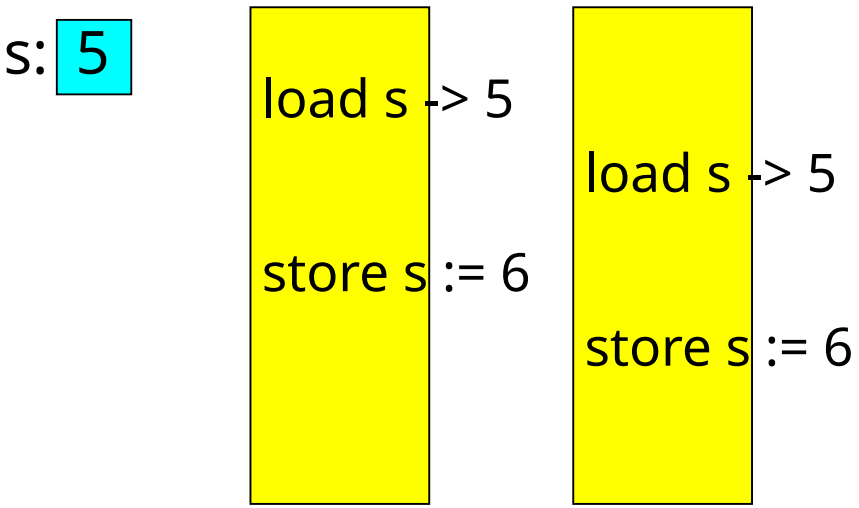
```
1  s = 0;
2  for (i = 0; i < n; i++) {
3    s += f(i);
4  }
```

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
- 2.21 Data Environment
  - `private`
  - `firstprivate`
  - `shared`
  - `reduction` (only for `parallel` and `for`)
  - `copyin`

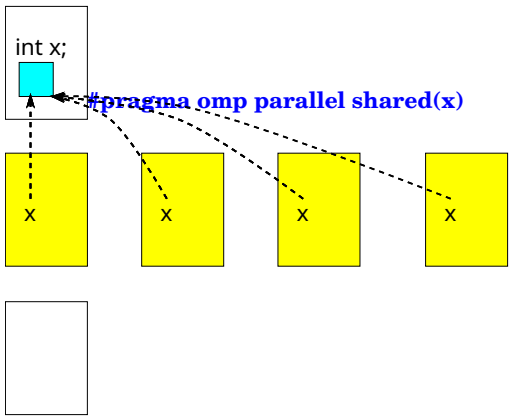
例

```
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 }
```

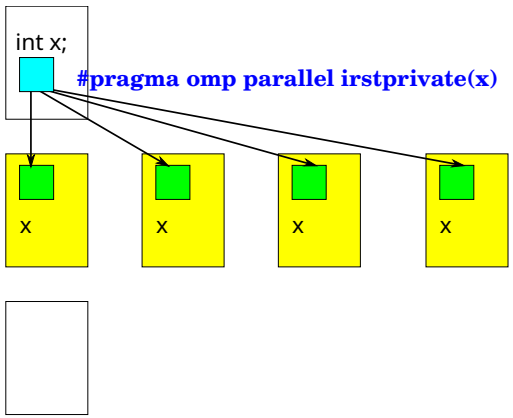
```
1  $ OMP_NUM_THREADS=2 ./a.out
2  S at 0x..777f494, P at 0x..80d0e28, L at 0x..80d0e2c
3  S at 0x..777f494, P at 0x..777f468, L at 0x..777f46c
```

挙動の図解

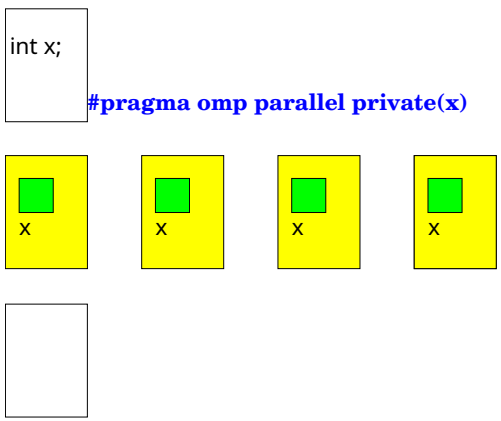
shared



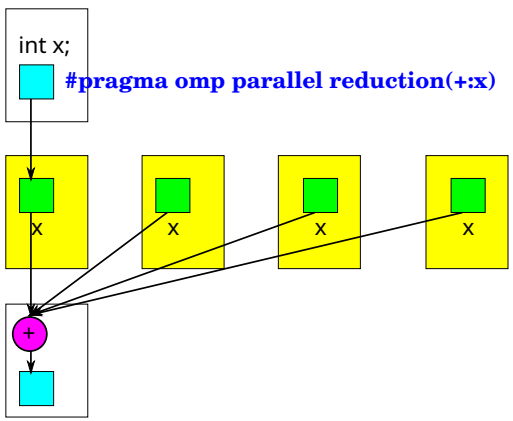
firstprivate



private



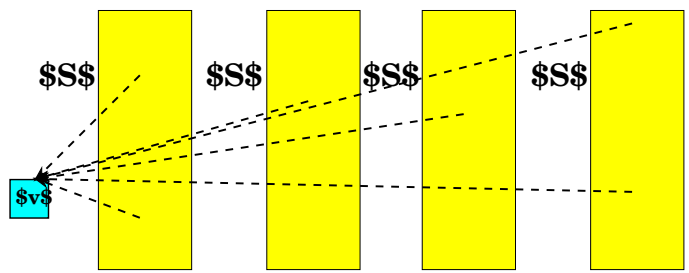
reduction



Reduction (2.21.5)

- “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

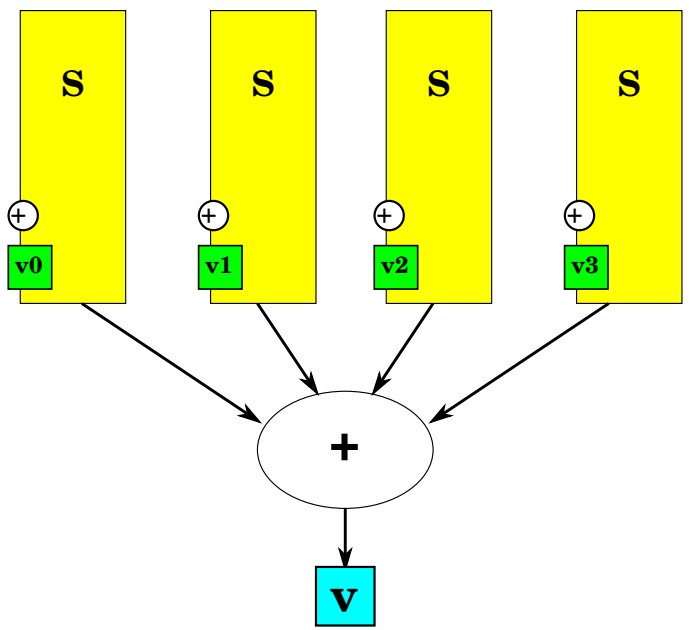
```
1  v = 0.0;
2  for (i = 0; i < n; i++) {
3    v += f(a + i * dt) * dt;
4  }
```



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



Reduction の記法とユーザ定義 reduction

- reduction (2.21.5.4):

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

  op は以下のどれか
  - `+, *, -, &, ^, |, &&, min, max, ||`
  - ユーザ定義の reduction 名

- ユーザ定義 reduction (2.21.5.7)

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

ユーザ定義の reduction の例

構造体 point に対する reduction

```
1  typedef struct {
2    int x; int y;
3  } point;
4  point add_point(point p, point q) {
5    /* p + q に相当する関数を定義 */
6    point r = { p.x + q.x, p.y + q.y };
7    return r;
8  }
9  // “ap” という名前で reduction を定義
10 #pragma omp declare reduction(ap:  point:  omp_out=add_point(omp_out, omp_in))
11
12 int main(int argc, char ** argv) {
13   int n = atoi(argv[1]);
14   point p = { 0.0, 0.0 };
15   int i;
16 #pragma omp parallel for reduction(ap : p)
17   for (i = 0; i < n; i++) {
18     point q = { i, i };
19     p = add_point(p, q);
20   }
21   printf("%d %d\n", p.x, p.y);
22   return 0;
23 }
```

