OpenACCとLibsci_ACCによるアクセラレータ向けプログラムミング

寺西慶太
Cray Inc.

エクサスケール畑を耕すのは大変！

かつてシーモア・クレイは、並列計算を畑の耕作に例えて「ニワトリ512羽とウシ2頭、あなたならどちらに耕させるかね？」と尋ねられたことがあるといいます。

それから時を経て、この問いは自ら答えを導き出す結果となりました。消費電力の制約によってCPUメーカーは「ウシ」（＝パワフルなシングルコア・デバイス）から、マルチ・メニーコアな「ニワトリ」への移行を余儀なくされたからです・

"An exascale supercomputer will take this a step further, connecting tens of thousands of many-core nodes.

"Application programmers face the challenge of harnessing the power of tens of millions of threads."

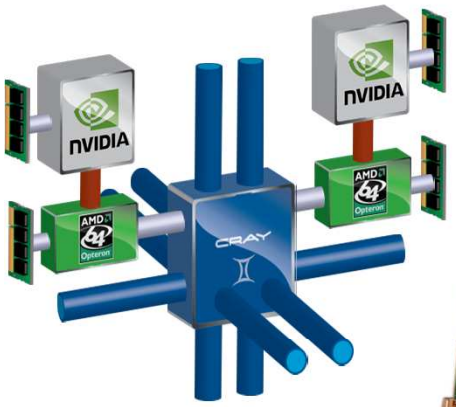
EPCC News, issue 70 (Autumn 2011)

三層の並列化

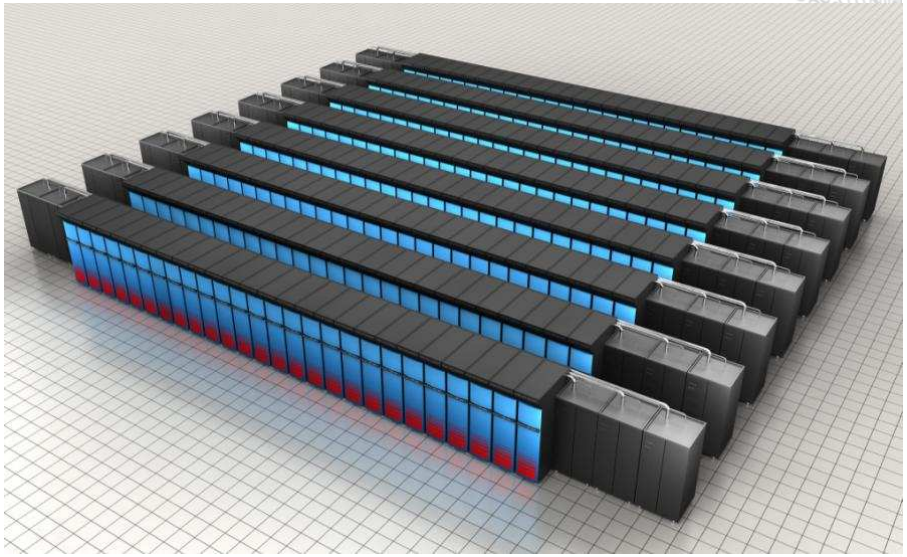
- 現状と同様、ノード間、ソケット間でのはMPIで並列化
- ノード内では、共有メモリを意識した並列化
- コード最深部のループはベクトル化による並列化
- 今後、この3つを統合的に直接管理できるプログラム言語が開発される可能性はある。しかし、それが普及するには以下の条件が必要
  - 使いやすさ
  - 従来の複雑なプログラミングによる手法と同様な性能が出せる
  - ライブラリ、デバッガ、ツールのサポート

The Cray XK6 hybrid architecture

- 2011年5月発表
- NVIDIA製、Fermi X2090 GPUを搭載
  - Keplerにアップグレード可
- AMD製Interlagos CPU
- Cray Gemini　インターコネクト
  - 高バンド幅、低レイテンシ
- 統合された、X86/GPU プログラム環境
- Cray XE6と生産ラインが完全互換
  - キャビネット、ブレード、冷却モジュール
- Cray XT/XE からアップグレード可



アクセラレータ搭載の2大ペタフロップシステム



Blue Waters: Sustained Petascale Performance

- 実科学アプリケーション向けシステム
- 235 XE キャビネット + 30 XK キャビネット
  - > 25K compute nodes
- 11.5 PFLOPS
- 1.5 Petabytes メモリ
- 25 Petabytes ストレージ
  - 1 TB/sec IOバンド幅
- Cray's scalable Linux Environment
- HPC-focused GPU/CPU Programming Environment

Titan: A "Jaguar-Size" System with GPUs

- 200 キャビネット
- 18,688 ノード
- 25x32x24 3D トーラス (22.5 TB/s global BW)
- 128 I/O ブレード (512 PCIe-2 @ 16 GB/s bidir)
- 1,278 TB メモリ
- 4,352 sq. ft.
- 10 MW

Cray のアクセラレータコンピューティングへのビジョン

- **プログラミングの複雑さの克服！**
  - **複数のプラットフォームで動く**単一のプログラミングモデルが必須
    - **ポータブルな** 表現で各レベルの並列化が実装でき
    - プログラミングモデル、最適化手法がマルチコアx86CPUとあまり変わらない
    - **ユーザは同じソースコードで各プラットフォームに合わせて実装ができる**
- **Crayは統合されたプログラミング環境を、コンパイラ、ライブラリ、ツールによって提供し、高性能なアプリケーション開発を容易にすることを目標に研究開発**
- **Crayの提供するプログラミング環境**
  - OpenACCディレクティブが実装されたFortran, C, C++ コンパイラ
    - ディレクティブによるアクセラレータプログラミングと最適化
  - Crayコンパイラと統合された性能ツールとデバッガ
    - CUDAレベルでデバグ、コード性能解析をする必要がない
  - アクセラレータ向け科学計算ライブラリ

XKシステムノード上でのプログラミング

- Fortran, C, and C++ コンパイラ
- **OpenACC ディレクティブでプログラムを記述**
  - データ転送、ポインタの受け渡し等の記述が容易
  - コンパイラがアクセラレータ、x86向け両方の最適化
  - CUDAで書かれたカーネル、関数の組み込みも可能
  - **ノード並列デバッガ** DDT、TotalViewの利用が可能
- 開発中のCray **Reveal**はコンパラが生成するソースコードの内部表記を元に性能解析、最適化の作業を支援
  - GUIでソースコードを閲覧しながらループのGPU並列化、ベクトル化等を行える
    - **スコーピングで** コードの移植、最適化を支援
    - **Crayの性能解析ツールの情報と組み合わせて、コードの最適化も可能**
- 科学計算ライブラリ
  - OpenACC、CUDAと互換
  - 従来のAPIをそのまま継承
  - Crayの自動チューニング技術

基本例題: リダクション

- 配列の総和を求める
- Fortranだと4行

```
a=0.0

do i = 1,n
 a = a + b(i)
end do
```

CUDAで書いたリダクションコード

```
__global__ void reduce0(int *g_idata, int
*g_odata)
{
extern __shared__ int sdata[];

unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x +
threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();

for(unsigned int s=1; s < blockDim.x; s *= 2) {
if ((tid % (2*s)) == 0) {
sdata[tid] += sdata[tid + s];
}
__syncthreads();
}

if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}


extern "C" void reduce0_cuda_(int *n, int *a,
int *b)
{
int *b_d,red;
const int b_size = *n;


cudaMalloc((void **) &b_d , sizeof(int)*b_size);
cudaMemcpy(b_d, b, sizeof(int)*b_size,
cudaMemcpyHostToDevice);
```

```
dim3 dimBlock(128, 1, 1);
dim3 dimGrid(2048, 1, 1);
dim3 small_dimGrid(16, 1, 1);

int smemSize = 128 * sizeof(int);
int *buffer_d, *red_d;
int *small_buffer_d;

cudaMalloc((void **) &buffer_d ,
sizeof(int)*2048);
cudaMalloc((void **) &small_buffer_d ,
sizeof(int)*16);
cudaMalloc((void **) &red_d , sizeof(int));


reduce0<<< dimGrid, dimBlock, smemSize >>>(b_d,
buffer_d);

reduce0<<< small_dimGrid, dimBlock, smemSize
>>>(buffer_d, small_buffer_d);

reduce0<<< 1, 16, smemSize >>>(small_buffer_d,
red_d);

cudaMemcpy(&red, red_d, sizeof(int),
cudaMemcpyDeviceToHost);

*a = red;

cudaFree(buffer_d);
cudaFree(small_buffer_d);
cudaFree(b_d);
}
```

