# ZettaScaler/PEZY-SCの紹介と今後の方向性

～自動チューニング技術の現状と応用に関するシンポジウム発表資料

2016/12/26

PEZY Computing, K.K.

# PEZYグループ

創業：2010年1月
社員数：23名

## 株式会社PEZY Computing
（ペジーコンピューティング）

- 独自メニーコア・プロセッサ開発
- 同汎用PCIeボード開発
- 同独自システムボード開発
- 同アプリケーション開発
- 半導体2.5次元実装技術開発
- ウェハ極薄化応用技術開発

創業：2014年4月
社員数：13名

## 株式会社ExaScaler
（エクサスケーラー）

- 液浸冷却技術開発
- HPC液浸システム開発
- 液浸スパコンシステム開発
- 液浸冷却水槽販売
- 液浸冷却システム販売
- 液浸冷却用ボード類販売

## UltraMemory株式会社
（ウルトラメモリ）

- 超広帯域独自DRAM開発
- DRAM積層技術開発
- 磁界結合メモリIF開発
- ウェハ極薄化応用技術開発
- 広帯域、高速DRAM開発
- 最先端汎用DRAM受託開発

創業：2013年11月
社員数：41名

PEZY Computing：メニーコア・プロセッサ
UltraMemory：超広帯域積層カスタムDRAM
ExaScaler：液浸冷却システム

- 組み合わせることにより、最終システムとしてスーパーコンピュータを開発
- 各社の要素技術を個別に製品展開

# 主な内容

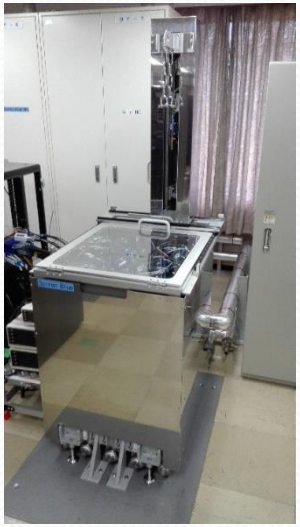
- ZettaScaler1.x/PEZY-SCの概要
- プログラミング概要
- 今後の展開
- その他の話題

# ZettaScaler1.x/ PEZY-SCの概要

# ZettaScaler-1.xシステム


Suiren（睡蓮）　ZettaScaler-1.5
2014.10Install 2016.5 Upgrade
(32node to 48node)


Shoubu（菖蒲） ZettaScaler-1.6
2015.6Install 2016.5 Upgrade



Suiren Blue（青睡蓮） Ze
2015.5 Install
2016.5 upgrade

Ajisai（紫陽花）
ZettaScaler-1.6
2015.10Install 2016.5Upgrade


Satsuki（皐月） ZettaScaler-1.6
2016.5 Install


Sakura（さくら）
ZettaScaler-1.6
2016.5 Install


- 世界で最も高速なコンピュータシステムの上位500位までを定期的にランク付けし、評価するプロジェクト。1993年に発足し、スーパーコンピュータのリストの更新を年2回発表
  単位：FLOPS（Floating Operation per Second）
  浮動小数点命令を1秒間に何回実行するか？
- ハイパフォーマンスコンピューティング(HPC)における傾向を追跡・分析するための基準を提供することを目的とし、LINPACKと呼ばれる行列計算ベンチマークによりランク付けを行っている。
- 大規模システムを構築するには資金力も必要


- 世界で最もエネルギー消費効率の良いスーパーコンピュータを定期的にランク付けし評価するプロジェクト
- 単位：FLOPS/W　FLOPSあたりの消費電力。
- スーパーコンピュータにおけるグリーンITの指標の1つともされ、日本では「スーパーコンピュータの省エネ性能ランキング」などと呼ばれる事もある。
- 省電力、高効率実行の技術力が問われる

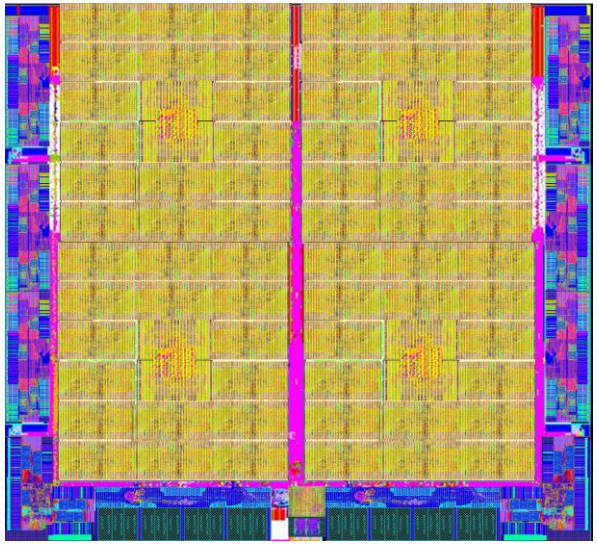
# Green500での成果

理研情報基盤センターに設置した菖蒲が
2015/6, 11, 2016/6の3期連続で首位

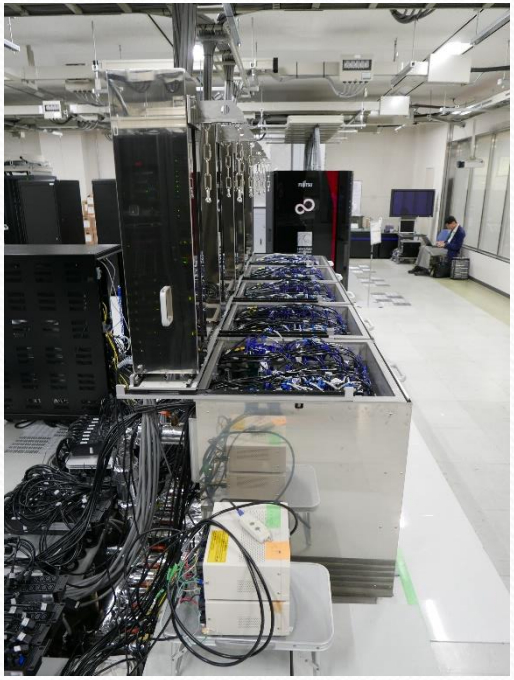
皐月も2位を獲得

# ZettaScaler-1.xのキーテクノロジー

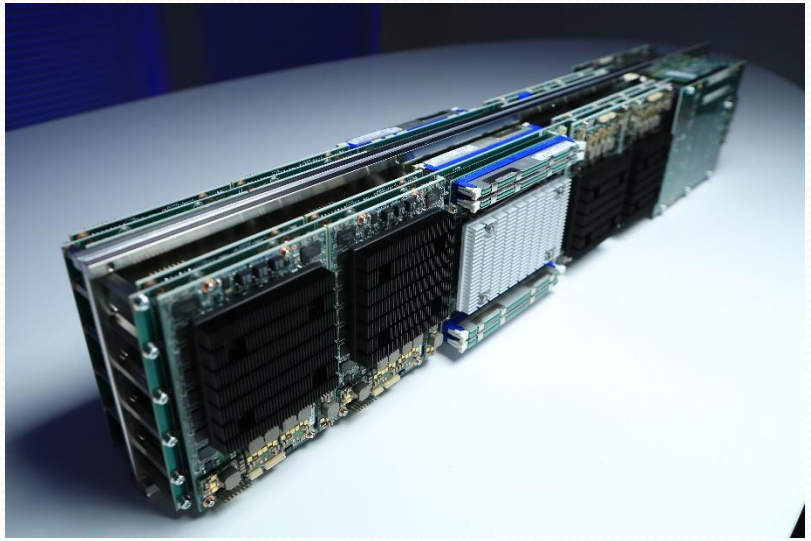
1,024メニーコアプロセッサ, “PEZY-SC”



液浸冷却技術“ESLiC”