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

ロードマップ

計算科学で代表的なワークロード

OpenMP によるマルチコア CPU 並列化
- `parallel` pragma
- Work sharing 構文
- データ共有

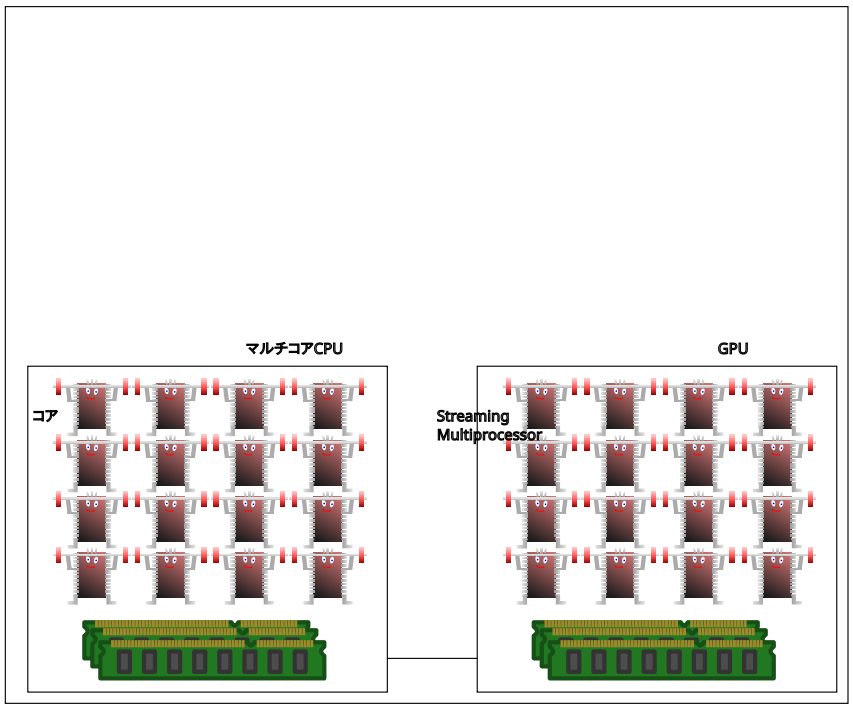
OpenMP による GPU 並列化

CPU の性能

SIMD 命令
- SIMD 命令
- ベクトル型拡張
- ベクタ intrinsics 関数

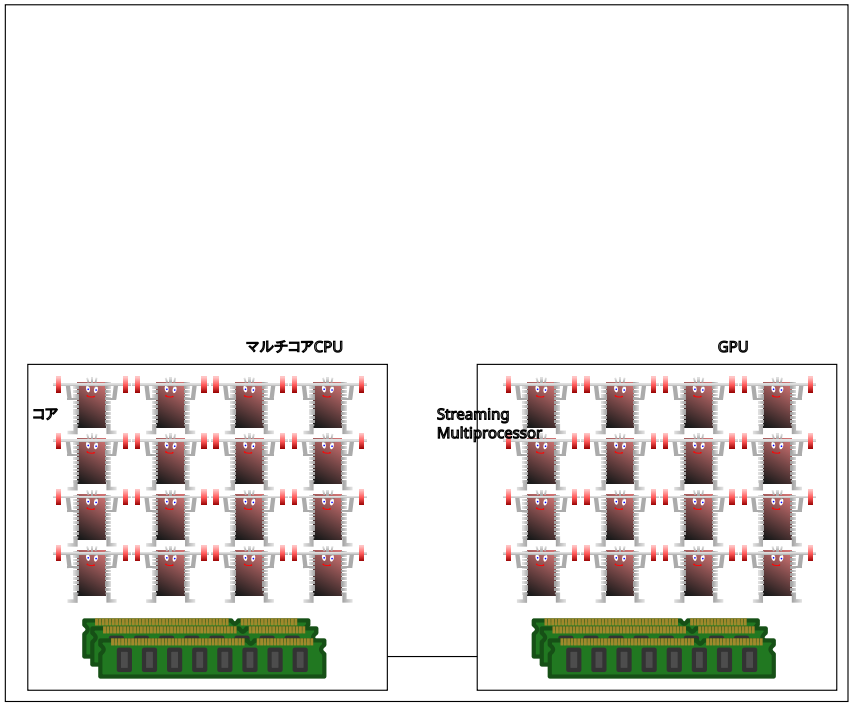
GPU プログラミング vs. CPU (だけの) プログラミング

1. CPU と GPU は別のマシン
   - ⇒ どこを GPU で実行 (にオフロード) するかを指定する必要がある (`#pragma omp target`)
2. CPU↔GPU 間でメモリが (ハードウェアで) 共有されているわけではない
   - ⇒ データを明示的に移動させる必要がある (場合がある) (`map` 節)
   - 注: 最近のハードウェアでは不要になりつつある