更に最適化されたリダクション

```
template<class T>
struct SharedMemory
{
    __device__ inline operator       T*()
    {
        extern __shared__ int __smem[];
        return (T*)__smem;
    }

    __device__ inline operator const T*() const
    {
        extern __shared__ int __smem[];
        return (T*)__smem;
    }
};

template <class T, unsigned int blockSize, bool nIsPow2>
__global__ void
reduce6(T *g_idata, T *g_odata, unsigned int n)
{
    T *sdata = SharedMemory<T>();

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;

    T mySum = 0;
    while (i < n)
    {
        mySum += g_idata[i];
        if (nIsPow2 || i + blockSize < n)
            mySum += g_idata[i+blockSize];
        i += gridSize;
    }
sdata[tid] = mySum;
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] = mySum = mySum
+ sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] = mySum = mySum
+ sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <  64) { sdata[tid] = mySum = mySum
+ sdata[tid +  64]; } __syncthreads(); }
```

```
  if (tid < 32)
   {
      volatile T* smem = sdata;
      if (blockSize >=  64) { smem[tid] = mySum = mySum + smem[tid + 32];  }
      if (blockSize >=  32) { smem[tid] = mySum = mySum + smem[tid + 16];  }
      if (blockSize >=  16) { smem[tid] = mySum = mySum + smem[tid + 8];  }
      if (blockSize >=   8) { smem[tid] = mySum = mySum + smem[tid + 4];  }
      if (blockSize >=   4) { smem[tid] = mySum = mySum + smem[tid + 2];  }
      if (blockSize >=   2) { smem[tid] = mySum = mySum + smem[tid + 1];  }
   }

   if (tid == 0)
      g_odata[blockIdx.x] = sdata[0];
}
extern "C" void reduce6_cuda_(int *n, int *a, int *b)
{
  int *b_d;
  const int b_size = *n;

  cudaMalloc((void **) &b_d , sizeof(int)*b_size);
  cudaMemcpy(b_d, b, sizeof(int)*b_size, cudaMemcpyHostToDevice);

  dim3 dimBlock(128, 1, 1);
  dim3 dimGrid(128, 1, 1);
  dim3 small_dimGrid(1, 1, 1);
  int smemSize = 128 * sizeof(int);
  int *buffer_d;
  int small_buffer[4],*small_buffer_d;

  cudaMalloc((void **) &buffer_d , sizeof(int)*128);
  cudaMalloc((void **) &small_buffer_d , sizeof(int));
  reduce6<int,128,false><<< dimGrid, dimBlock, smemSize >>>(b_d,buffer_d,
b_size);
  reduce6<int,128,false><<< small_dimGrid, dimBlock, smemSize
>>>(buffer_d, small_buffer_d,128);
  cudaMemcpy(small_buffer, small_buffer_d, sizeof(int),
cudaMemcpyDeviceToHost);

  *a = *small_buffer;

  cudaFree(buffer_d);
  cudaFree(small_buffer_d);
  cudaFree(b_d);
}
```

OpenACCでリダクションを実装する場合

- コンパイラが以下の機能を実行:
  - !$ACC内の並列化のできるループを確認
  - カーネル化する必要があるか判断
  - アクセラレータ向けコードCPU向けコードに分割
  - ホスト側とアクセラレータ側で計算実行の分担
    - MIMD もしくはSIMDスタイルで実行
  - データ転送
    - GPUメモリの割り当てと開放を!$ACC領域の最初と最後で実行
    - CPUとGPUでデータ転送

```
!$acc data present(a,b)
!$acc parallel

a = 0.0

!$acc loop reduction(+:a)

do i = 1,n
  a = a + b(i)
end do

!$acc end parallel
!$acc end data
```

コンパイラからの実行オブジェクト以外の出力

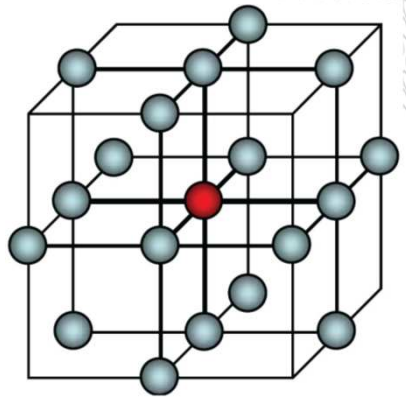
```
 90.        subroutine sum_of_int_4(n,a,b)
 91.        use global_data
 92.        integer*4 a,b(n)
 93.        integer*8 start_clock, elapsed_clocks, end_clock
 94.        !$acc data present(a,b)
 95.  G----< !$acc parallel
 96.  G      a = 0.0
 97.  G      !$acc loop reduction(+:a)
 98.  G g--< do i = 1,n
 99.  G g      a = a + b(i)
100.  G g--> end do
101.  G----> !$acc end parallel
102.        !$acc end data
103.        end subroutine sum_of_int_4
```

```
ftn-6413 ftn: ACCEL File = gpu_reduce_int_cce.F90, Line = 94
  A data region was created at line 94 and ending at line 107.

ftn-6405 ftn: ACCEL File = gpu_reduce_int_cce.F90, Line = 95
  A region starting at line 95 and ending at line 101 was placed on
the accelerator.

ftn-6430 ftn: ACCEL File = gpu_reduce_int_cce.F90, Line = 98
  A loop starting at line 98 was partitioned across the threadblocks
and the 128 threads within a threadblock.
```

リダクションの性能

プログラム言語	実行元	コードの長さ	Gflops性能	X86 1コアに対する性能
Fortran	x86CPU1コア	4	2.0 Gflops	1.0
CUDA	GPU	30	1.74 Gflops	0.87
最適化版 CUDA	GPU	69	10.5 Gflops	5.25
OpenACC	GPU	9	8.32 Gflops	4.16

ケーススタディ: 姫野ベンチマーク



- 並列3次元ポワソン方程式ソルバ
  - 19格子点
  - メモリアクセス中心、メモリバンド幅が性能を左右
- ソースコード
  - http://accc.riken.jp/HPC_e/himenobmt_e.html
- Fortran Coarray (CAF) による実装
  - **約600行**
  - **アクセラレータ向け実装に要したディレクティブは27**
- ストロングスケーリング
  - XLサイズ: 1024 x 512 x 512
  - ハロ交換が性能のボトルネック
  - データ転送とカーネル実行を非同期にすることで、通信と計算のオーバーラップ

姫野ベンチマーク：ヤコビ法カーネル

- 格子点が圧力要素を格納する配列Pに対応
- 配列wrk2に更新された値を保存
- 制御変数wgosa を計算
- ベンチマークでは決められた回数このカーネルが実行される

```
DO K=2,kmax-1
 DO J=2,jmax-1
  DO I=2,imax-1
   S0=a(I,J,K,1)*p(I+1,J, K )
     +a(I,J,K,2)*p(I, J+1,K ) &
     +a(I,J,K,3)*p(I, J, K+1) &
     +b(I,J,K,1)*(p(I+1,J+1,K )-p(I+1,J-1,K )  &
                 -p(I-1,J+1,K )+p(I-1,J-1,K )) &
     +b(I,J,K,2)*(p(I, J+1,K+1)-p(I, J-1,K+1)  &
                 -p(I, J+1,K-1)+p(I, J-1,K-1)) &
     +b(I,J,K,3)*(p(I+1,J, K+1)-p(I-1,J, K+1)  &
                 -p(I+1,J, K-1)+p(I-1,J, K-1)) &
     +c(I,J,K,1)*p(I-1,J, K ) &
     +c(I,J,K,2)*p(I, J-1,K ) &
     +c(I,J,K,3)*p(I, J, K-1) &
     + wrk1(I,J,K)

   SS = (S0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
   wgosa = wgosa+ SS*SS
   wrk2(I,J,K)=p(I,J,K)+OMEGA *SS
  ENDDO
 ENDDO
ENDDO
```

fwd n.n.
n.n.n.
bwd n.n.

姫野ベンチマーク：分散並列での実装

- 外側のループはnn回実行
- ヤコビ法カーネルは、新しい圧力の値の配列 wrk2 と 制御変数wgosa を引数に実行
- 更新されたwrk2の値をpにコピー
- ハロ領域の値は隣接したPE間で通信
- AllReduceで制御変数 wgosa を求める

```
DO loop = 1, nn

  compute Jacobi: wrk2, wgosa

  copy back wrk2 into p

  pack halo from p into send buf

  exchange halos with neighbor PEs

  unpack halo into p from recv buf

  Allreduce to sum wgosa across Pes

ENDDO
```

GPUでのメモリ割り当て

- **data**ディレクティブを使いメインプログラム実行時にGPU上メモリの割り当てを行う
- **data** ディレクティブに**present**を付け加える事で既に割り当てられたGPUメモリは維持される
- **Present** 使用時は**copy*** は使えなくなる。CPUとGPU間のデータ転送**update** ディレクティブを使う

```
PROGRAM himenobmtxp
...
!$acc data create                    &
!$acc&  (p,a,b,c,wrk1,wrk2,bnd,      &
!$acc&  sendbuffx_up,sendbuffx_dn,&
!$acc&  sendbuffy_up,sendbuffy_dn,&
!$acc&  sendbuffz_up,sendbuffz_dn)
...
!$acc end data


SUBROUTINE jacobi(nn,gosa)
!$acc data present                   &
!$acc&  (p,a,b,c,wrk1,wrk2,bnd,      &
!$acc&  sendbuffx_up,sendbuffx_dn,&
!$acc&  sendbuffy_up,sendbuffy_dn,&
!$acc&  sendbuffz_up,sendbuffz_dn)
```

ヤコビ法カーネルのGPU化

- parallel loop ディレクティブ内のゴードがGPU化
- gosawとgosaは既にdata ディレクティブ でGPUメモリ上にあるので何もする必要なし
- OpenMP と同様 reductionを使うことで wgosa を計算
- vector_length はスレッドブロック内のスレッド数を変更する際に使う(デフォルトは128)

```
DO loop=1,nn
  gosa = 0
  wgosa = 0
!$acc parallel loop                    &
!$acc&  private(s0,ss)                 &
!$acc&  reduction(+:wgosa)             &
!$acc&  vector_length(256)
  DO K=2,kmax-1
    DO J=2,jmax-1
      DO I=2,imax-1
        S0=a(I,J,K,1)*p(I+1,J, K )&
        ...
        wgosa = wgosa + SS*SS
      ENDDO
    ENDDO
  ENDDO
```