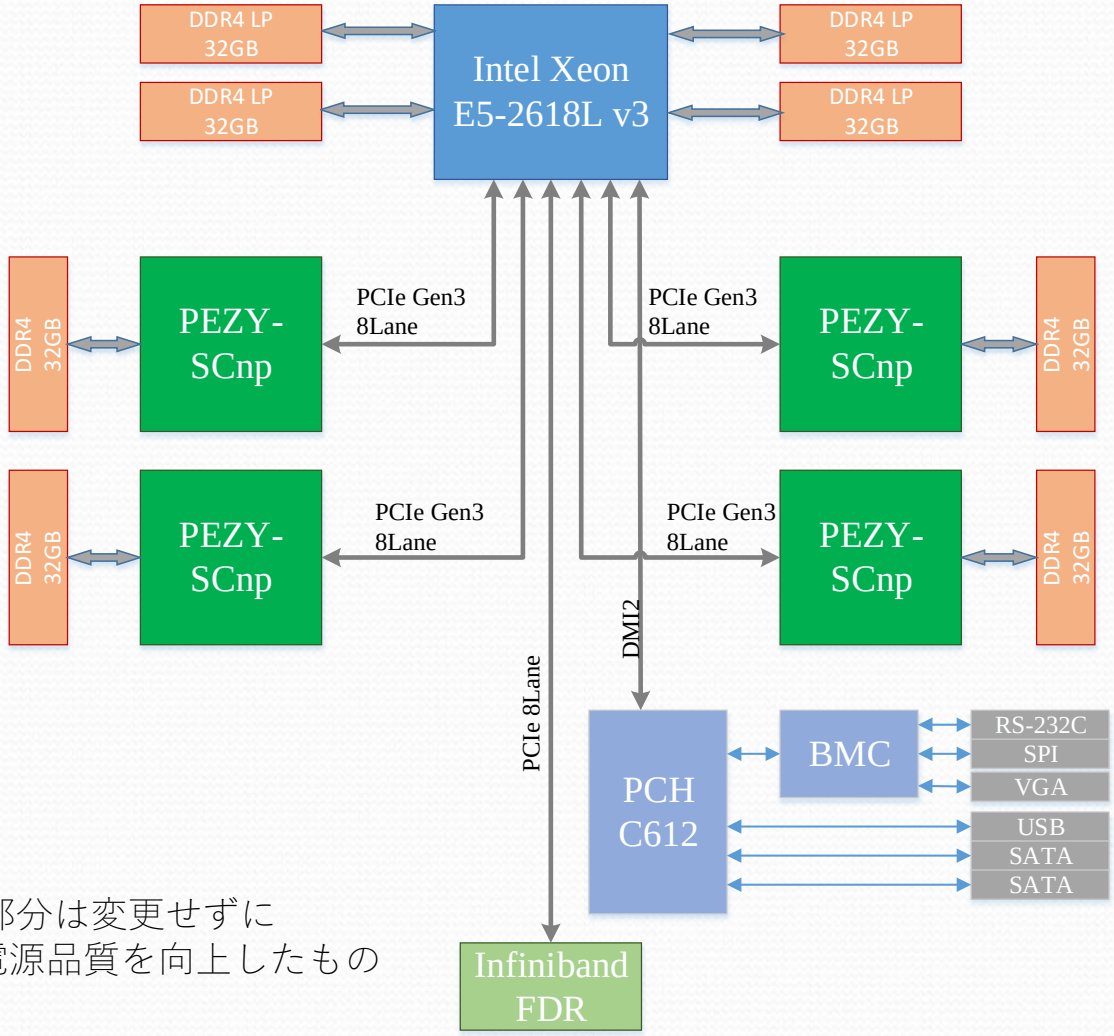


液浸サーバー“Brick”高密度実装技術



# 菖蒲ZettaScaler-1.6システム

- ノード：1つのXeonに4個のPEZY-SCnpが接続されている




PEZY-SCnpはPEZY-SCの半導体部分は変更せずに
パッケージとしての信号品質と電源品質を向上したもの

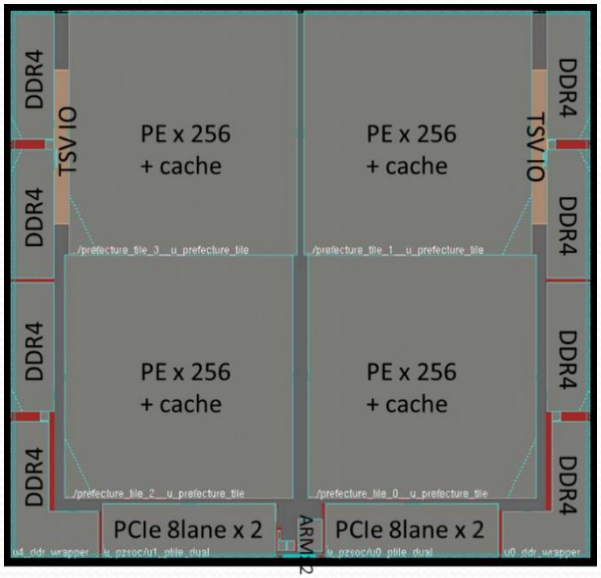
# 菖蒲ZettaScaler-1.6システム

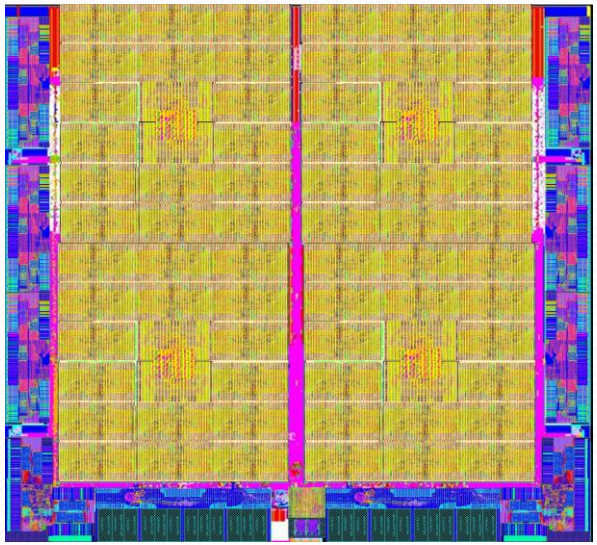
- ブリック：4ノードの集合体

- 液浸層：16ブリックから構成
- 全体システム:5液浸層から構成

# 第2世代プロセッサ「PEZY-SC」

| Name | | PEZY-SC |
|---|---|---|
| 製造プロセス | | TSMC28HPM |
| コア性能 | 動作周波数 | 733MHz(Target) |
| | キャッシュ | L1: 1MB, L2: 4MB, L3: 8MB |
| 周辺回路 | 動作周波数 | 66MHz |
| IPs | 内蔵CPU | ARM926 x 2<br>Cache L1:32KB*2, L2: 64KB |
| | PCIe | PCIe Gen3 x 8Lane 4Port<br>(8GB/s x 4 = 32GB/s) |
| | DDR | DDR4 64bit 2,400MHz 8Port<br>(19.2GB/s x 8 = 153.6GB/s) |
| コア（PE）数 | | 1,024 PE |
| 演算性能 | | 3.0T Flops (単精度浮動小数点)<br>1.5T Flops (倍精度浮動小数点) |
| 消費電力 | | 70W (Leak: 10W, Dynamic: 60W)<br>46W@533MHz (PEZY-1以下) |
| パッケージ | DDR版 | 47.5*47.5mm (2,112pin) |
| | Wide-IO版 | 20*60mm CSP (#pin: TBD) |

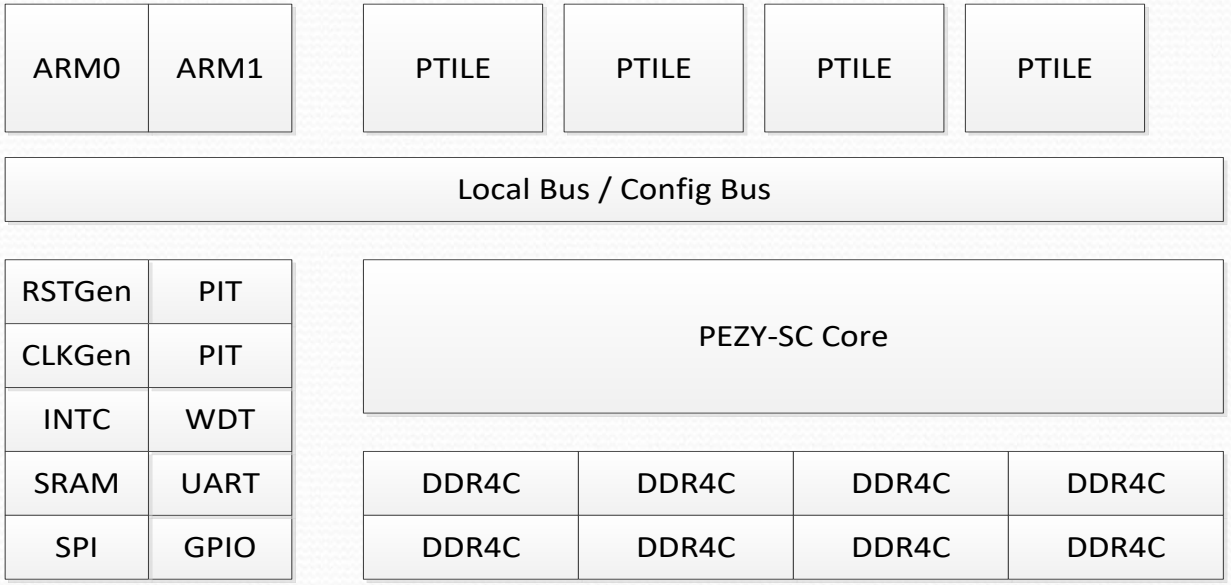


# PEZY-SCの特徴

- 高性能
  - 8スレッドSMT（Simultaneous Multi-threading）
    - 4スレッドを順番に切り替え　x 2面
    - 8スレッド分のレジスタファイルを用意
  - Deep pipelining (16Stages)
  - 潤沢なオンチップキャッシュ、メモリ
- 低消費電力、高密度実装
  - 極端に高い周波数は狙わない
  - 各PEはシンプルに
    - In-order 2way SuperScaler
    - 分岐予測なし
    - キャッシュ間コンシステンシはソフトウェア責任
    - 独自ISAによる必要命令の絞り込み