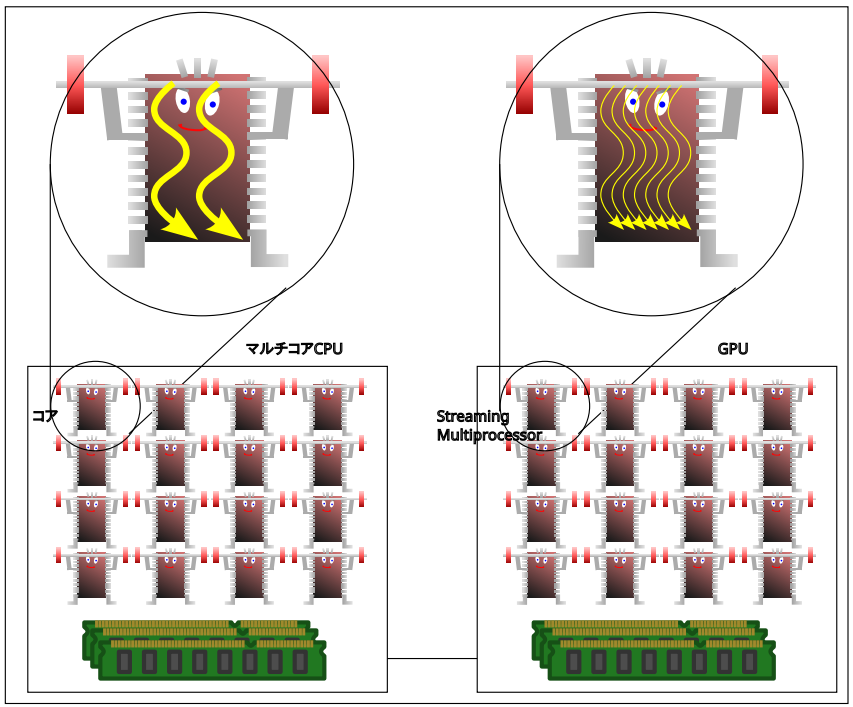
CPU vs. GPU : アーキテクチャ・特性の違い

- 共通点: マルチコア並列性
  - GPU の Streaming Multiprocessor ≈ CPU のコア



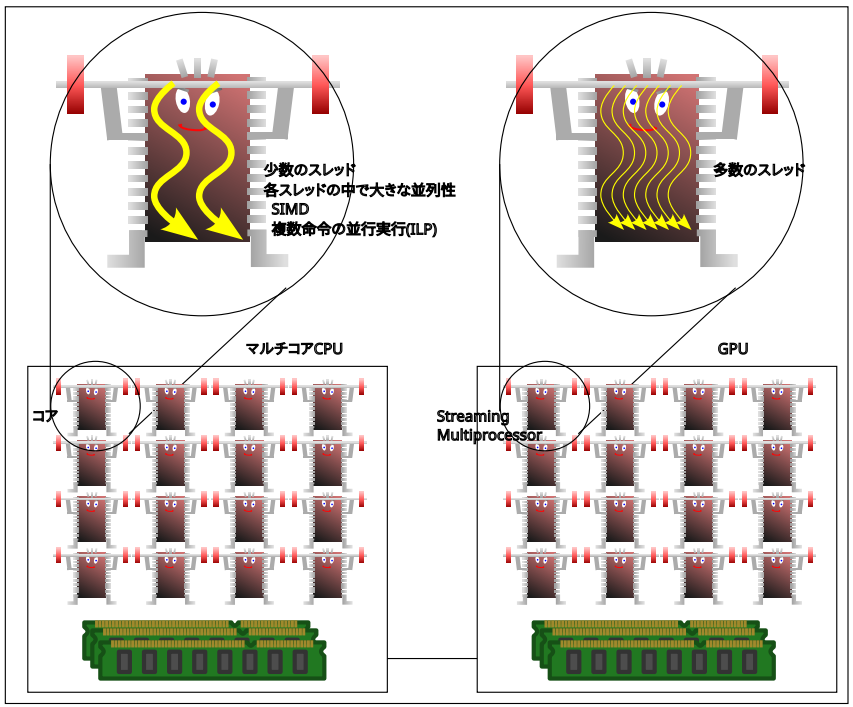
CPU vs. GPU：アーキテクチャ・特性の違い

- 共通点: マルチコア並列性
  - GPU の Streaming Multiprocessor ≈ CPU のコア
- 主な違い: コア内の性能 (並列性) 向上手段
  - GPU
    - 多数の「スレッド」をオーバーラップ実行
  - CPU
    - 1 命令で多数の演算を実行 (SIMD)
    - 1 スレッド内の多数の命令をオーバーラップ実行 (ILP)



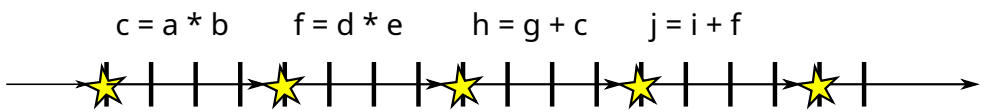
CPU vs. GPU：アーキテクチャ・特性の違い

- 共通点: マルチコア並列性
  - GPU の Streaming Multiprocessor ≈ CPU のコア
- 主な違い: コア内の性能 (並列性) 向上手段
  - GPU
    - 多数の「スレッド」をオーバーラップ実行
  - CPU
    - 1 命令で多数の演算を実行 (SIMD)
    - 1 スレッド内の多数の命令をオーバーラップ実行 (ILP)



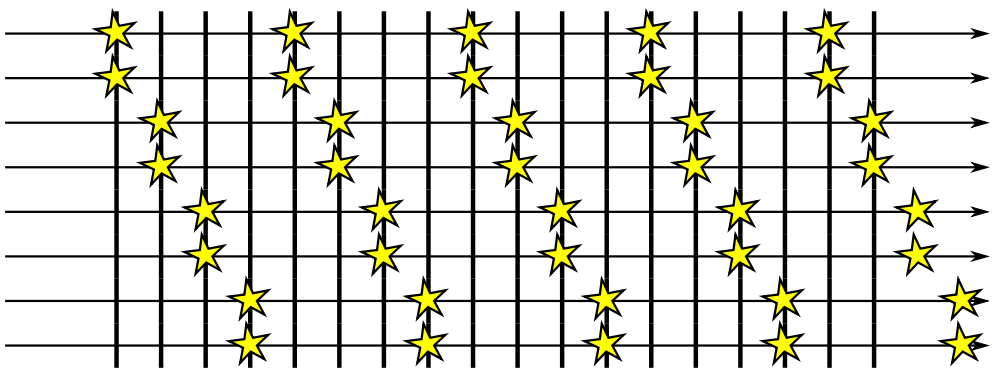
GPU「1 コア内の多数のスレッド」の意味

- 各命令の結果が出るには数〜数百クロックの時間がかかり, GPU の各スレッドは先の命令が終わるまで次の命令の実行はできない
- ⇒ 1 スレッドだけではコアの性能をほとんど生かせない
  - 比喩: ガラガラの高速道路. いくら空いていても 1 台の車が, 東京から大阪まで行く時間は 1 時間以内にはならない



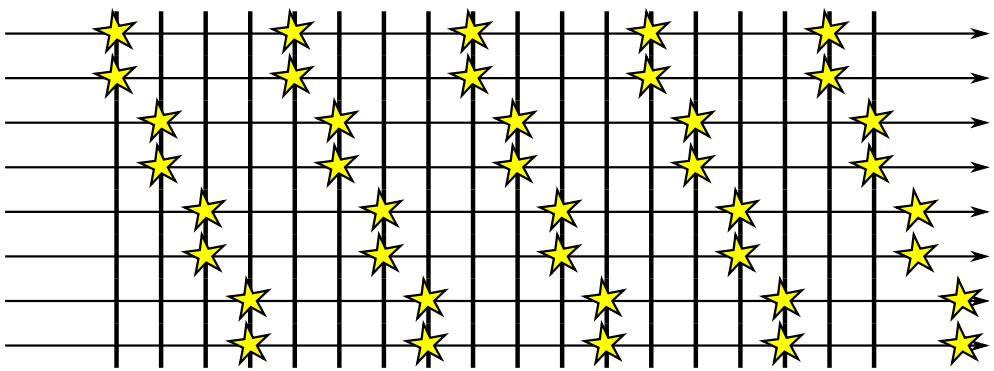
GPU「1 コア内の多数のスレッド」の意味

- 各命令の結果が出るには数〜数百クロックの時間がかかり, GPU の各スレッドは先の命令が終わるまで次の命令の実行はできない
- ⇒ 1 スレッドだけではコアの性能をほとんど生かせない
  - 比喩: ガラガラの高速道路. いくら空いていても 1 台の車が, 東京から大阪まで行く時間は 1 時間以内にはならない
- そこで GPU は多数のスレッドを少しずつ前進させて性能を稼ぐ
  - 比喩: 混んでる高速道路. 渋滞はあるかもしれないが, 単位時間あたりに運ぶ荷物 (人) の量は増える



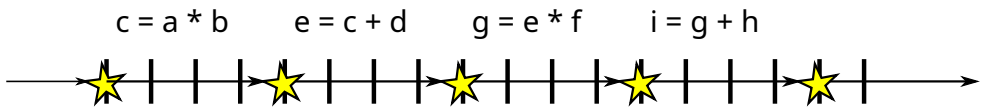
GPU「1 コア内の多数のスレッド」の意味

- 各命令の結果が出るには数〜数百クロックの時間がかかり, GPU の各スレッドは先の命令が終わるまで次の命令の実行はできない
- ⇒ 1 スレッドだけではコアの性能をほとんど生かせない
  - 比喩: ガラガラの高速道路. いくら空いていても 1 台の車が, 東京から大阪まで行く時間は 1 時間以内にはならない
- そこで GPU は多数のスレッドを少しずつ前進させて性能を稼ぐ
  - 比喩: 混んでる高速道路. 渋滞はあるかもしれないが, 単位時間あたりに運ぶ荷物 (人) の量は増える
- GPU ハードウェアは多数のスレッドを細かい時間スケール (e.g., クロックごと) でかわりばんこに実行



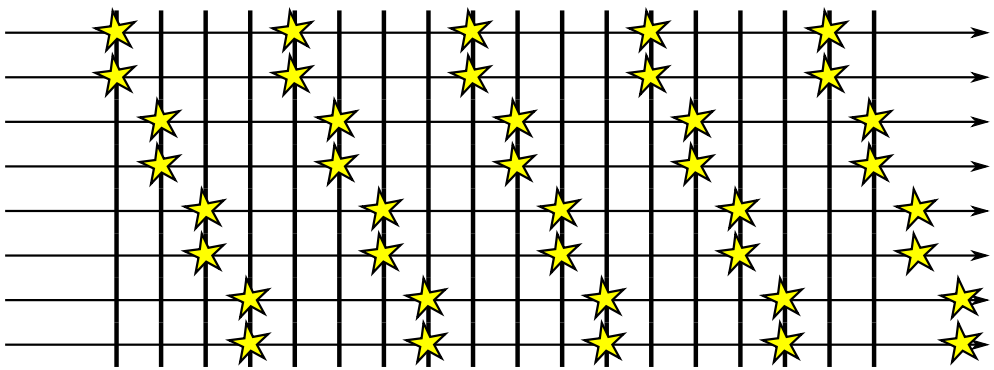
対比: CPU「1 スレッド内の並列性」

- GPU が常に「スレッド」で並列性を抽出するのに対し, CPU は 1 コア内のスレッド数はわずか (1〜数個)
- 変わりに 1 スレッド内の命令で多数の並列性を抽出する
  - SIMD : 1 つの命令で多数の演算
  - 命令レベル並列 : 依存関係のない (先行命令の結果を待たなくても実行できる) 命令をオーバーラップして実行