ハロ領域のバッファのコピー

- wrk2 にあるハロの値はGPUに割り当てられたsendバッファにコピーされる
- parallel 領域内では loop ディレクティブを使って、ループをGPU化
- sendバッファはupdateでCPU側のデータをコピー
- 同様にrecvバッファ もupdateデレクティブでCPU側にあるデータをGPU へコピー

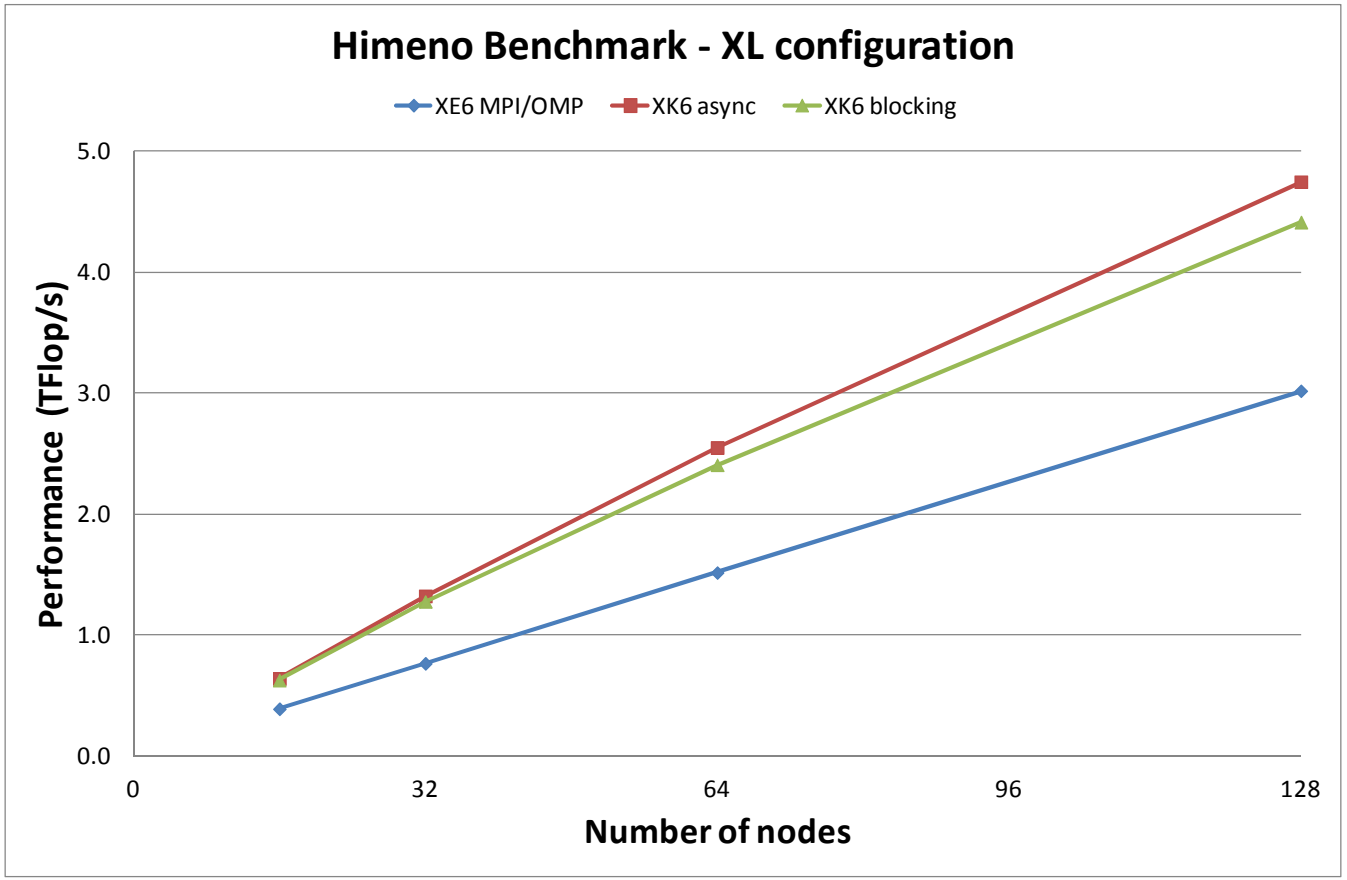
```
!$acc parallel
!$acc loop
DO j = 2,jmax-1
  DO i = 2,imax-1
    sendbuffz_dn(i,j)= wrk2(i,j,2)
    sendbuffz_up(i,j)= wrk2(i,j,kmax-1)
  ENDDO
ENDDO
!$acc end loop
 ...
!$acc loop
 ...
!$acc end loop
!$acc end parallel

!$acc update host
!$acc&       (sendbuffz_dn,sendbuffz_up)
```

姫野ベンチマーク：性能評価

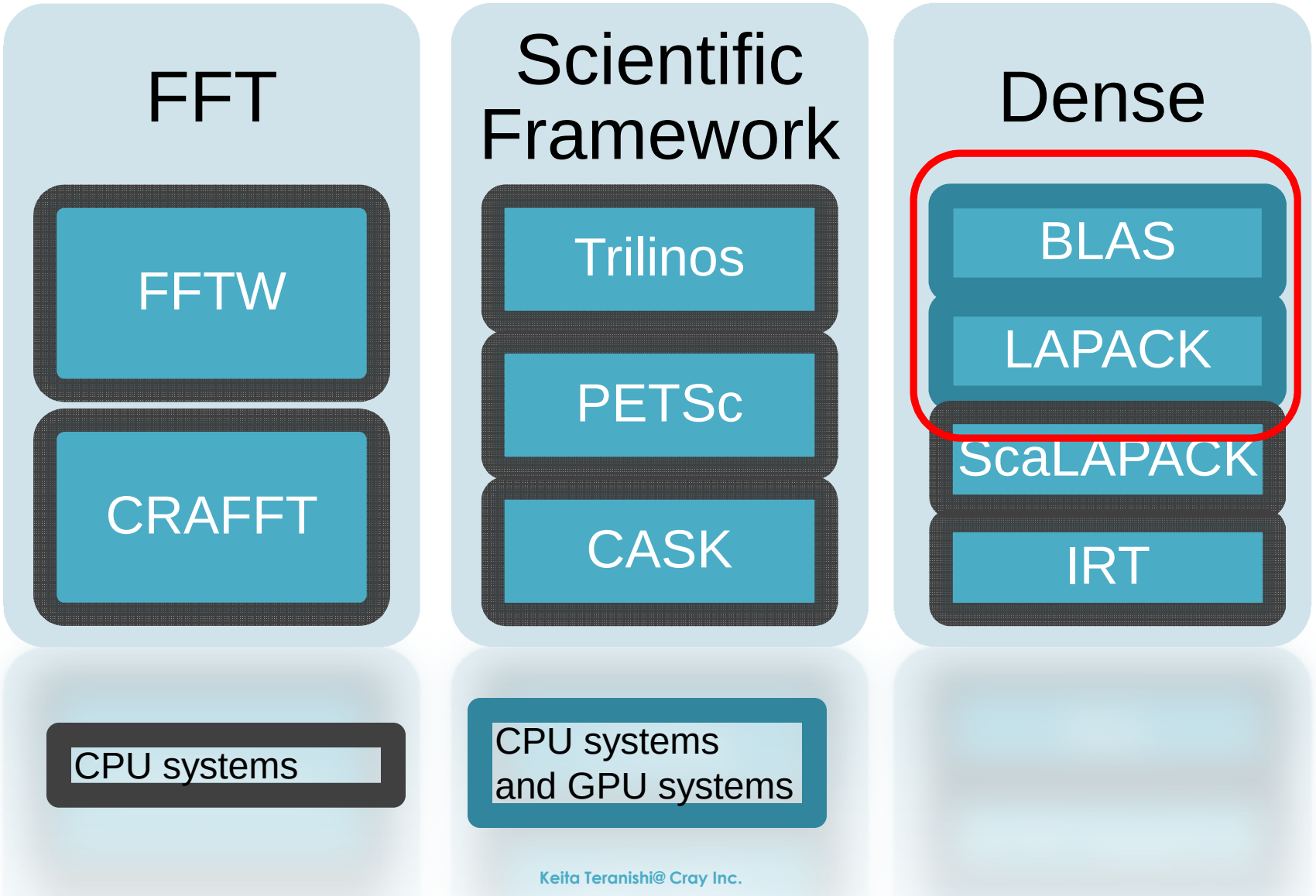
- **Cray XK6 :**
  - 1 AMD IL-16 2.1GHz nodes, 16 cores per node
  - Nvidia Tesla X2090 GPU, 1 GPU per node
  - 1ノードあたり1 PE (GPU)
  - XLサイズの問題に 16 ノード必要
  - 同期と非同期バージョンを実行
- **Cray XE6**:
  - Dual AMD IL-16 2.1 GHz nodes, 32 cores per node
  - 全コアを使用
  - 1PEあたり1-4 OpenMP スレッドを使用
    - 1ノードあたり、8-32PEs
- **XLサイズの問題で性能評価**