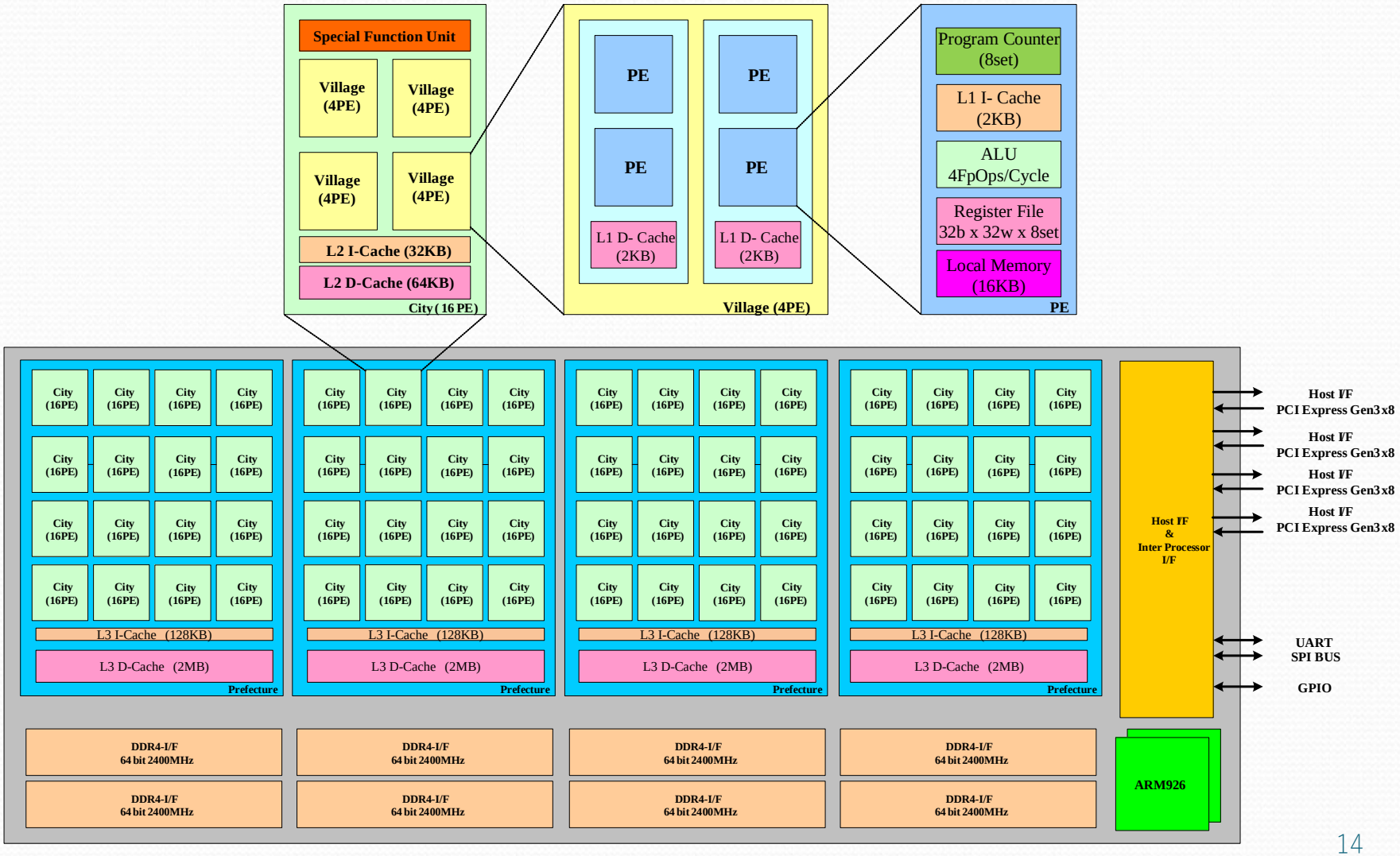
# PEZY-SCのブロック図




- PTILE: PCIe Gen3
- ARM926は2個搭載　役割分担に対する制約はない
  - L1I 16KB / L1D 16KB / L2 32KB
  - ITCM 16KB / DTCM 16KB
  - MMU

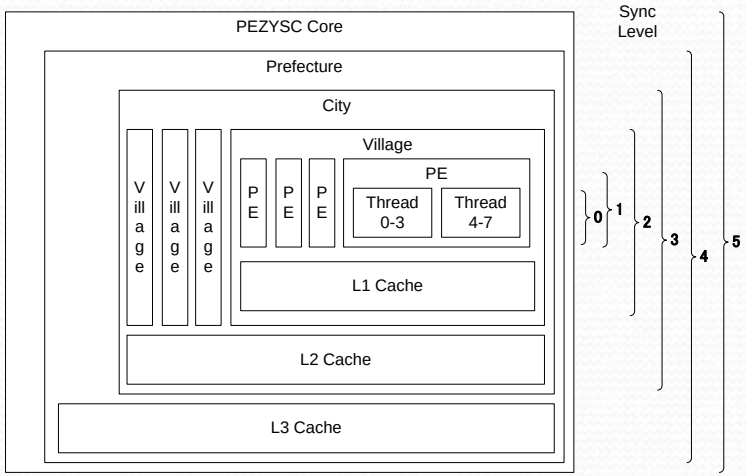
# PEZY-SCプロセッサ全体構成

3レイヤーの階層構造を持ったMIMD型メニコアプロセッサ
(4PE x 4(village) x 16(city) x 4(prefecture) = 1024PE)



# 階層構造と同期メカニズム

- スレッドを階層管理
- 同期レベル（バリア同期）
  - Level 0 :スレッドレベル、 PE内の0-3スレッド、または4-7スレッド
  - Level 1 : PEレベル、PE内の8スレッド
  - Level 2 : Villageレベル、4つのPEとL1キャッシュ
  - Level 3 : Cityレベル、16のPEとL1/L2キャッシュまで
  - Level 4 : Prefectureレベル、256のPEとL1/L2/L3キャッシュまで
  - Level 5 : PEZY-SCレベル、1024のPEとL1/L2/L3キャッシュまで



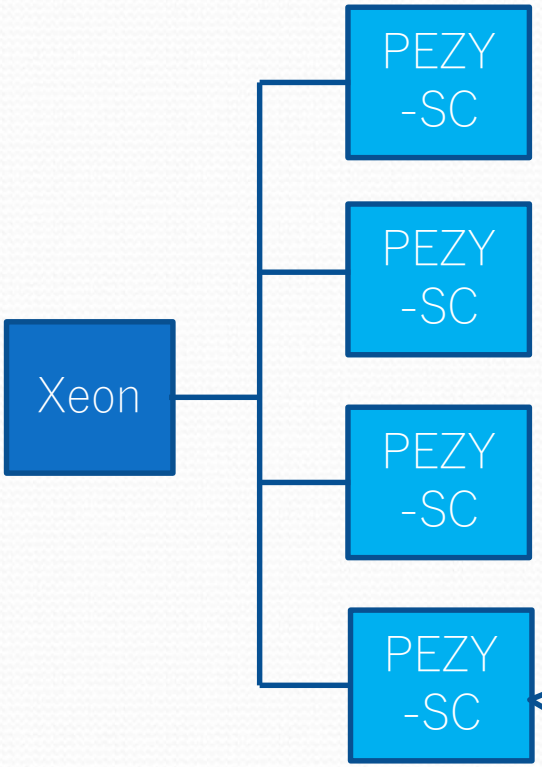
# オンチップキャッシュ

| | level | Size(B) | Chip Total(B) | Way | Entry | Line 長(B) | 接続 |
|---|---|---|---|---|---|---|---|
| データ<br>キャッシュ | L1 | 2K | 1M | 8 | 4 | 64 | 2PEに1つ |
| | L2 | 64k | 4M | 8 | 32 | 256 | Ciry毎<br>L1 8 個に対して |
| | L3 | 2M | 8M | 8 | 256 | 1k | Prefecture毎<br>L2 16 個に対して |
| 命令<br>キャッシュ | L1 | 2K | 2M | 8 | 2 | 128 | PE毎 |
| | L2 | 32K | 2M | 4 | 32 | 256 | City毎<br>PE 16個 |
| | L3 | 128K | 512K | 4 | 32 | 1K | Prefecture毎<br>L2 16 個 |