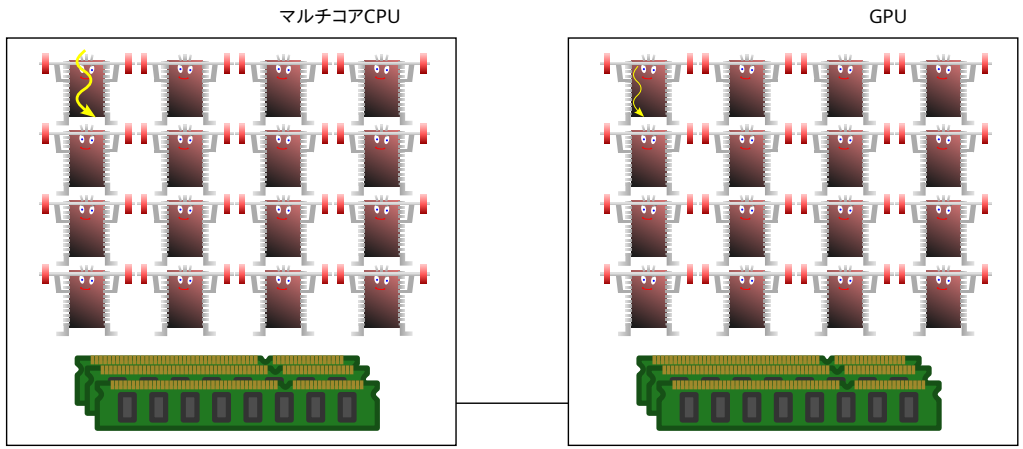


対比: CPU「1 スレッド内の並列性」

- GPU が常に「スレッド」で並列性を抽出するのに対し, CPU は 1 コア内のスレッド数はわずか (1〜数個)
- 変わりに 1 スレッド内の命令で多数の並列性を抽出する
  - SIMD : 1 つの命令で多数の演算
  - 命令レベル並列 : 依存関係のない (先行命令の結果を待たなくても実行できる) 命令をオーバーラップして実行

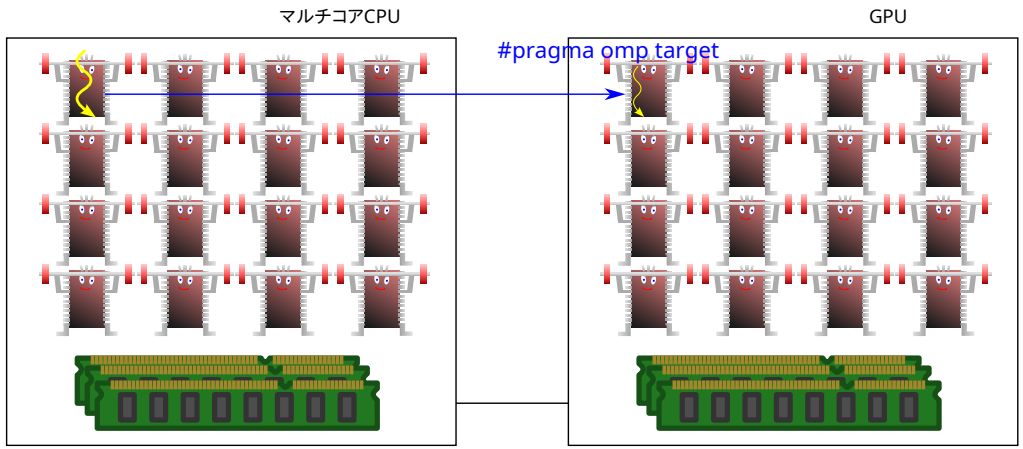


OpenMP プログラミングで出現する構文 (1) — 制御の移動, スレッド生成



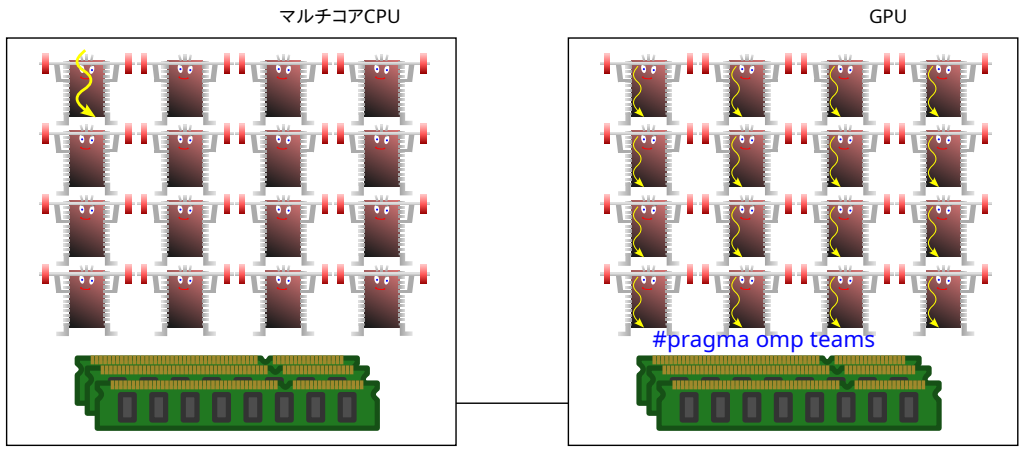
OpenMP プログラミングで出現する構文 (1) — 制御の移動, スレッド生成

- `#pragma omp target` 実行を GPU に「移す」



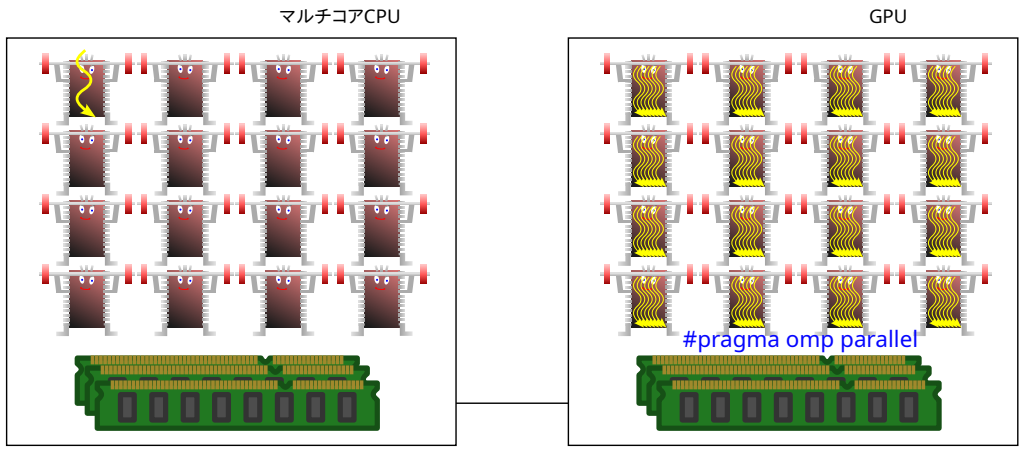
OpenMP プログラミングで出現する構文 (1) — 制御の移動, スレッド生成

- `#pragma omp target` 実行を GPU に「移す」
- `#pragma omp teams` GPU に多数の「チーム」を作る
  - 1 チームは 1 つの Streaming Multiprocessor 上で実行
- `#pragma omp distribute` for 文の繰り返しを多数のチームで分割