姫野ベンチマーク：OpenACCによる性能向上

- CPUx2のXEノードの1.6倍の性能向上
- OpenACC async 命令で約 8% の性能向上



Crayの科学計算ライブラリ



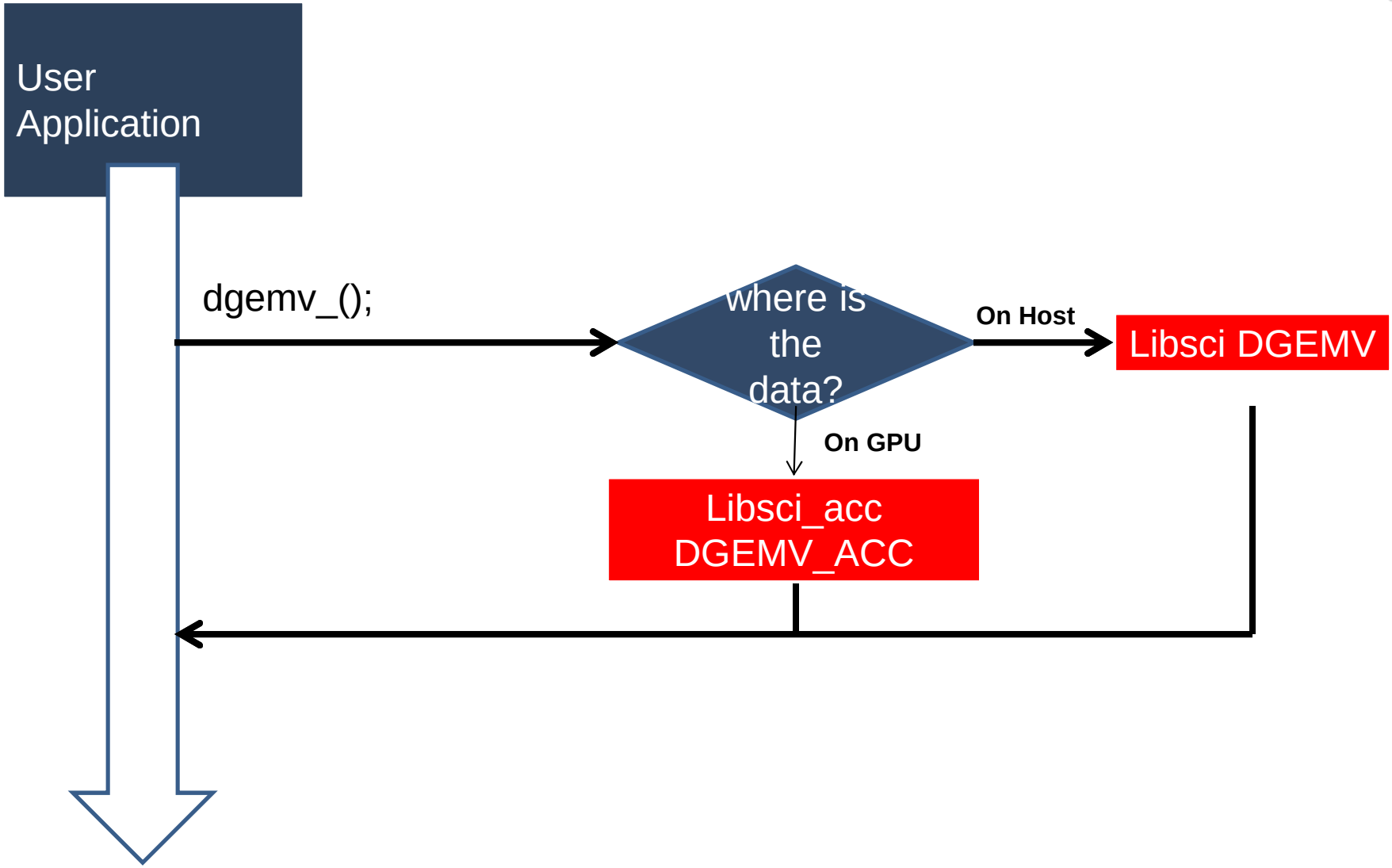
Libsci_acc

- XK向けBLAS とLAPACK
- CUDA、OpenACCの両方に対応
- 2種類のインターフェース
  - Simpleインターフェース
    - ソースコードの変更なく、GPUの使用
  - Expertインターフェース
    - 僅かなコードの変更でGPUを有効に利用

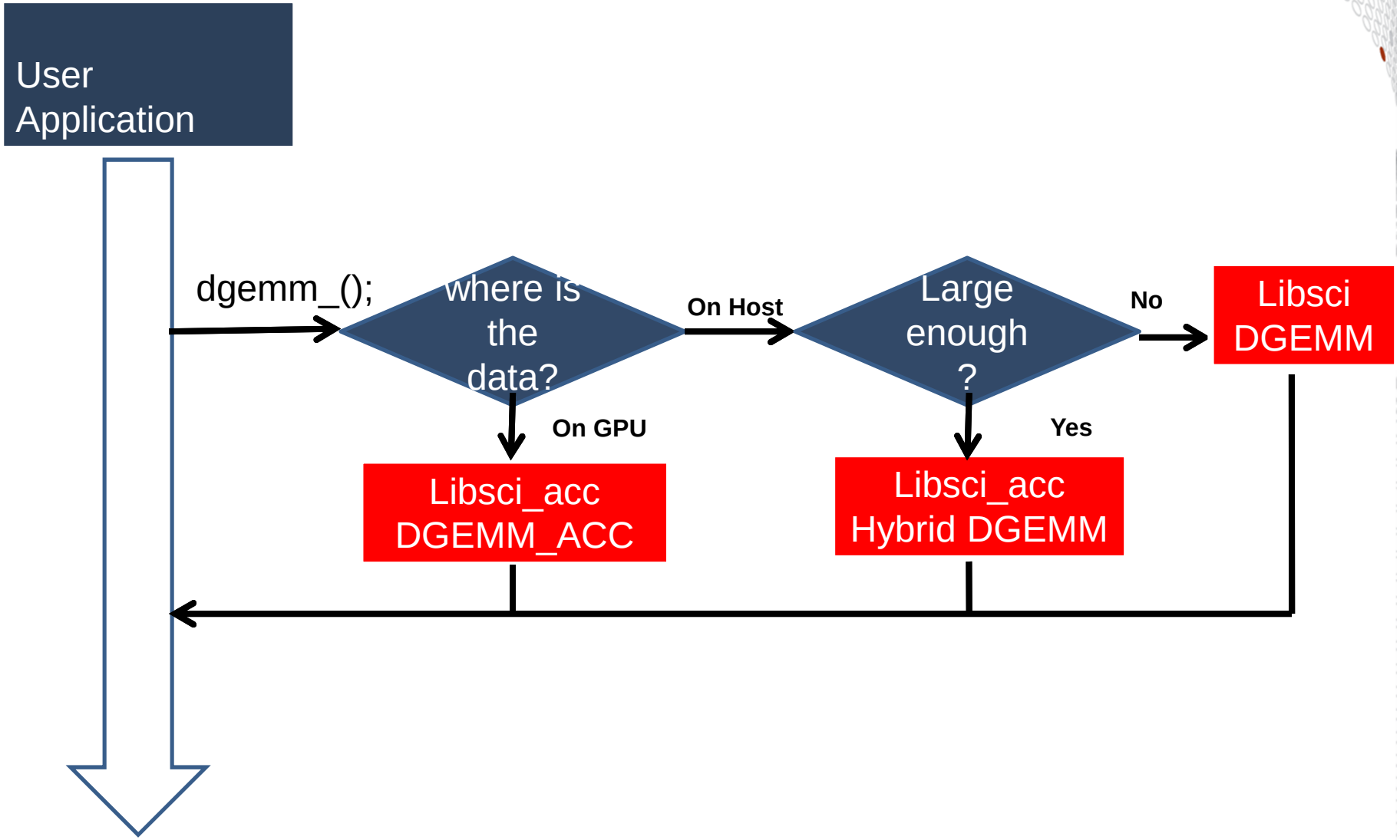
Libsci_accのインターフェース

- **libsci_acc simple interface**
  - コード変更をしないで、GPUの性能を引出す
    - **GPU,CPU,ハイブリッド実行の自動判別**
    - **GPU初心者向け**
- **libsci_acc expert interface**
  - ユーザ側で、CPU,GPU、ハイブリッド実行を直接指示
  - 自動判別をスキップ
  - インタフェースの拡張を予定
    - 様々なオプションで性能をより引出せる