複数PE間のメモリコンシステンシはソフトウェア責任、
PE毎に16KBのローカルメモリを備える

# プログラミング概要

# プログラミング対象



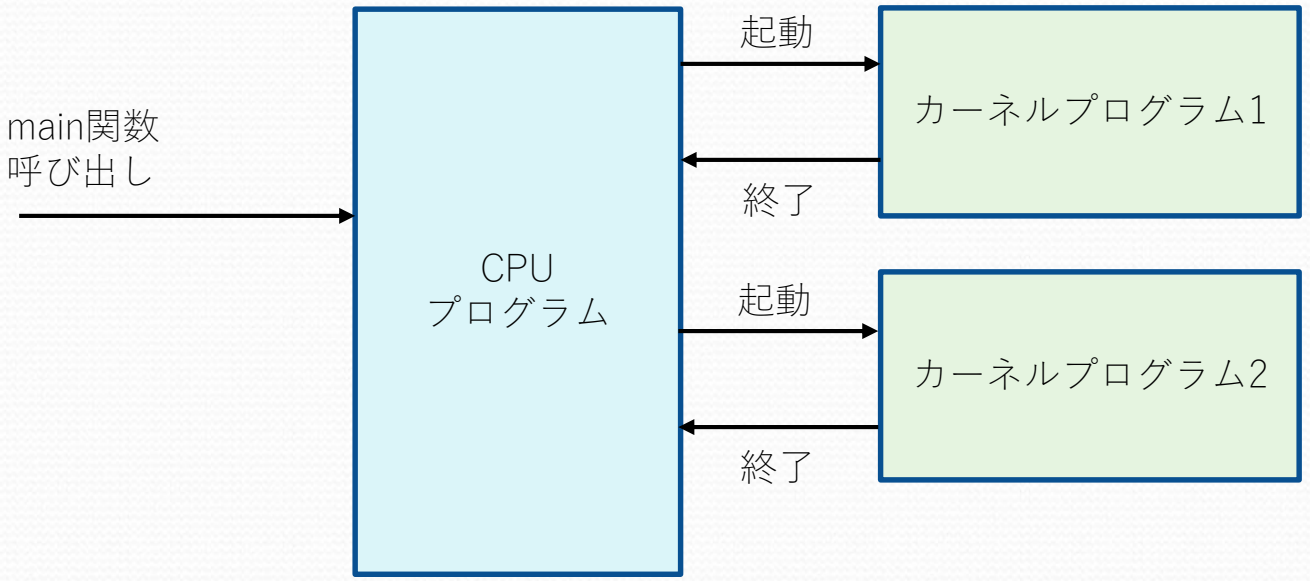

<演算リソース>
・1024個の演算コア(PE)
・1PEあたり8個のスレッド

<メモリ>
・32GBのデバイスメモリ
・1PEあたり16KBのローカルメモリ

# 作成するプログラム

- 2種類のプログラムを作成する必要がある
  - CPU上のプログラム（C++で記述）
  - PEZY-SC上のカーネルプログラム(PZCLで記述)

※PZCL=カーネルプログラムを記述するPEZY独自仕様の言語
コンパイラはllvmを用いている。




上図のようにCPUプログラムからカーネルプログラムを起動する

# 特殊な関数

- カーネルプログラムで利用可能な、PEZY-SC制御に必要な組み込み関数がある。

  - sync_L1（L1キャッシュにアクセスする単位でのスレッド同期）
  - sync_L2（L2キャッシュにアクセスする単位でのスレッド同期）
  - sync_L3（L3キャッシュにアクセスする単位でのスレッド同期）
  - sync　　（sync_L3と同等）

  - flush_L1（L1キャッシュのフラッシュ）
  - flush_L2（L2キャッシュのフラッシュ）
  - flush_L3（L3キャッシュのフラッシュ）
  - flush　　（flush_L3と同等）

  - get_pid (PE ID取得)
  - get_tid (PE内スレッドID取得)
  - chgthread (PE内スレッドの表裏切り替え)

# カーネルプログラムの構造

- 基本的な構造

```
void pzc_foo(…)
{
    • PE ID取得(get_pid)
    • PE内スレッドID取得(get_tid)

    • 自スレッドに割り当てられた処理の実行

    • 出力バッファフラッシュ(flush)
}
```

# pzcAddサンプル

- カーネルは起動するとユニークな tid,pid を持って、 CPUから指定されたスレッド分実行される。

tid=0,pid=0

```
void pzc_Add(float* a, float* b, float* c, int count)
{
      int tid   = get_tid();
      int pid   = get_pid();
      int index = pid * get_maxtid() + tid;

      if(index >= count)    return;

      c[index] = a[index] + b[index];

      flush(); // cache flush
}
```

tid=1,pid=0

```
void pzc_Add(float* a, float* b, float* c, int count)
{
      int tid   = get_tid();
      int pid   = get_pid();
      int index = pid * get_maxtid() + tid;

      if(index >= count)    return;

      c[index] = a[index] + b[index];

      flush(); // cache flush
}
```

…

tid=7,pid=N

```
void pzc_Add(float* a, float* b, float* c, int count)
{
      int tid   = get_tid();
      int pid   = get_pid();
      int index = pid * get_maxtid() + tid;

      if(index >= count)    return;

      c[index] = a[index] + b[index];

      flush(); // cache flush
}
```

- １つのPEには8スレッドが存在する
  - スレッド数を128で起動した場合、128/8=16個のPEが実行される
- 8192を超えるスレッド数で起動する場合、CPUから複数回に分けて起動される

# 簡単な最適化の説明

- 前述のpzcAddサンプルを用いて、PEZY-SC内での簡単な最適化の説明を行う

- ここでは以下のような最適化を行っている
  - カーネル呼び出しのオーバヘッドの削減
  - chgthreadを用いたレイテンシーの隠蔽
  - 同期を用いたキャッシュアクセスの効率化

# オーバヘッド削減(1/2)

- 以下のコードをスレッド数=要素数として起動する場合、8192を超えるサイズを処理しようとした場合にカーネルが複数回起動されるため、カーネル呼び出しのオーバヘッドが増加する

```
void pzc_Add(float* a, float* b, float* c, int count)
{
	int tid   = get_tid(); // thread ID (0 - 7)
	int pid   = get_pid(); // PE ID
	int index = pid * get_maxtid() + tid;

	if(index >= count) return;

	c[index] = a[index] + b[index];

	flush(); // cache flush
}
```

# オーバヘッド削減(2/2)

- 以下のようにカーネルコードを修正し、CPUからの呼び出し時のスレッド数を固定にしても、1回のカーネル呼び出しで全要素の処理を行えることとなる。

- これによってオーバヘッドを減らすことができる。

```
void pzc_Add(float* a, float* b, float* c, int count)
{
    int tid   = get_tid(); // thread ID (0 - 7)
    int pid   = get_pid(); // PE ID

    int offset  = pid * get_maxtid() + tid;
    int  step   = get_maxtid() * get_maxpid();

    for(int pos = offset; pos < count; pos += step) {
        c[pos] = a[pos] + b[pos];
    }
    flush();
}
```

# 寄り道:CPUエミュレート

- このようにカーネルの中でループさせることは別のメリットもある。
- CPUで1スレッドでの動作として、この関数を同じように動作させることができる→ソースを共有したデバッグに有効

```
void pzc_Add(float* a, float* b, float* c, int count)
{
    int tid   = get_tid(); // thread ID (0 - 7)
    int pid   = get_pid(); // PE ID

    int offset  = pid * get_maxtid() + tid;
    int  step   = get_maxtid() * get_maxpid();

    for(int pos = offset; pos < count; pos += step) {
        c[pos] = a[pos] + b[pos];
    }
    flush();
}
```

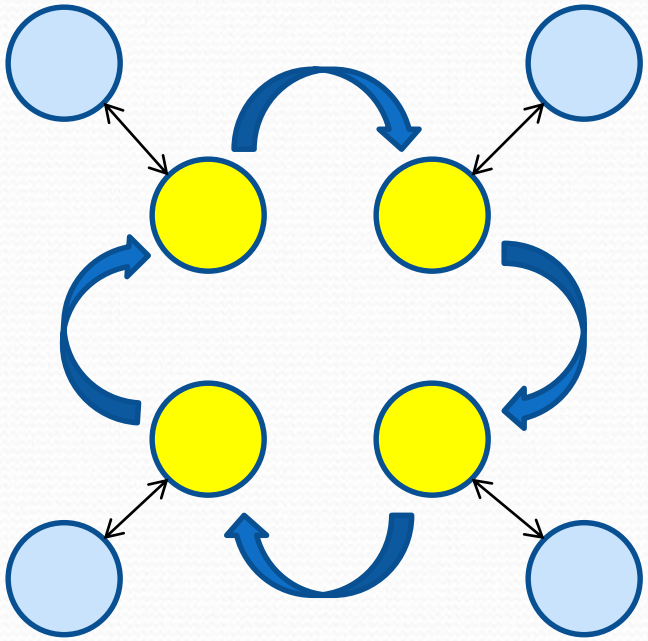
CPUでは

get_tid() … 常に0
get_pid() … 常に0
get_maxtid() … 1
get_maxpid() … 1

# スレッドの切り替え(1/3)

- １つのPEに8スレッド存在するが、一度には4スレッドのみが動作する。
  - 表裏で4スレッドずつ。

- sync/flushなどの同期やchgthreadを使用しないと、表裏が切り替わらない。



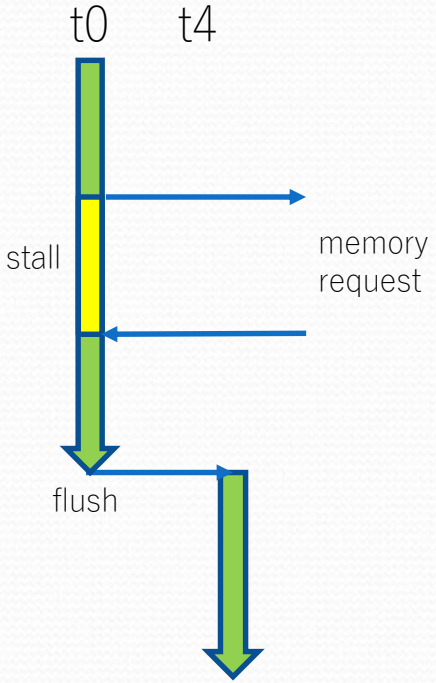
# スレッドの切り替え (2/3)