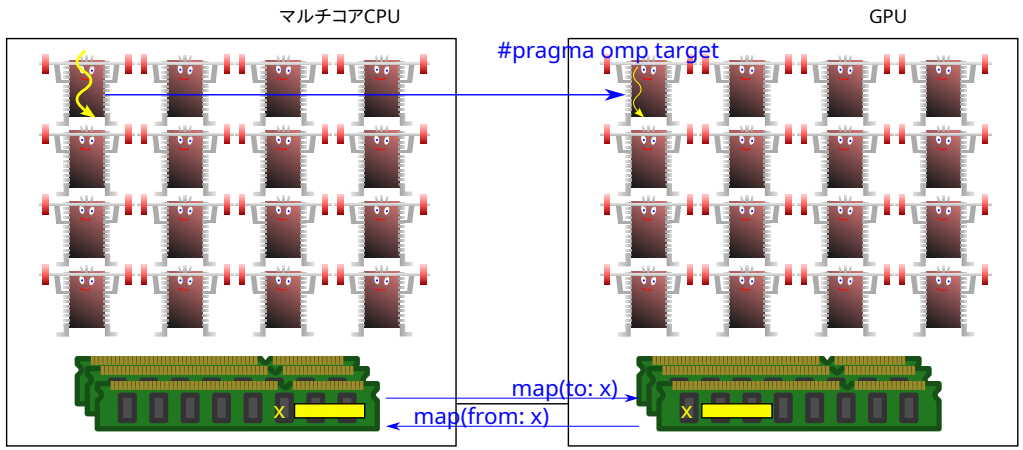
OpenMP プログラミングで出現する構文 (1) — 制御の移動, スレッド生成

- `#pragma omp target` 実行を GPU に「移す」
- `#pragma omp teams` GPU に多数の「チーム」を作る
  - 1 チームは 1 つの Streaming Multiprocessor 上で実行
- `#pragma omp distribute` for 文の繰り返しを多数のチームで分割
- 以下は CPU でも既出
- `#pragma omp parallel` 各チームに多数の「スレッド」を作る
- `#pragma omp for` for 文の繰り返しを多数のスレッドで分割



OpenMP プログラミングで出現する構文 (2) — データマッピング

- `#pragma omp target map(to:`x`)` x を target 実行に先立ち, 正しくする (≈ CPU → GPU 転送)
- `#pragma omp target map(from:`x`)` x を target 実行後, 正しくする (≈ GPU → CPU 転送)
- `#pragma omp target map(tofrom:`x`)` `to:`, `from:` 両方の効果
- 注: 実行を GPU に移さずにデータだけを移す指令`#pragma omp target data map(...)` もある



頻出単純パターン

- GPU 全体を使って並列実行

```
1 #pragma omp target teams distribute parallel for num_teams(..)
            num_threads(..)  map(...)
2 for (i = 0; i < ...; i++) {
3   ...
4 }
```

- キーワードの多さにめまいがするが,
  - CPU の `parallel` (スレッド生成) + `for` (ループの分割) の組合わせに加え,
  - GPU では `teams` (スレッド生成) + `distributed` (ループの分割) の組合わせが加わる

  と思えば良い
- CPU ではチームは 1 つだけと決まっていると思っても良い

ロードマップ

計算科学で代表的なワークロード

OpenMP によるマルチコア CPU 並列化
- `parallel` pragma
- Work sharing 構文
- データ共有

OpenMP による GPU 並列化

CPU の性能

SIMD 命令
- SIMD 命令
- ベクトル型拡張
- ベクタ intrinsics 関数

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	≈ 300W
NVIDIA Tesla V100	7.5	15.0	≈ 300W
NVIDIA Tesla A100 PCIe (*)	9.7	19.5	≈ 300W
NVIDIA Tesla H100 PCIe (*)	30.0	60.0	≈ 350W
Intel Knights Landing	3.0	6.0	≈ 250W
Intel Platinum 8368 (**)	2.9	5.8	≈ 270W
Intel Platinum 8360Y (***)	2.8	5.5	≈ 250W

- (*) Tensor Core (行列積用の特殊命令) を使わない場合
- (**) 学習環境 Jupyter (mdx)
- (***) 演習環境 Wisteria Aquarius (Intel CPU + GPU)

「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

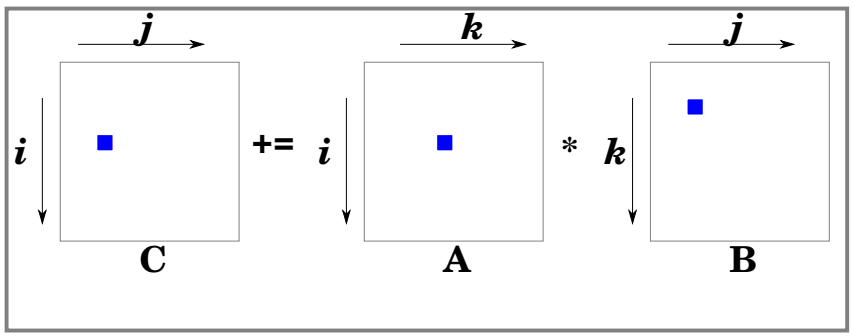
「性能」について知っておかねばならない事

- 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「性能」 2.8 TFLOPS のプロセッサ Intel Xeon Platinum 8360Y CPU (Oakbridge CX の計算ノードのプロセッサ) で実行したときの性能は?

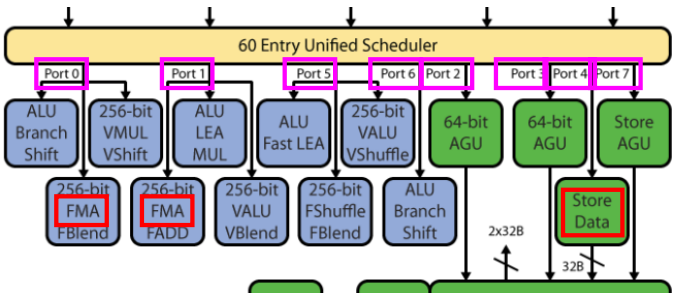
実際の結果

```
A = 1000 x 1000 (4000000 bytes)
B = 1000 x 1000 (4000000 bytes)
C = 1000 x 1000 (4000000 bytes)
repeat C += A * B 1 times
2000000000 flops, total 12000000 bytes
2762354648 clocks
1.025528 sec
0.724 flops/clock
1.950215 GFLOPS
2.762 clocks/muladd
OK: max relative error = 0.000000
```

- CPU の最大性能 4.8 TFLOPS と比べると約 1/2461 (!)
- ちなみに計算ノードには同 CPU が 2 個積まれているので, 計算ノードの最大性能は 9.6 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



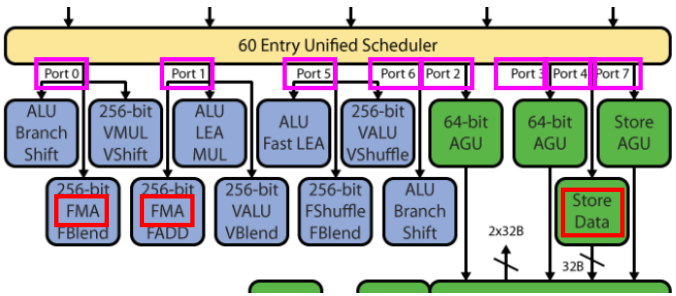
source:

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:

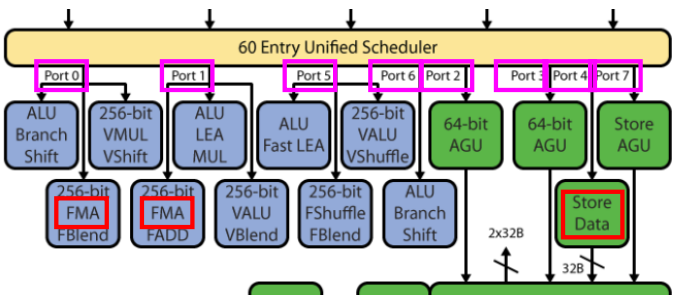
http://www.realworldtech.com/haswell-cpu/