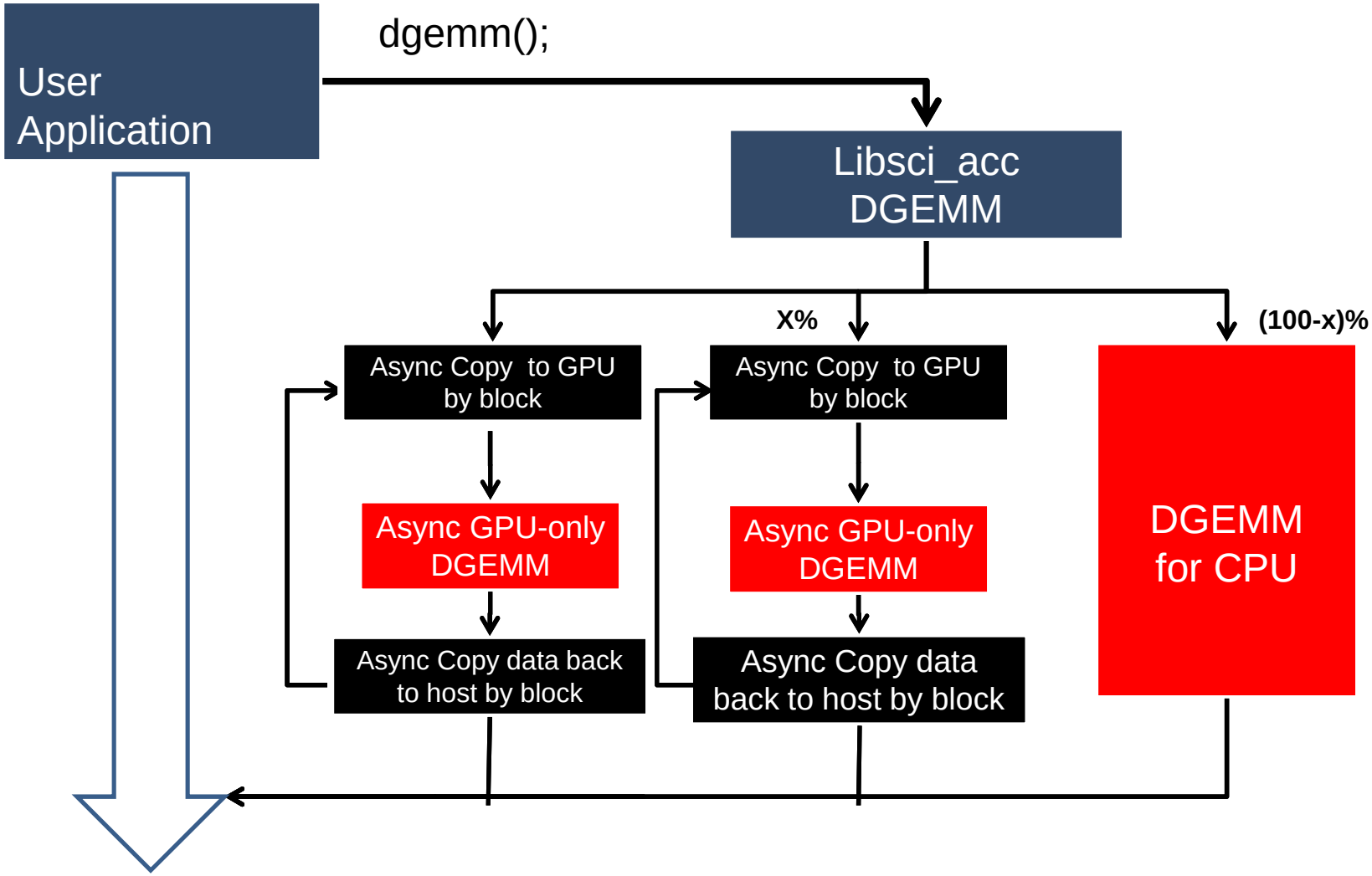
Libsci_acc: Simple Interface for BLAS1 and BLAS2



Libsci_acc: Simple Interface for BLAS3 and LAPACK



Libsci_acc: Simple Interface – Hybrid DGEMM



Libsci_ACCルーチン

- BLAS
  - **[s,d,c,z]GEMM**
  - **[s,d,c,z]TRSM**
  - [z,c]HEMM
  - [s,d]SYMM
  - [s,d,c,z]SYRK
  - [z,d]HERK
  - [s,d,c,z]SYR2K
  - [s,d,c,z]TRMM
  - **All level 2 BLAS**
  - **All level 1 BLAS**
- LAPACK
  - **[d,z]GETRF**
  - **[d,z]GETRS**
  - **[d,z]POTRF**
  - **[d,z]POTRS**
  - **[d,z]GESDD**
  - **[d,z]GEBRD**
  - **[d,z]GEQRF**
  - **[d,z]GELQF**

Eigenvalue Solvers

- **DSYEV**
- **ZHEEV**
- **DSYEVR**
- **ZHEEVR**
- **DSYEVD**
- **ZHEEVD**
- **DSYGVD**
- **ZHEGVD**
- **DGEEV**
- **ZGEEV**

Full-HYBRID HYBRID is planned **No HYBRID**

Libsci_accの使い方

- メインプログラムの最初に**libsci_acc_init**で初期化
- DGEMMの呼び出し方は従来のDGEMMと一緒
- 行列のサイズに合わせて、CPU,GPU,ハイブリッド実行を選択
- 行列データA,B,CはCPU側

```
call libsci_acc_init()
:
call libsci_host_alloc(a,8*m*lda)
:

call dgemm('n','n',m,n,k,alpha,&
           a,lda,b,ldb,beta,c,ldc)
```

Libsci_accの使い方：CPUルーチンの実行

- メインプログラムの最初に**libsci_acc_init**で初期化
- DGEMMの呼び出し方は従来のDGEMMと一緒
- 行列のサイズに合わせて、CPU,GPU,ハイブリッド実行を選択
- 行列データA,B,Cは CPU側

```
call libsci_acc_init()
:
:
:

call dgemm_cpu('n','n',m,n,k,alpha,&
               a,lda,b,ldb,beta,c,ldc)
```

Libsci_accの使い方:
OpenACCでGPUルーチンの実行

- CPU－GPU間のデータ転送はOpenACCで処理
- DGEMMのGPU用インターフェース

```
!$acc data copy(a,b,c)

!$acc parallel
!Do Something（GPU実行）
!$acc end parallel

!$acc host_data use_device(a,b,c)

call dgemm_acc('n','n',m,n,k,&
               alpha,a,lda,&
               b,ldb,beta,c,ldc)

!$acc end host_data
!$acc end data
```

Libsci_accの使い方:
OpenACCでGPUルーチンの実行

- CPU－GPU間のデータ転送はOpenACCで処理
- Simpleインターフェースで使用

```
!$acc data copy(a,b,c)

!$acc parallel
!Do Something（GPU実行）
!$acc end parallel

!$acc host_data use_device(a,b,c)

call dgemm ('n','n',m,n,k,&
                alpha,a,lda,&
                b,ldb,beta,c,ldc)

!$acc end host_data
!$acc end data
```

Libsci_acc Example
OpenACCでの使い方

- 基本的にBLASの呼び出し方と同じ
- 配列aはGPUメモリ上にあるのでそれに合わせた実装が内部で実行される。
- ピボット情報はCPUに保存される
  - GPUメモリ上の場合もサポート予定

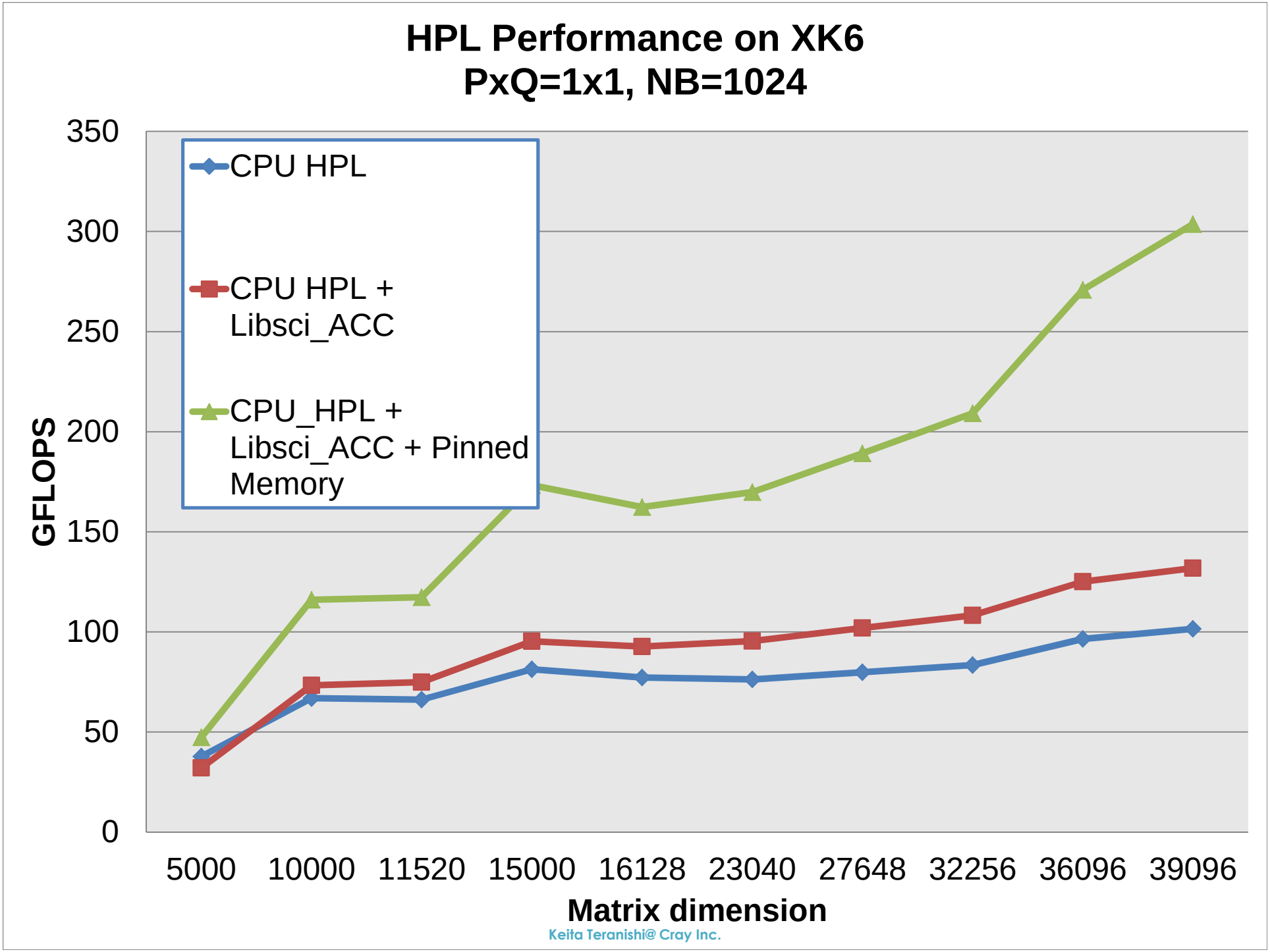
```
!$acc data copy(a)

!$acc parallel
!Do Something
!$acc end parallel

!$acc host_data use_device(a)
 call dgetrf (m,n,a,lda,ipiv,info)
!$acc end host_data
!$acc end data
```

HPLでLIBSCI_ACCを実行

- 2行だけ変更:
  - `malloc`を`libsci_acc_HostAllocc`に変更し、メモリを固定
  - `Free`を`libsci_acc_FreeHost`に変更

- 実行は通常のプログラムと同様

```
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR11R4L2       39096  1024     1     1             126.66              3.146e+02
```


HPL Performance on XK6
PxQ=1x1, NB=1024
GFLOPS
350
300
250
200
150
100
50
0
CPU HPL
CPU HPL + Libsci_ACC
CPU_HPL + Libsci_ACC + Pinned Memory
5000
10000
11520
15000
16128
23040
27648
32256
36096
39096
Matrix dimension
Keita Teranishi@ Cray Inc.