- 以下の実装では、ループの中にスレッドが切り替わる命令が無いので現在実行中の各スレッドが flushにたどり着くまで裏スレッドは処理されない。
  - アクセスのアドレスが不連続になり、キャッシュ効率が悪い
  - メモリアクセスのレイテンシーを隠蔽できない

```
void pzc_Add(float* a, float* b, float* c, int count)
{
    int tid = get_tid(); // thread ID (0 - 7)
    int pid = get_pid(); // PE ID

    int offset = pid * get_maxtid() + tid;
    int step = get_maxtid() * get_maxpid();

    for(int pos = offset; pos < count; pos += step) {
        c[pos] = a[pos] + b[pos];
    }
    flush();
}
```



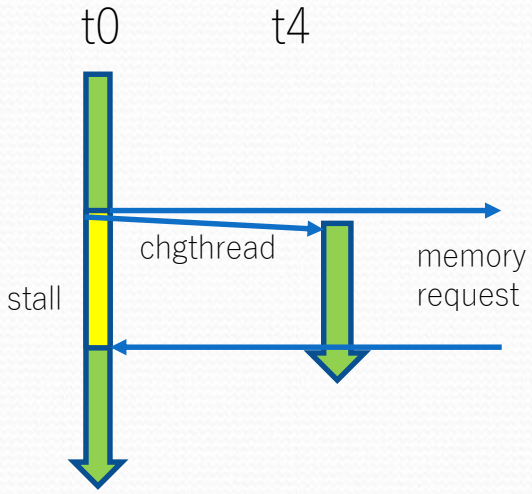
# スレッドの切り替え(3/3)

- 以下のようにa, bの読み込み後にchgthreadを入れる事で改善される。

```
void pzc_Add(float* a, float* b, float* c, int count)
{
    int tid   = get_tid(); // thread ID (0 - 7)
    int pid   = get_pid(); // PE ID

    int offset  = pid * get_maxtid() + tid;
    int  step   = get_maxtid() * get_maxpid();

    for(int pos = offset; pos < count; pos += step) {
        float a_ = a[pos];
        float b_ = b[pos];
        chgthread();
        c[pos] = a_ + b_;
    }
    flush();
}
```



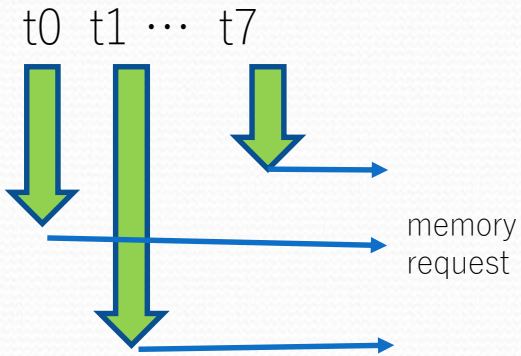
# メモリアクセスの同期(1/2)

- 以下の実装だと、各スレッドがメモリレイテンシーの状況によって進行度がばらばらになり、キャッシュアクセスが非効率となる場合がある。

```
void pzc_Add(float* a, float* b, float* c, int count)
{
    int tid = get_tid(); // thread ID (0 - 7)
    int pid = get_pid(); // PE ID

    int offset = pid * get_maxtid() + tid;
    int step = get_maxtid() * get_maxpid();

    for(int pos = offset; pos < count; pos += step) {
        float a_ = a[pos];
        float b_ = b[pos];
        chgthread();
        c[pos] = a_ + b_;
    }
    flush();
}
```



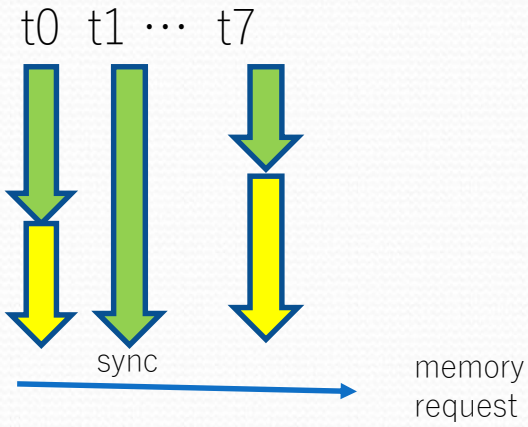
# メモリアクセスの同期(2/2)

- 以下のようにメモリアクセス前に同期を入れることにより、メモリアクセス性能が向上する場合がある
  ただし同期自体のペナルティがあるため、利用する／しない、あるいは同期レベルの選択に注意が必要

```
void pzc_Add(float* a, float* b, float* c, int count)
{
    int tid = get_tid(); // thread ID (0 - 7)
    int pid = get_pid(); // PE ID

    int offset = pid * get_maxtid() + tid;
    int step = get_maxtid() * get_maxpid();

    for(int pos = offset; pos < count; pos += step) {
        sync_L2();
        float a_ = a[pos];
        float b_ = b[pos];
        chgthread();
        c[pos] = a_ + b_;
    }
    flush();
}
```

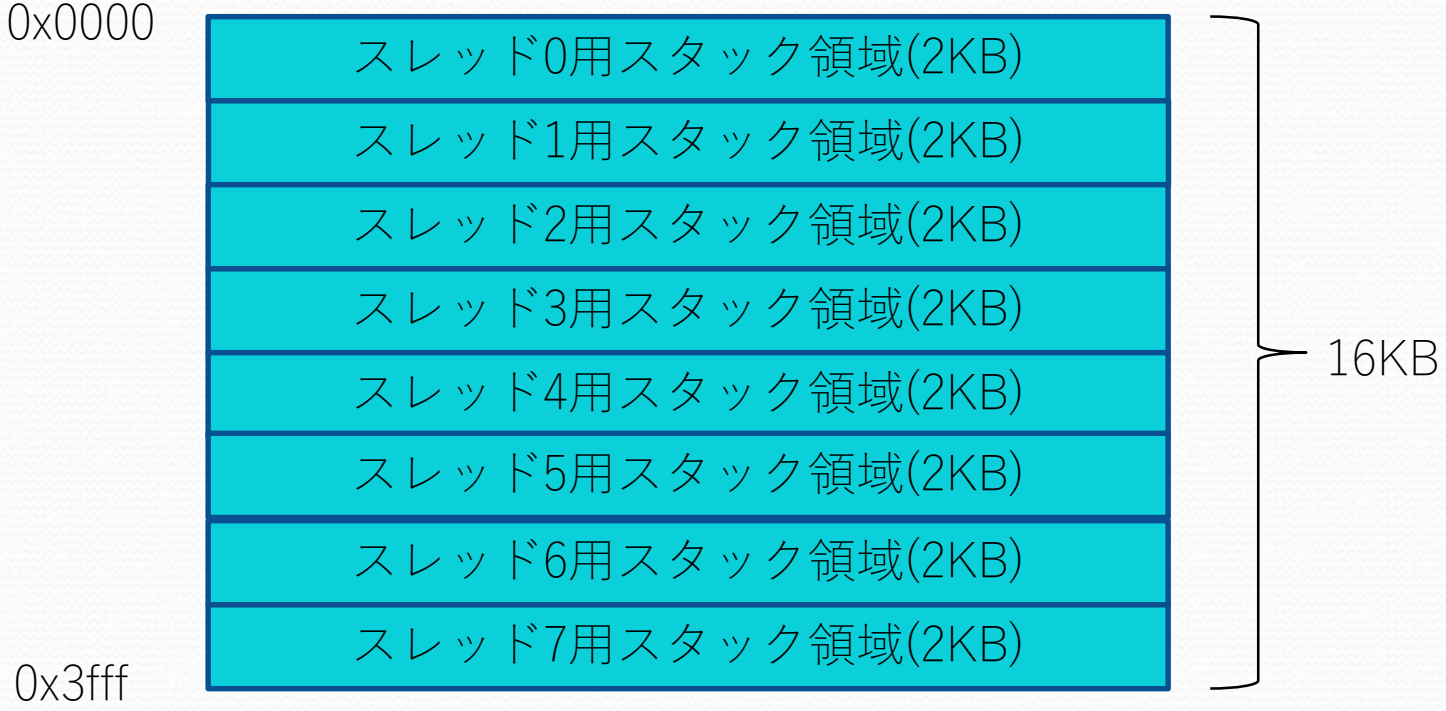


# PEZY-SCの効果的な利用

- スレッド、PE単位の並列性を活かす
- L1~L3キャッシュに優しいメモリ配置を行う
- CPUからカーネルの起動回数を減らす
- chgthread を用いてレイテンシーを隠蔽する
- 同期を適切に用いて、キャッシュの効率を上げる
- ローカルメモリを利用することでメモリアクセスを減らす
- その他各種設定（メモリ書き出し設定・カーネル呼び出し方法設定）→これについては今後必要に応じて情報公開します。