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

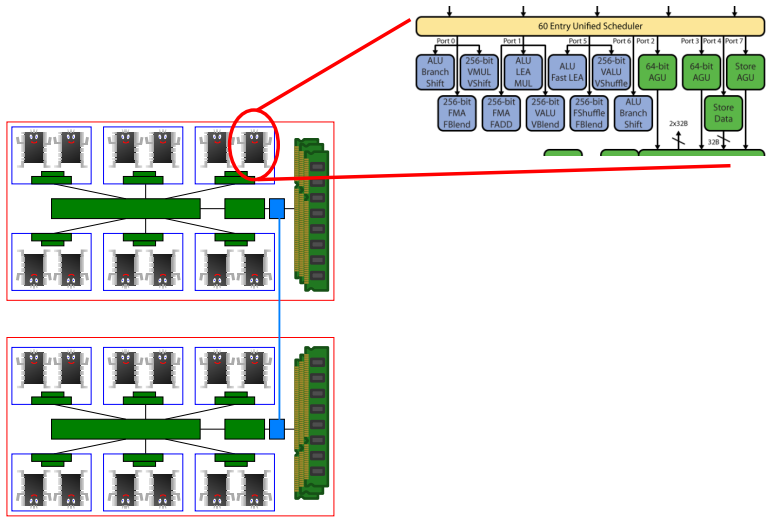


source:

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

例: Intel Platinum 8360Y の場合

- 単精度 (32 bit) 浮動小数点数の場合:

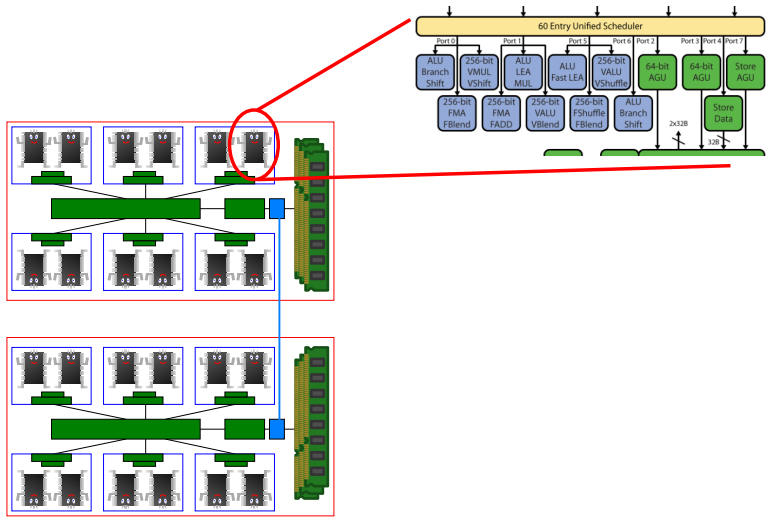


例: Intel Platinum 8360Y の場合

- 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

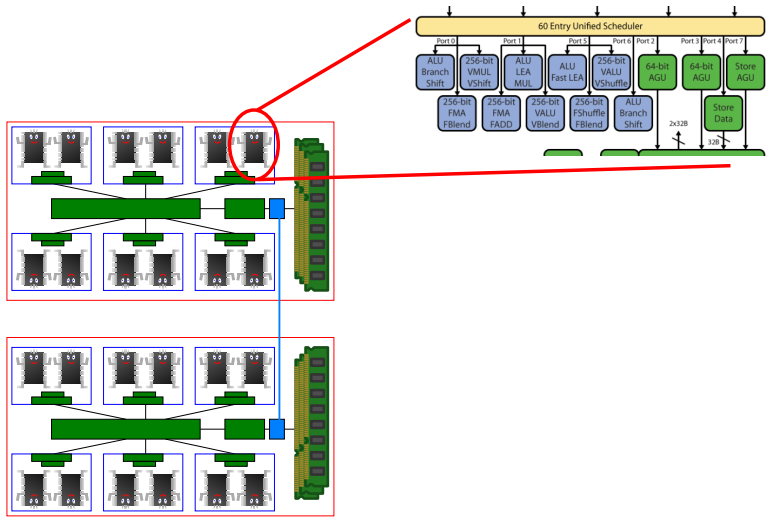
$=$ 32 flops



例: Intel Platinum 8360Y の場合

- 単精度 (32 bit) 浮動小数点数の場合:

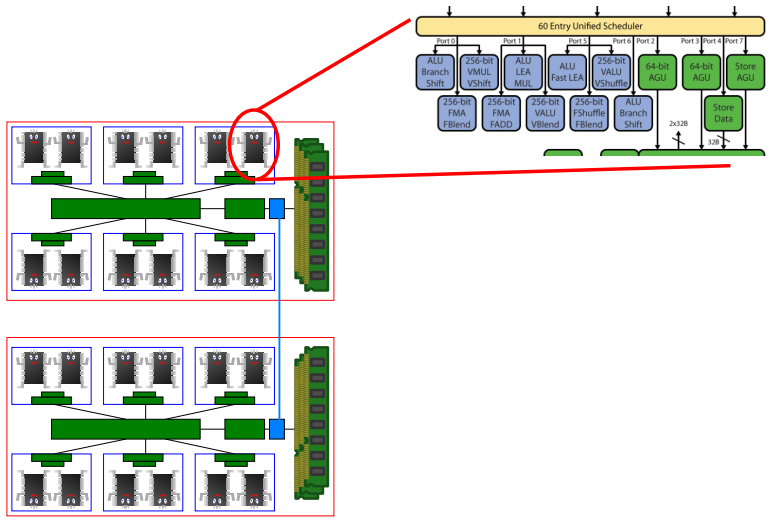
$$
\begin{aligned}
& 1\text{ 浮動小数点命令あたりの演算数} \\
\times\quad & 1\text{ クロックに実行できる浮動小数点命令数} \\
\\
= \quad & 32\text{ flops} \times 2
\end{aligned}
$$



例: Intel Platinum 8360Y の場合

- 単精度 (32 bit) 浮動小数点数の場合:

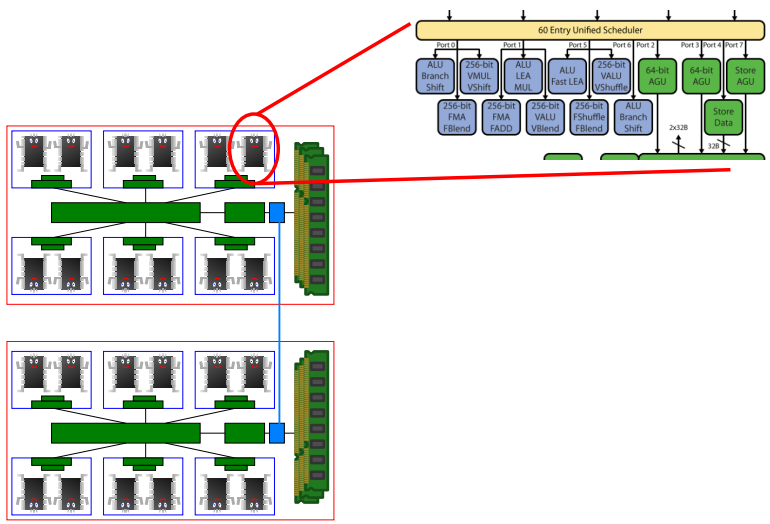
$$
\begin{aligned}
& 1 \text{ 浮動小数点命令あたりの演算数} \\
\times \quad & 1 \text{ クロックに実行できる浮動小数点命令数} \\
\times \quad & \text{クロック周波数 (秒あたりのクロック数)} \\
\\
= \quad & 32 \text{ flops} \times 2 \times 2.4 \text{ GHz } (1/\text{sec})
\end{aligned}
$$



例: Intel Platinum 8360Y の場合

- 単精度 (32 bit) 浮動小数点数の場合:

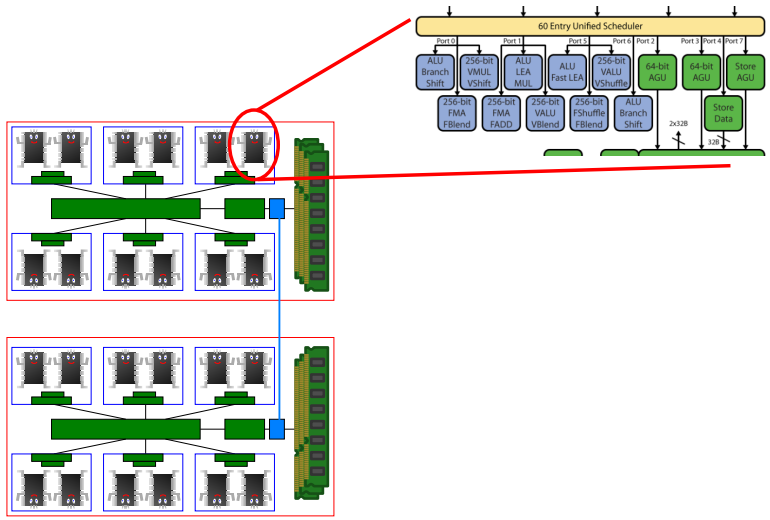
$$
\begin{aligned}
& 1\text{ 浮動小数点命令あたりの演算数} \\
\times\ & 1\text{ クロックに実行できる浮動小数点命令数} \\
\times\ & \text{クロック周波数 (秒あたりのクロック数)} \\
\times\ & \text{コア数} \\
=\ & 32\text{ flops} \times 2 \times 2.4\text{ GHz (1/sec)} \times 36
\end{aligned}
$$



例: Intel Platinum 8360Y の場合

- 単精度 (32 bit) 浮動小数点数の場合:

$$
\begin{aligned}
& 1 \text{ 浮動小数点命令あたりの演算数} \\
\times\ & 1 \text{ クロックに実行できる浮動小数点命令数} \\
\times\ & \text{クロック周波数 (秒あたりのクロック数)} \\
\times\ & \text{コア数} \\
=\ & 32 \text{ flops} \times 2 \times 2.4 \text{ GHz (1/sec)} \times 36 \\
=\ & 5529.6 \text{ GFLOPS (5.5 TFLOPS)}
\end{aligned}
$$

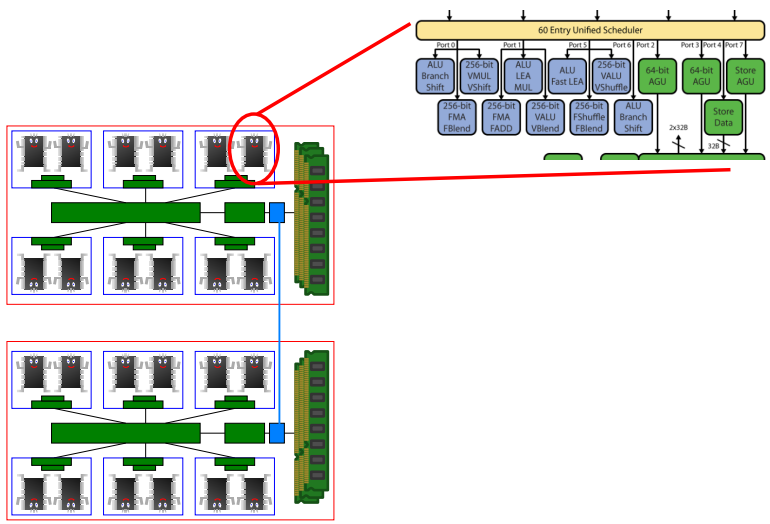


例: Intel Platinum 8360Y の場合

- 単精度 (32 bit) 浮動小数点数の場合:

$$
\begin{aligned}
& 1 \text{ 浮動小数点命令あたりの演算数} \\
\times\ & 1 \text{ クロックに実行できる浮動小数点命令数} \\
\times\ & \text{クロック周波数 (秒あたりのクロック数)} \\
\times\ & \text{コア数} \\
=\ & 32 \text{ flops} \times 2 \times 2.4 \text{ GHz (1/sec)} \times 36 \\
=\ & 5529.6 \text{ GFLOPS (5.5 TFLOPS)}
\end{aligned}
$$

- 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

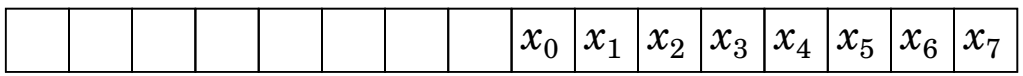


1 浮動小数点命令あたりの演算数について

- Intel Platinum は 512 bit 幅のレジスタ (SIMD レジスタ) を持つ

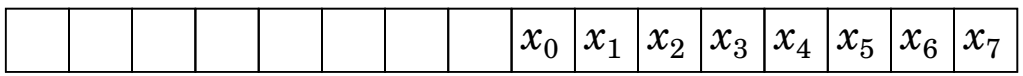
1 浮動小数点命令あたりの演算数について

- Intel Platinum は 512 bit 幅のレジスタ (SIMD レジスタ) を持つ
- 512 bit (64 bytes) = 単精度 (32 bit / 4 bytes)) 浮動小数点数が 16 つ



1 浮動小数点命令あたりの演算数について

- Intel Platinum は 512 bit 幅のレジスタ (SIMD レジスタ) を持つ
- 512 bit (64 bytes) = 単精度 (32 bit / 4 bytes)) 浮動小数点数が 16 つ

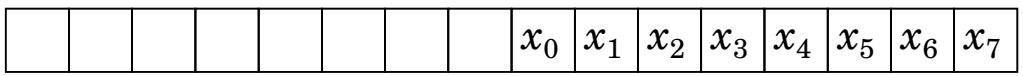


- 乗算と加算を同時に行う命令 (fmadd) があり, それを使うと 1 命令で $16 \times 2 = 32$ 演算ということになる

$$c = a * b + c$$

1 浮動小数点命令あたりの演算数について

- Intel Platinum は 512 bit 幅のレジスタ (SIMD レジスタ) を持つ
- 512 bit (64 bytes) = 単精度 (32 bit / 4 bytes)) 浮動小数点数が 16 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 $16 \times 2 = 32$ 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 8 つを保持することもできる (したがって fmadd の演算数 =16)

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	→	1/16	(6.25%)
並列化して (でき) いない	→	1/36	(2.78%)
どれも使って (え) ない	→	1/1152	(0.087%)

- ⇒ 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

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

OpenMP によるマルチコア CPU 並列化

- `parallel` pragma
- Work sharing 構文
- データ共有

OpenMP による GPU 並列化

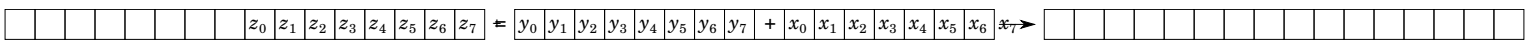
CPU の性能

SIMD 命令

- SIMD 命令
- ベクトル型拡張
- ベクタ intrinsics 関数

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

計算科学で代表的なワークロード

OpenMP によるマルチコア CPU 並列化

- `parallel` pragma
- Work sharing 構文
- データ共有

OpenMP による GPU 並列化

CPU の性能

SIMD 命令

- SIMD 命令
- ベクトル型拡張
- ベクタ intrinsics 関数

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)
- 最近は AVX-512 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-512 命令 (アセンブリ言語)