実アプリケーションでの利用

- **WL-LSMS**
  - 2008年ゴードンベル賞、オークリッジ国立研究所Jaguar(XT5)で実装
  - 金属が絶縁体になる温度を予測する事を可能に
- **カーネル部分**

```
do iblk=nblk,2,-1
   m=n
   ioff=joff
   n=blk_sz(iblk-1)
   joff=joff-n
   call zgesv( m, ioff,a(ioff+1,ioff+1),lda,ipvt,a(ioff+1,1),lda,info)
   if(iblk.gt.2) then
          call zgemm('n','n',n,ioff-k+1,na-ioff,cmone,a(joff+1,ioff+1),lda,
  &     a(ioff+1,k),lda,cone,a(joff+1,k),lda)
           call zgemm('n','n',joff,n,na-ioff,cmone,a(1,ioff+1),lda,
  &     a(ioff+1,joff+1),lda,cone,a(1,joff+1),lda)
   endif
end do
```

	Jaguarpf node (12 cores AMD Istanbul)	Fermi C2050 using Cray Libsci
Time (sec)	13.5	6.4

GAMESS

- **Iowa State Universityで開発された計算化学パッケージ**
  http://www.msg.ameslab.gov/gamess/

- **関数CCSD(T)が実行時間の大半を占める**
  - Method to calculate electronic correlation energy in water clusters
  - 3重ループ:
    - ノード間通信
    - 配列データの書き換え
    - 多数のDGEMMの呼び出し
  - GPU計算に最適
    - ホスト、デバイス間のデータ転送
    - 配列データをGPU上で実行
    - Libsci_ACC版DGEMM
  - **OpenACCとLibsci_ACC を併用**

GAMMES：OpenACCによるソース

```
!$acc data present(t2_i,t2_j,vm_ij,vm_ji,vm_ii,ve_i,ve_j,v3)
!$acc host_data use_device(t2_i,t2_j,vm_ij,vm_ji,vm_ii,ve_i,ve_j)

!$acc host_data use_device(v3)
call dgemm('n','n',nr,nu,no,om,t2_i(sr,1),nu2,vm_ij,no,zero,v3(sr),nu2)
! #1: Type A

!$acc wait  ! for ve_j

!$acc host_data use_device(v3)
call dgemm(‘n’,‘n’,nr,nu,no,om,t2_i(sr,1),nu2,vm_ji,no,one,v3(sr),nu2)
call dgemm('n','n',nu,nr,nu,one,t2_i(1,j),nu,ve_i(t3off),nu,one,v3(t3off),nu)
!$acc end host_data

!$acc end host_data
!$acc end data
```

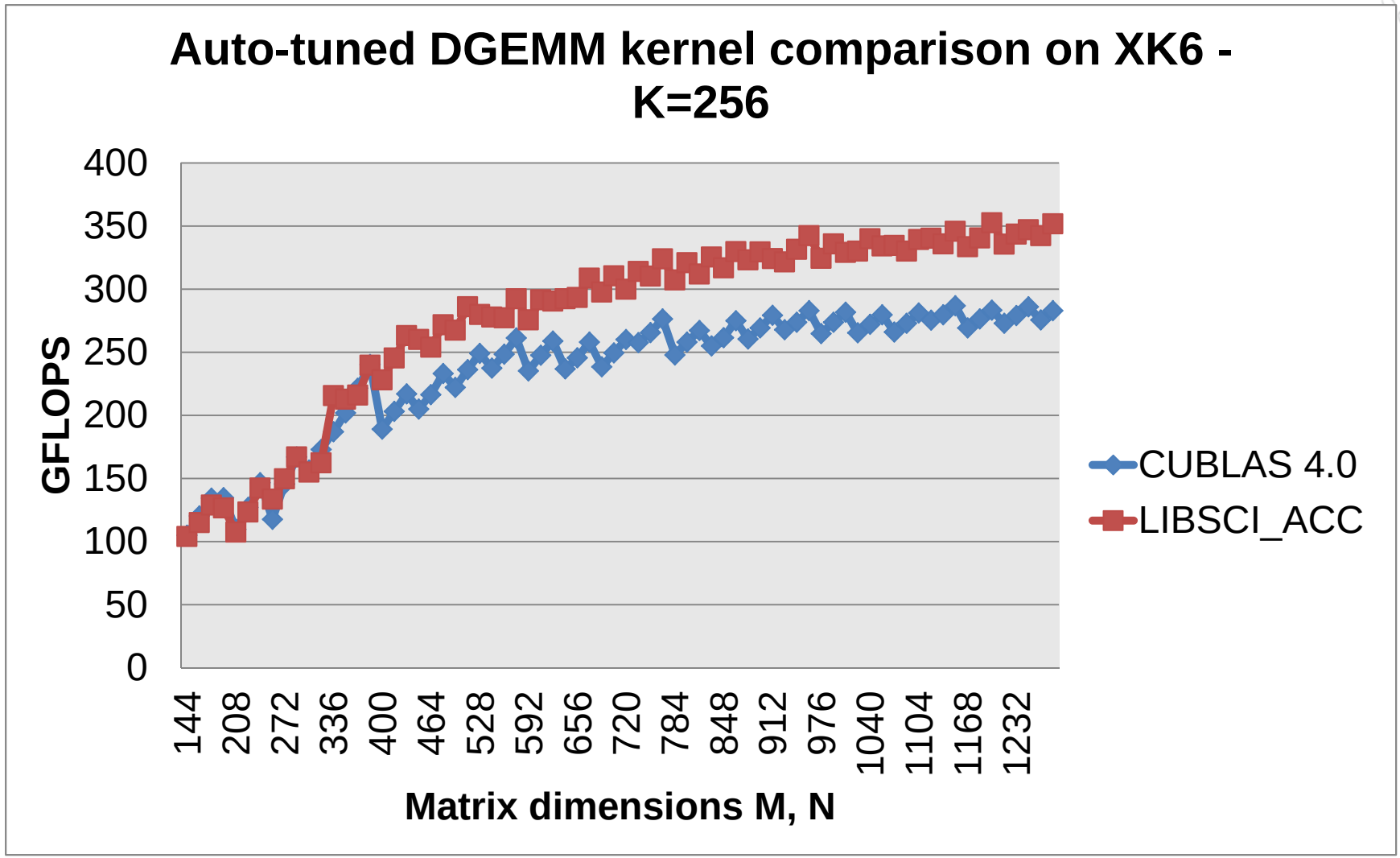
OpenACC vs. CUDA

- ソースコード
  - OpenACC – ディレクティブを**75行追加**
  - CUDA – **1800行**
- **CCSD()の性能**
  - OpenACC – 36.3 秒
  - CUDA – 34.8 秒

自動チューニング

- **Cray の自動チューニングフレームワークでBLASカーネルを最適化**
  - コードの自動生成
    - 100前後のカーネルの性能を評価
  - オフラインで性能評価
    - 1万通り以上のm,n,kで性能評価
  - 評価結果をもとに、各問題サイズ、入力パラメータに対し最高の性能を出せるカーネルを自動的に実行
  - 実行時に、性能データベースが更新され、それにより最適化の精度が上がる
  - 開発中

Auto-Tuned DGEMM



CUBLAS4.1 improved performance. We are targeting to replace CUBLAS5.0 for Kepler.