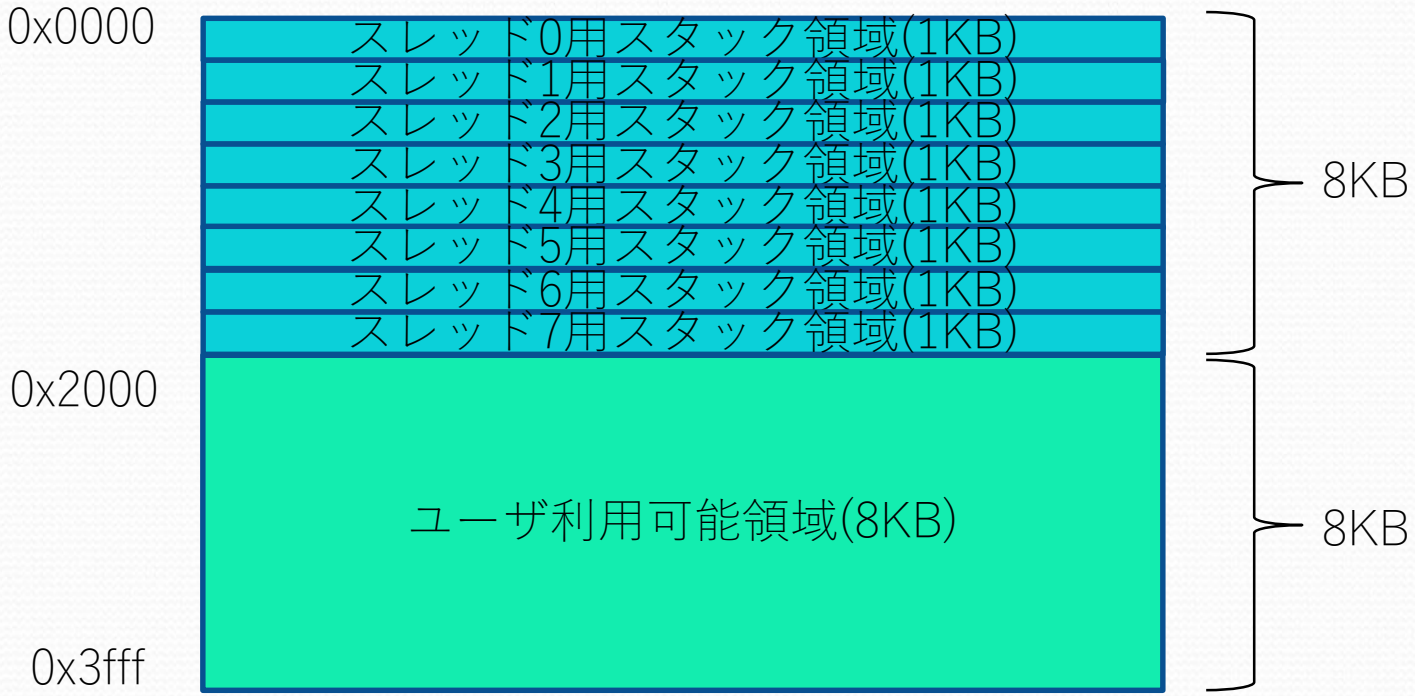
# ローカルメモリの利用(1/2)

- PE毎に16KBのローカルメモリをカーネルプログラムで利用できる
- デフォルトではPE内の8スレッドのスタック領域として、2KBずつを割り振られている



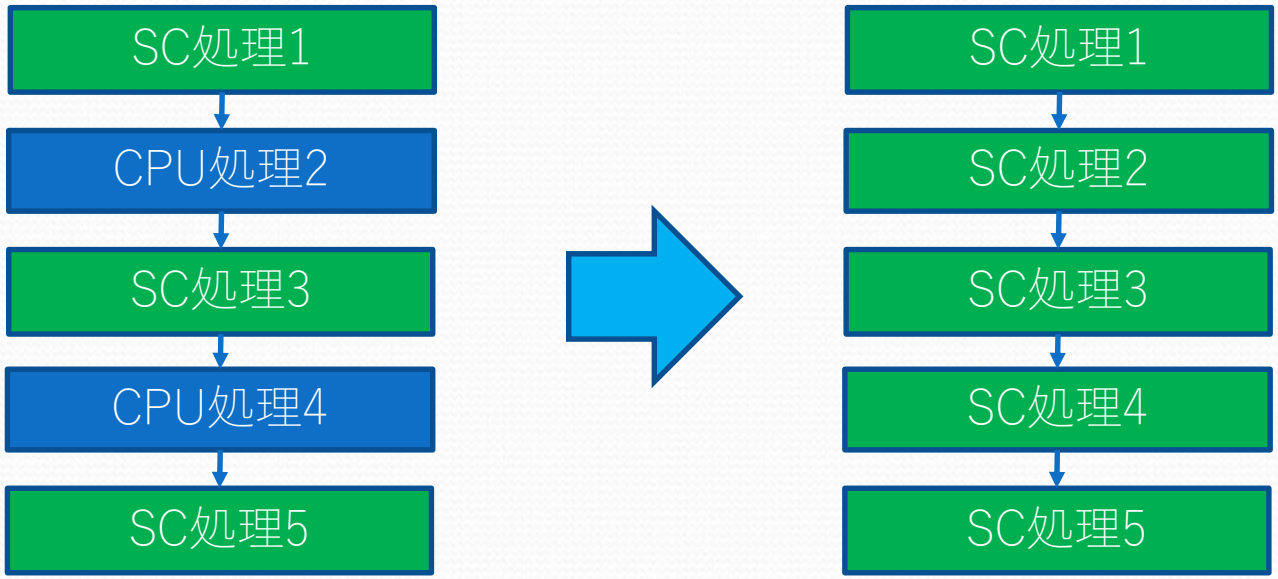
# ローカルメモリの利用(2/2)

- このままではユーザが利用できないため、スレッド用のスタック領域を削減する（下図はスレッド毎のスタックサイズを1KBとした場合）



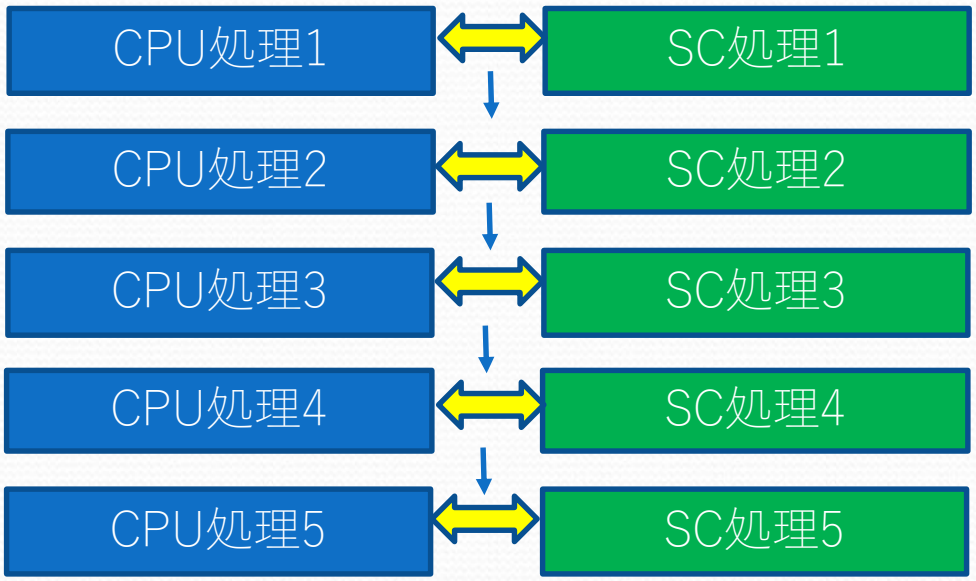
# プログラミングのパターン

- PEZY-SCのカーネルプログラムはなるべく全処理を一括で持っていきたい
  - MIMDでプログラミングに自由度があるので、多少並列度が落ちるところもとりあえずカーネルには載せることは容易



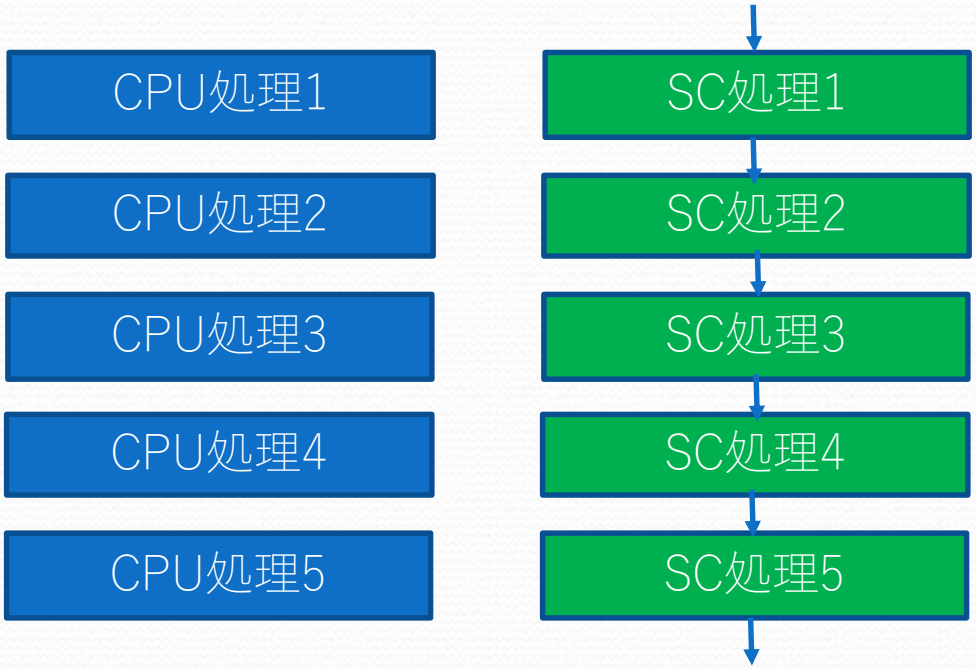
# プログラミングのパターン

- フロントエンドがclangであり、ほとんどのケースではSCとCPUでのソースコードの共有が容易。
- デバッグ時には細かい単位で切り替えながら不具合を特定することが非常に有効