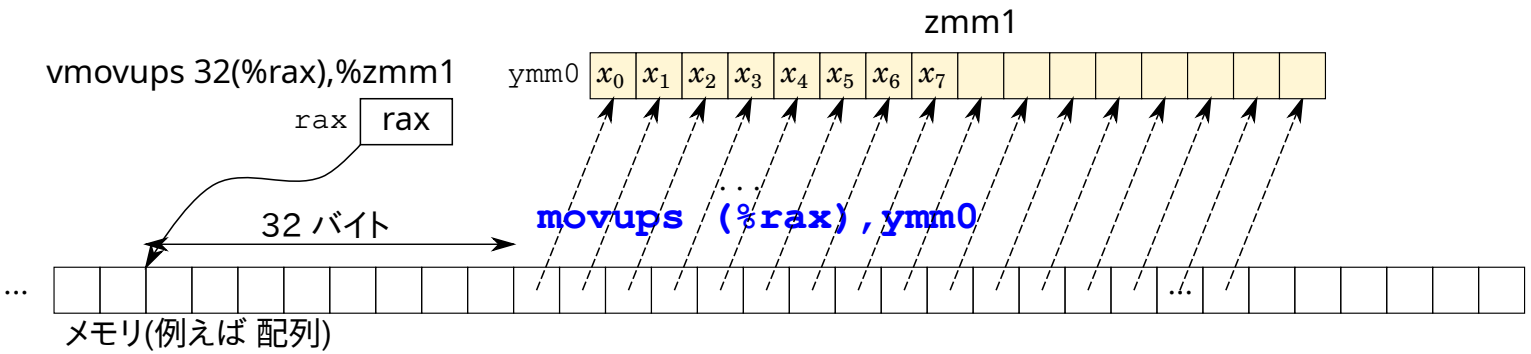
演算	表記	意味 (C 風表記で)
乗算	`vmulps %zmm0,%zmm1,%zmm2`	`zmm2 = zmm1 * zmm0`
加算	`vaddps %zmm0,%zmm1,%zmm2`	`zmm2 = zmm1 + zmm0`
乗加算	`vfmadd132ps %zmm0,%zmm1,%zmm2`	`zmm2 = zmm0*zmm2+zmm1`
ロード	`vmovups 400(%rax),%zmm0`	`zmm0 = *(rax+400)`
ストア	`vmovups %zmm0,400(%rax)`	`*(rax+400) = zmm0`

- `zmm0`, `zmm1` ... SIMD レジスタ (通常 zmm レジスタ)
  - 16 個の単精度浮動小数点数 (C の `float`), 8 個の倍精度浮動小数点数 (C の `double`) を保持できる
- `zmm0`, ..., `zmm31` まで 32 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	`vmovups 40(%rax),%zmm1`	`zmm1 = *(rax+40)`
ストア	`vmouups %zmm1,40(%rax)`	`*(rax+40) = zmm1`

- `%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
- `32(%rax)` は (`%rax` + 32) 番地を指定する記法
- `vmovups` は, 指定したアドレスから連続した 512 bit (64 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.`)
- SIMD 版も 512 bit 版 (zmm), 256 bit 版 (ymm), 128 bit 版 (xmm)

	演算対象	SIMD /スカラ?	幅 (bits)	ISA
`vmulps %zmm0,%zmm1,%zmm2`	16 SPs	SIMD	512	AVX-512
`vmulpd %zmm0,%zmm1,%zmm2`	8 DPs	SIMD	512	AVX-512
`vmulps %ymm0,%ymm1,%ymm2`	8 SPs	SIMD	256	AVX
`vmulpd %ymm0,%ymm1,%ymm2`	4 DPs	SIMD	256	AVX
`mulps %xmm0,%xmm1`	4 SPs	SIMD	128	SSE
`mulpd %xmm0,%xmm1`	2 DPs	SIMD	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, ymm$i$ の下半分)
- ymm0, ..., ymm15 : 256 bit AVX registers (ymm$i$ は, zmm$i$ 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. ベクトル型拡張
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
  - ベクトル型拡張
  - 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

OpenMP によるマルチコア CPU 並列化

- `parallel` pragma
- Work sharing 構文
- データ共有

OpenMP による GPU 並列化

CPU の性能

SIMD 命令

- SIMD 命令
- ベクトル型拡張
- ベクタ intrinsics 関数

ベクトル型

- 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(64),aligned(sizeof(float))));
```

- `floatv` という新しい型が定義され, `float` を 16 つ (64 バイト分) 並べたもの, という意味になる
- GCC, Clang, NVIDIA HPC SDK, Intel C Compiler (ICC) などで使える
- 通常の四則演算がベクトル型に対しても可能 (そして, 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;
```

- 多くのコンパイラはスカラーとベクトルの混合も許す

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

ベクトル化する上での有用オプション

- `-fast`：ベクトル化を含めた最適化
- `-Minfo`：ベクトル化されたループを報告
- `-Mneginfo`：ベクトル化できなかったループを報告
- だがベクトル化が首尾よく行ったかどうかを確かめる方法はアセンブリコードを見ること

生成されたアセンブリコードを見る

- NVIDIA コンパイラは
  - `-Mkeepasm` で, 生成されたアセンブリコード (`.s`) を残す
- 使い方
  - 小さな関数を書いて生成されたコードを見る (コンパイラの能力を探る)

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

  - ベクトル化してほしい部分 (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");
```

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,...,150 };
```

2. 各要素を配列のようにセット

```
floatv uniform(float u) {
  floatv v;
  for (i = 0; i < 16; i++) {
    v[i] = u;
  }
  return v;
}
```

うまくすると全体が効率的な SIMD 命令で実行される

3. 16 要素を明示的に指定する intrinsics 関数

```
floatv v = _mm512_set_ps(0,10,20,...,150);/* これで v = 0,10,20,...,150 */
```

方法 1 と異なり変数初期化以外の場所でも使える.

4. スカラ一個から全要素同一のベクトルを作る intrinsics 関数

```
floatv v = _mm512_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 16 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+15]} */
```

5. 同じことをする intrinsics 関数

```
1 float * a;
2       ...
3 floatv v = _mm512_loadu_ps(&a[i]);
4 /* これで v = {a[i],a[i+1], ...,a[i+15]} */
```

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

- 配列のように要素をアクセス

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

- スカラ型配列の連続 16 要素にストアする

```
1 float * a;
2 floatv v;
3       ...
4 *((floatv *)(&a[i])) = v;
5 /* これで a[i],a[i+1], ...,a[i+15] <- {v[0],v[1],...,v[15]} */
```

- 同じことをする intrinsics 関数

```
1 float * a;
2       ...
3 _mm512_storeu_ps(&a[i], v);
4 /* これで a[i],a[i+1], ...,a[i+15] <- {v[0],v[1],...,v[15]} */
```

ロードマップ

計算科学で代表的なワークロード

OpenMP によるマルチコア CPU 並列化

- `parallel` pragma
- Work sharing 構文
- データ共有

OpenMP による GPU 並列化

CPU の性能

SIMD 命令

- SIMD 命令
- ベクトル型拡張
- ベクタ intrinsics 関数

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 (名前の慣習)

- ベクタ型:
  - `__m128` (128 bit 浮動小数点数),
  - `__m256` (256 bit 浮動小数点数),
  - `__m512` (512 bit 浮動小数点数),
  - ...
- 関数群
  - `_mm_xxx` (128 bit),
  - `_mm256_xxx` (256 bit),
  - `_mm512_xxx` (512 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.
  - `_mm512_loadu_ps`
  - `_mm512_storeu_ps`
  - `_mm512_add_ps`, `_mm512_mul_ps`, `_mm512_fmadd_ps`, ...
- もっとも四則演算は intrinsics を使うまでもない

まとめ

- GPU の最大性能 = コア内の多数のスレッドの並列性 × マルチコア並列性
- CPU の最大性能 = 1 スレッド内の SIMD 命令利用 × 1 スレッド内の複数命令同時実行 (ILP) × 1 コア内の少数のスレッドの並列性 × マルチコア並列性
- 現状 それぞれのマスターが必要:
  - SIMD : ベクトル型, intrinsics
  - ILP : 意識しなくてもある程度は勝手に. 時として意識必要
  - CPU マルチコア : OpenMP `#pragma omp parallel` (+ `for`)
  - GPU マルチコア : OpenMP `#pragma omp target teams` (+ `distribute`)
  - マルチノード : MPI
- 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

課題

- ベースとなるコード

  `https://gitlab.eidos.ic.i.u-tokyo.ac.jp/tau/cs-intro-taura`
- Oakbridge CX 上で 行列積 (02mm), N 体問題 (03nbody) の高速化 (SIMD 化, マルチコア並列化, 命令レベル並列性の向上など) に取り組んでください
- 詳細は以下を参照
  - 02mm/README.md
  - 03nbody/README.md
- 提出物:
  - コード: Oakbrdge CX 上で `/work/gt11/ユーザ名/submit/cs-intro-taura` というフォルダにすべてのコードを置く
  - レポート: ITC-LMS に提出. 内容:
    - 上記のフォルダを明示すること
    - 取り組んだ内容, コードの変更の説明
    - SIMD 化されていることの確認
    - 性能測定結果
    - 性能の分析 (特に, CPU の限界性能との関係)
    - など