今後のリリース予定

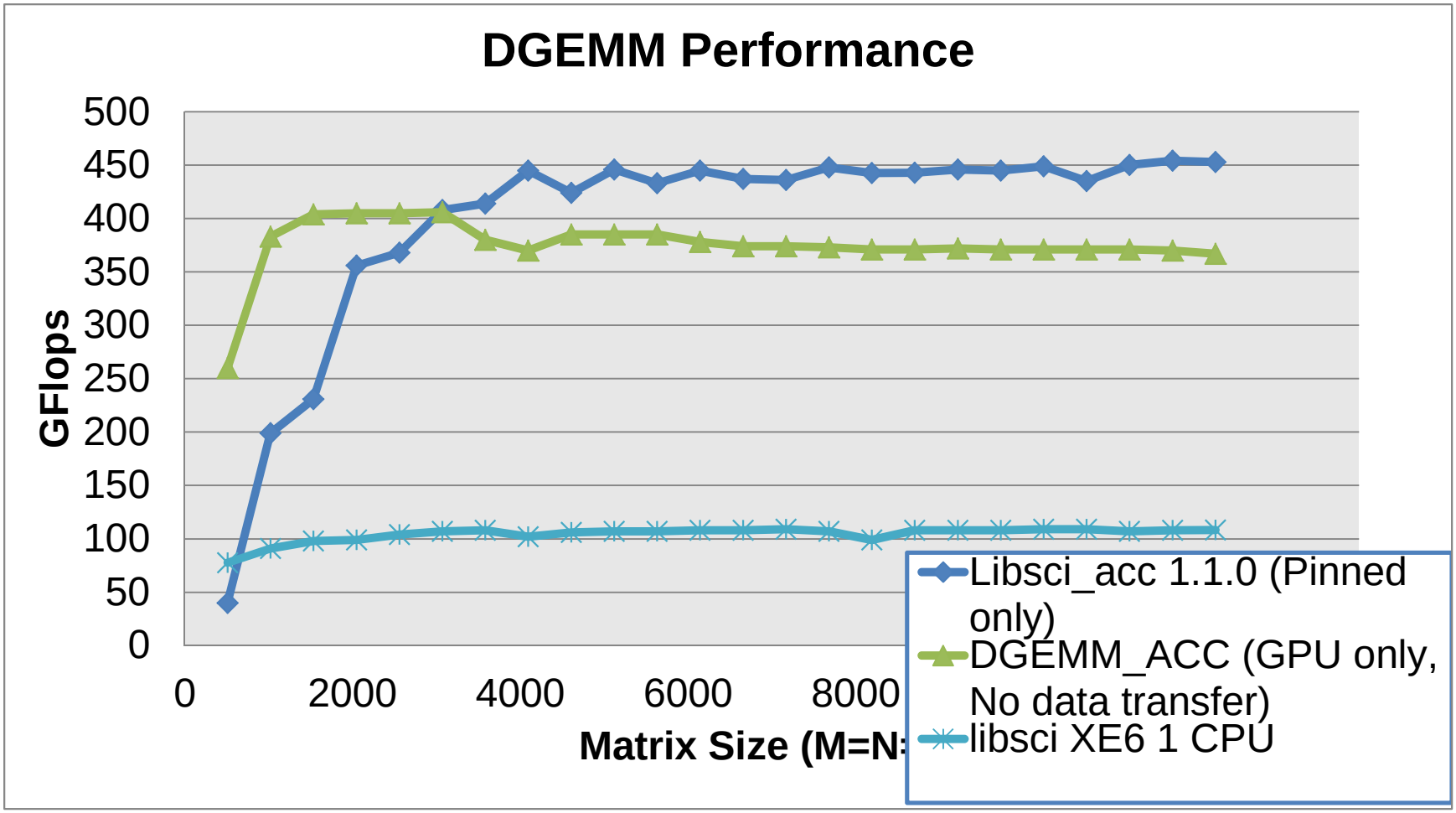
- CUDA 5 とKepler のサポート
- 自動チューニング機能付き BLAS
- LAPACKのサポートの強化
  - 性能、サポートするルーチン数
- GEMM以外のLevel 3 BLASのハイブリッド化
- スレッドセーフのサポート

THANK YOU!

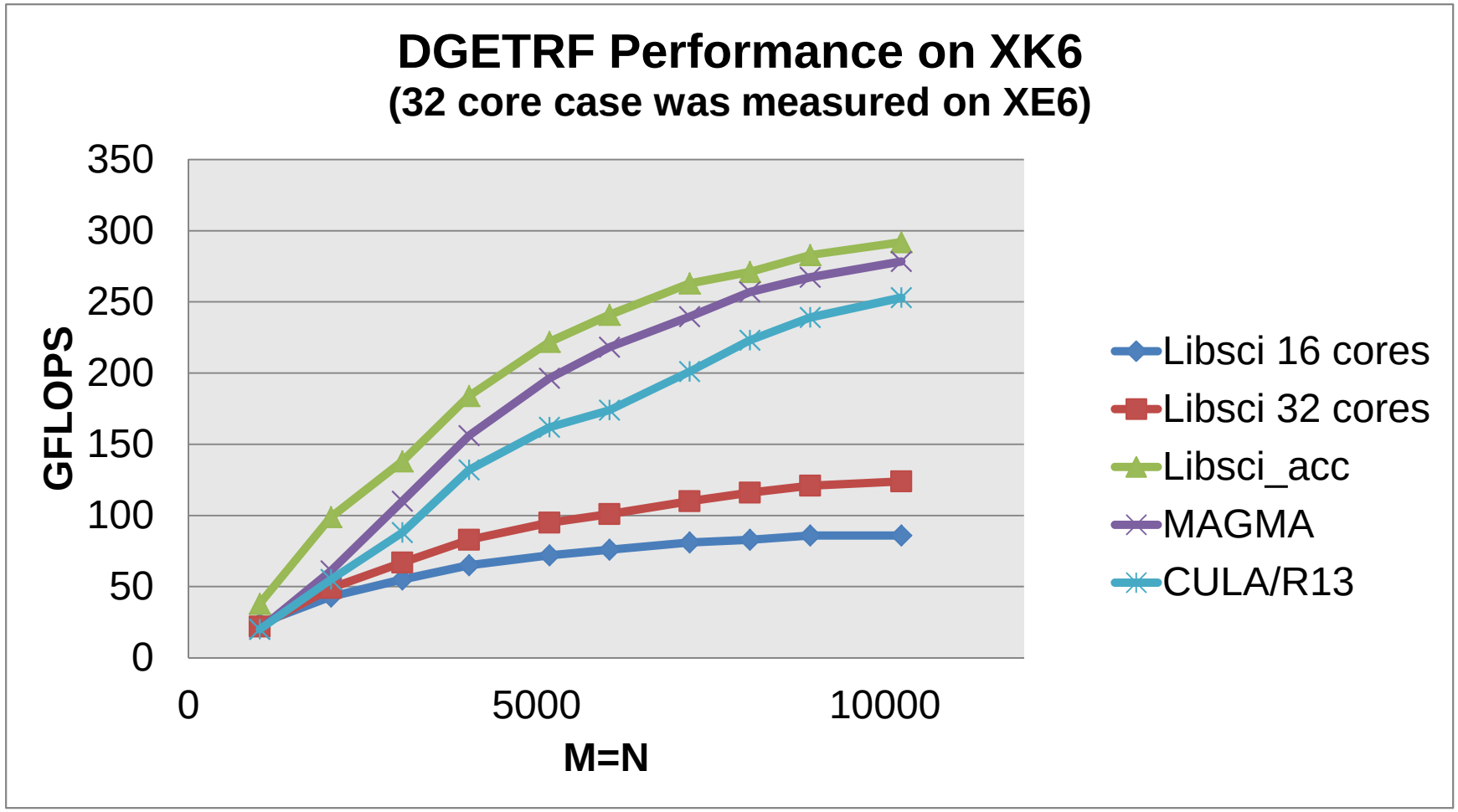
Keita Teranishi@ Cray Inc.