# プログラミングのパターン

- 最終的な実行はなるべくカーネル処理だけとする



# その他の話題

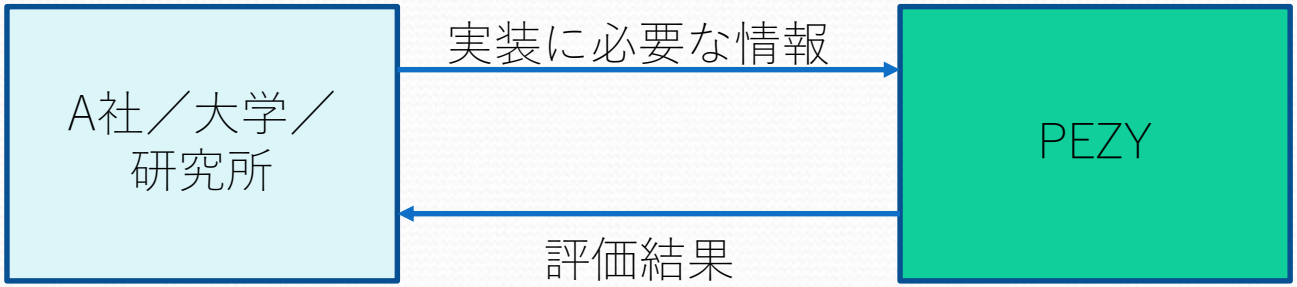
# 共同開発のパターン

- そもそもPEZY-SCは利用できそうだろうか？
- 自分のところで評価するのは負荷が高い。。。

# 共同開発のパターン1

- そもそもPEZY-SCは利用できそうだろうか？
- 自分のところで評価するのは負荷が高い。。。

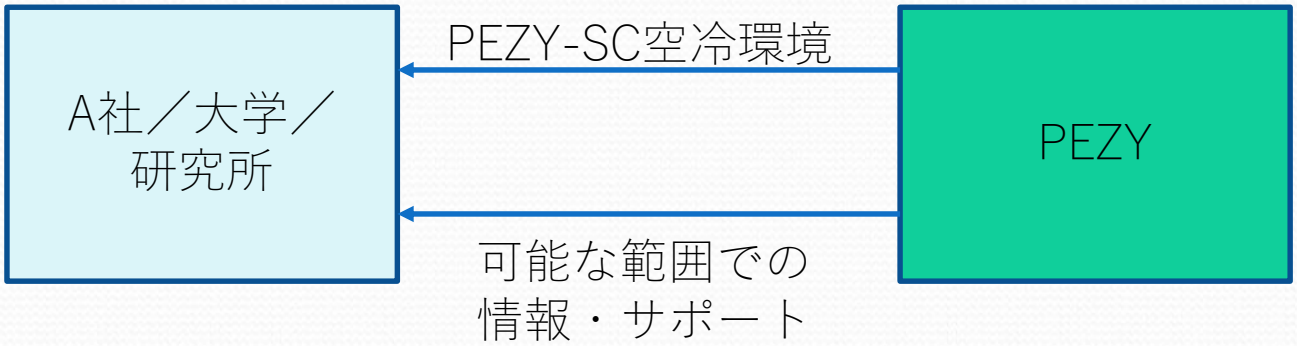
→（可能な範囲で）実装に必要な情報をご提供頂き、PEZY側で（可能な範囲で）評価を行う（基本はNDAベース）



# 共同開発のパターン2

- そもそもPEZY-SCは利用できそうだろうか？

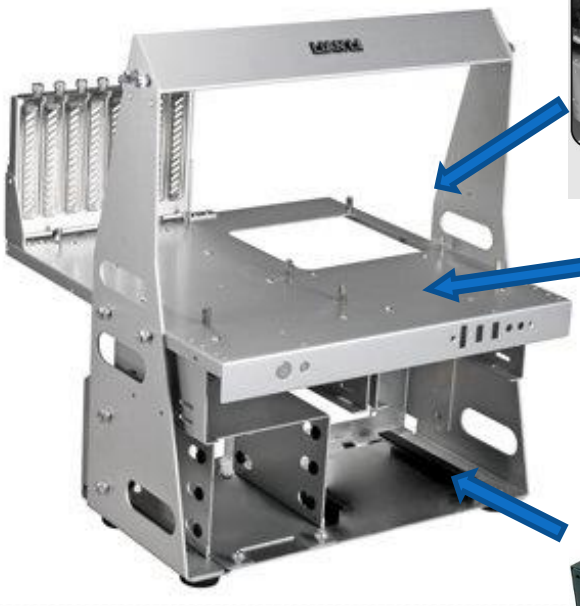
- まずは簡単に触ってみたい。。。
- PEZY-SCを空冷環境下でご提供
  - ただし、開発途上のものなので十分な情報やサポートを保証できるものではありません。
    （弊社側で可能な範囲でのご提供となります）



# PEZY-SC評価システム例

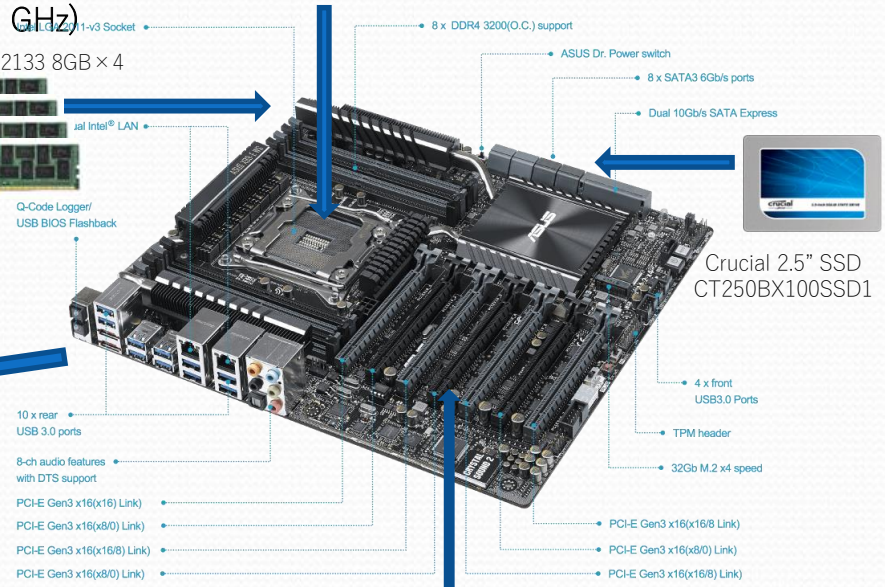
Intel® Xeon® Processor E5-2650 v3 (25M Cache, 2.30 GHz)

Samsung DDR4-2133 8GB × 4

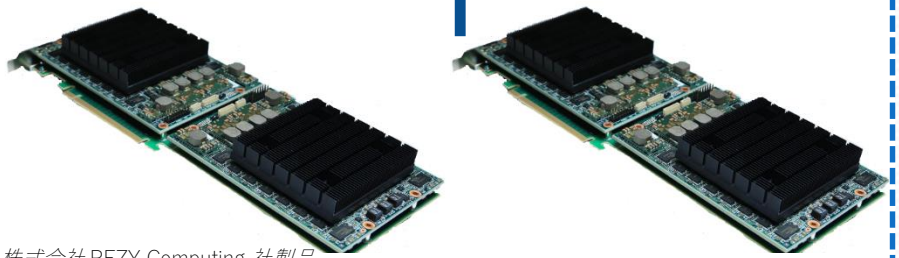
LIANLI ATX PC-T60A

RA-750S




Crucial 2.5” SSD
CT250BX100SSD1

ASUS X99E WS



*株式会社 PEZY Computing 社製品*

PEZY-SC Dual Board x 2

# 共同開発のパターン3

- そもそもPEZY-SCは利用できそうだろうか？

- 液浸環境下でスパコン構成を試してみたい。

→菖蒲システムの利用公募

http://accc.riken.jp/news1/2016-07-01/

こちらも十分な情報やサポートを保証できるものではありません（可能な範囲でのご提供となります）

研究開発用途で開発情報を公開可能ならばお勧め！

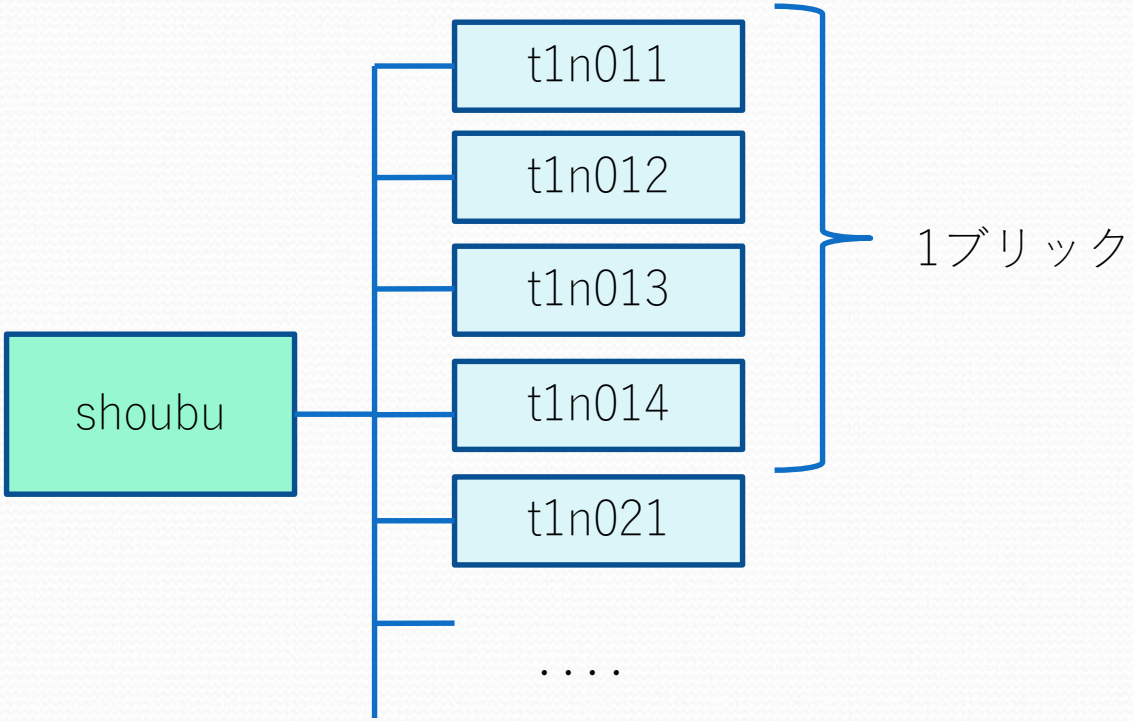
個人でも応募可！！

# 菖蒲システムでできること

- 複数のコンピュートノードを用いた大規模な並列計算が可能。MPIの利用が可能。
- 現状は1タンク=16ブリック=256ノードが開発者に常時提供されている。必要に応じて全システムでの利用も可能。
- ジョブ管理システムslurmの利用が可能。
- フロントエンド、コンピュートノードともにlinux(centOS7)が入っており、一般的なlinuxのライブラリやツールが利用可能。

# 菖蒲の構成

- フロントエンドとコンピュートノードから構成される。
- 4つのコンピュートノードは1つのブリックを構成する。
- また、各コンピュートノードはそれぞれ1個のXeonと4個のPEZY-SCを所持する。
- フロントエンド、コンピュートノードはInfinibandにより結合されている。



# ジョブ管理システムの利用

- 複数の人が菖蒲システムを利用するためにジョブシステム(slurm)が導入されている。これにより特定のコンピュートノードを意識せずに利用ができる。

  - ssh shoubu.riken.jp
    のようにしてフロントエンドにログインする。
  - フロントエンド上でプログラムの編集、ビルドを行う。

  - sbatch –nodes <ノード数> --ntasks-per-node <ノードあたりのMPIプロセス数> tst.sh

```
#!/bin/sh
#SBATCH –p debug
#SBATCH –exclusive

mpirun … //MPIを用いる場合
```

# 今後の展開

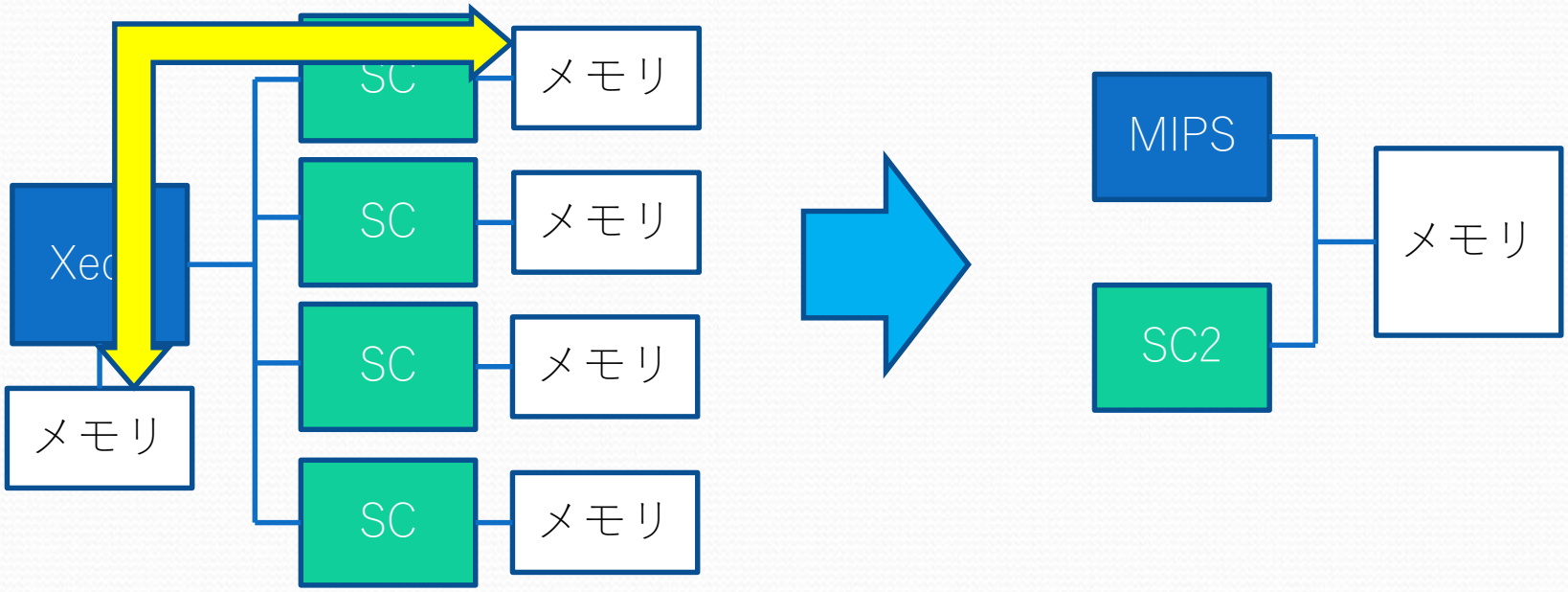
# 今後の展開

- 新プロセッサ　PEZY-SC2の開発
  - 2,048コアの演算PE＋MIPSプロセッサ内蔵
  - TCIインタフェースによる、メモリ帯域の飛躍的拡大
- Brickボード、液浸冷却システムのブラッシュアップ
  - 新ブリック構成で冷却効率を向上
- ZettaScaler-2.xシリーズ
  - これらの新規開発要素を組み合わせた、新しいスーパーコンピュータの実現

# PEZY-SC2の特徴

- CPUがMIPSとなりSC2とメモリ空間を共有する

→従来XeonとSCの間で必要であったメモリ転送が必要なくなる。



# PEZY-SC2の特徴

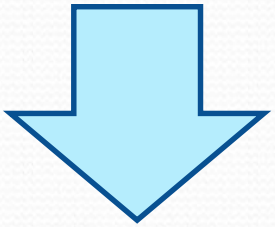
- CPUとSC2の協調動作の強化
- 各種命令セットの補強
- （大きな変更なく）SCのプログラムをそのままコンパイル・実行できる

# 外部に公開している情報

- 若干のサンプルプログラム
- Doxygenで自動生成されたAPIリファレンス
- 簡単なアーキテクチャ説明資料
- 簡単なプログラミングマニュアル

# 外部に公開している情報

- 若干のサンプルプログラム
- Doxygenで自動生成されたAPIリファレンス
- 簡単なアーキテクチャ説明資料
- 簡単なプログラミングマニュアル



- ユーザポータルを作成してここに各種情報を集約していく予定です（2017/1予定）
  - PEZYと個別にNDAのやり取りを行い、その後に参加して頂くようになります。

# 開発中/予定のソフトウェア

- 物理系シミュレーション
- 開発環境
  - OpenACC／OpenCL／PUDA(!)･･･
- 量子計算シミュレーション
- メタゲノム解析ツール
- ニューラルネット
  - Caffe／･･･
- 数値計算ライブラリ
  - BLAS／FFT･･･
- ･･･

# ご興味がありましたら

ishikawa@pezy.co.jp

お気軽にご連絡ください