Libsci_ACCの性能

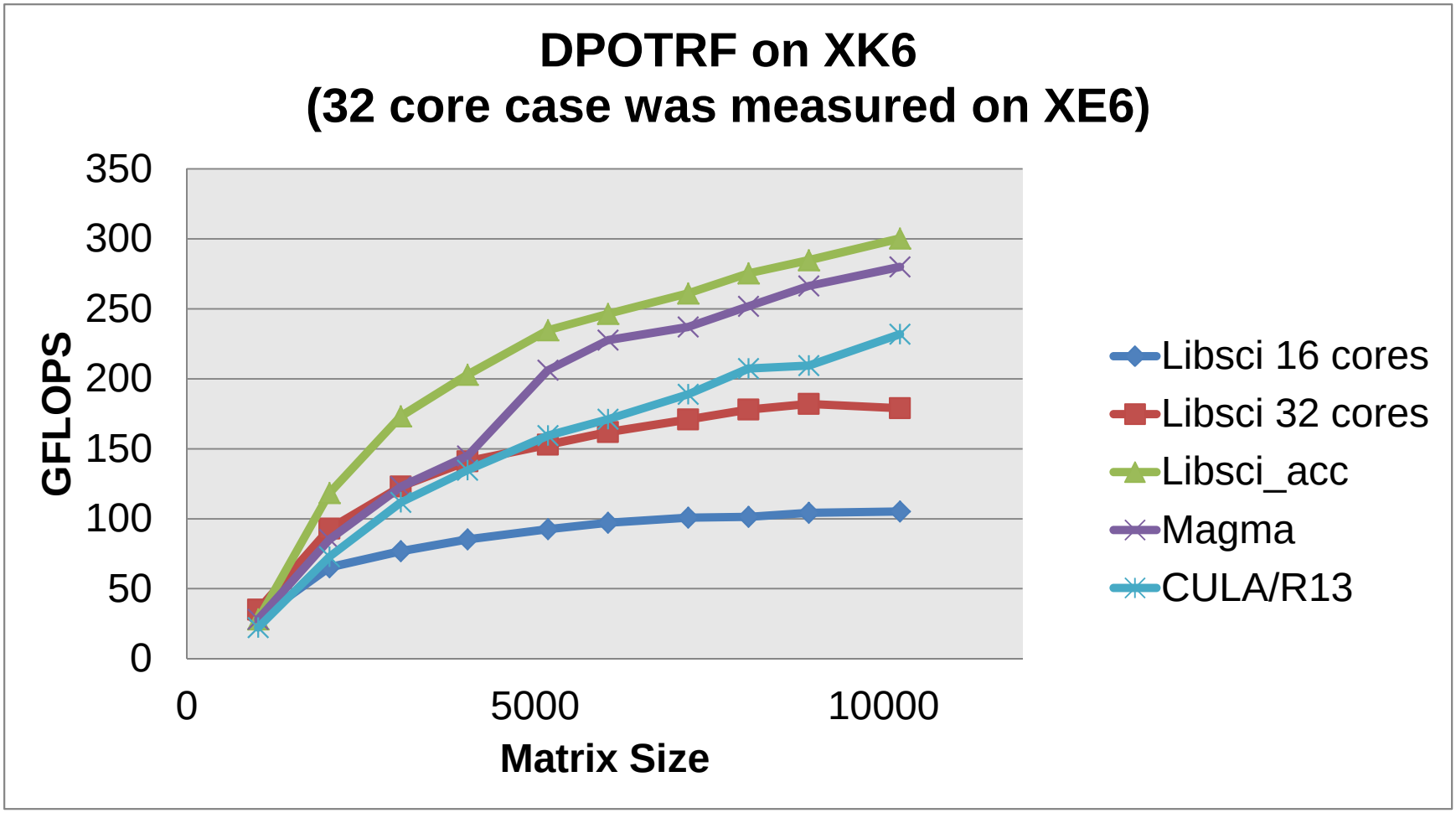
DGEMM


DGEMM Performance
GFlops
500
450
400
350
300
250
200
150
100
50
0
0
2000
4000
6000
8000
Matrix Size (M=N=
Libsci_acc 1.1.0 (Pinned only)
DGEMM_ACC (GPU only, No data transfer)
libsci XE6 1 CPU

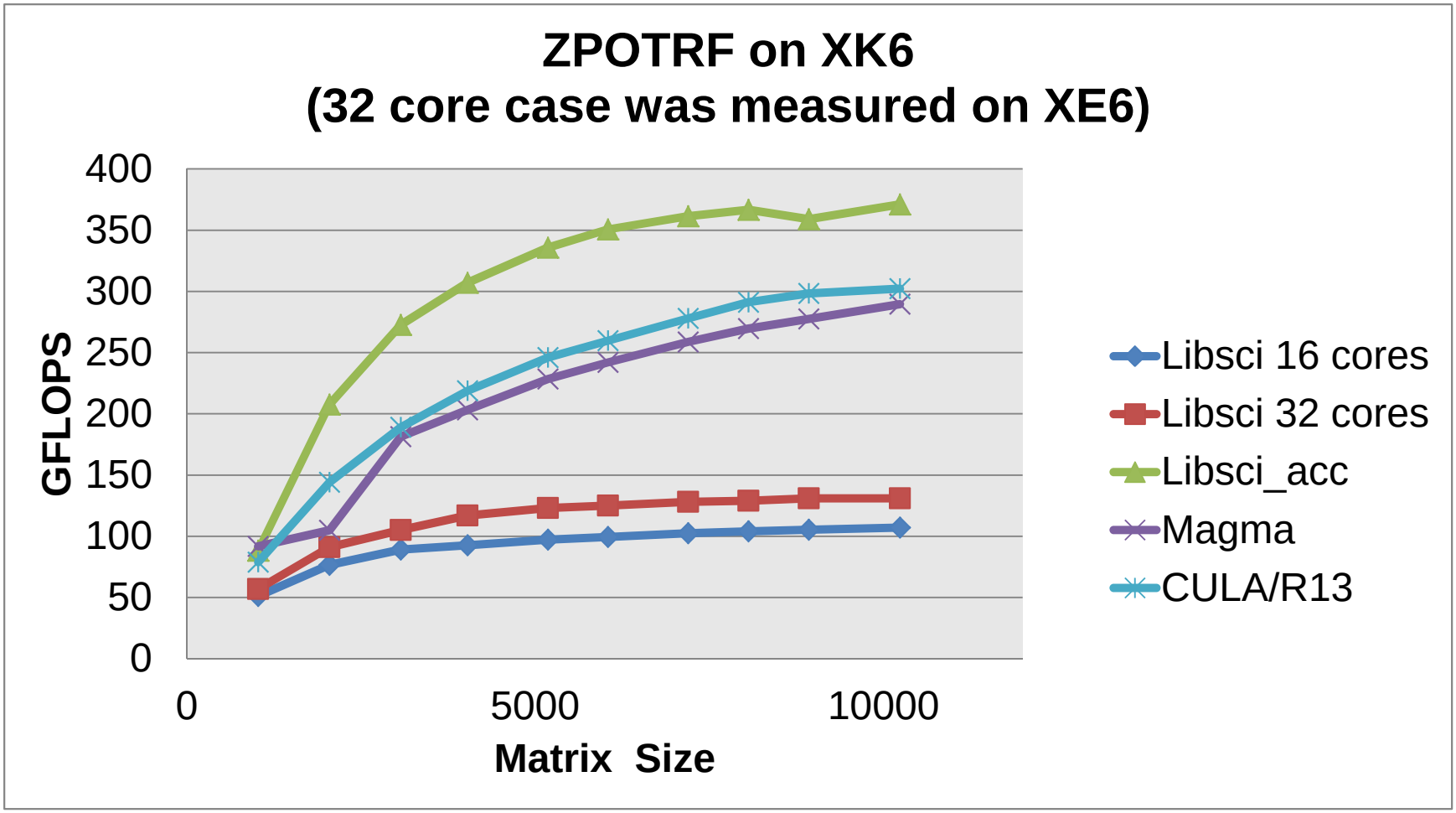
Libsci_ACC – LU分解



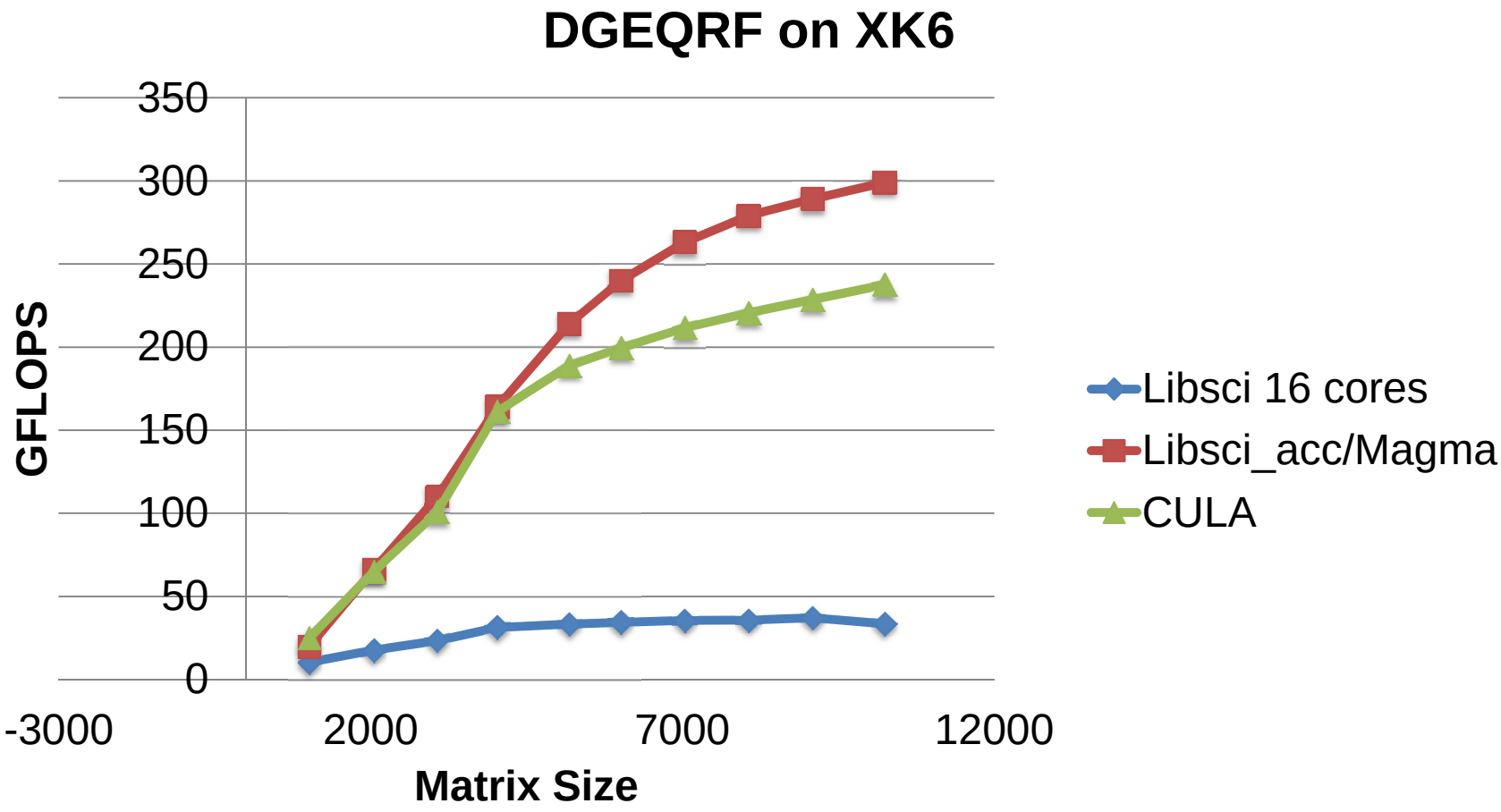
Libsci_ACC – コレスキー分解



Libsci_ACC – 倍精度複素数コレスキー分解

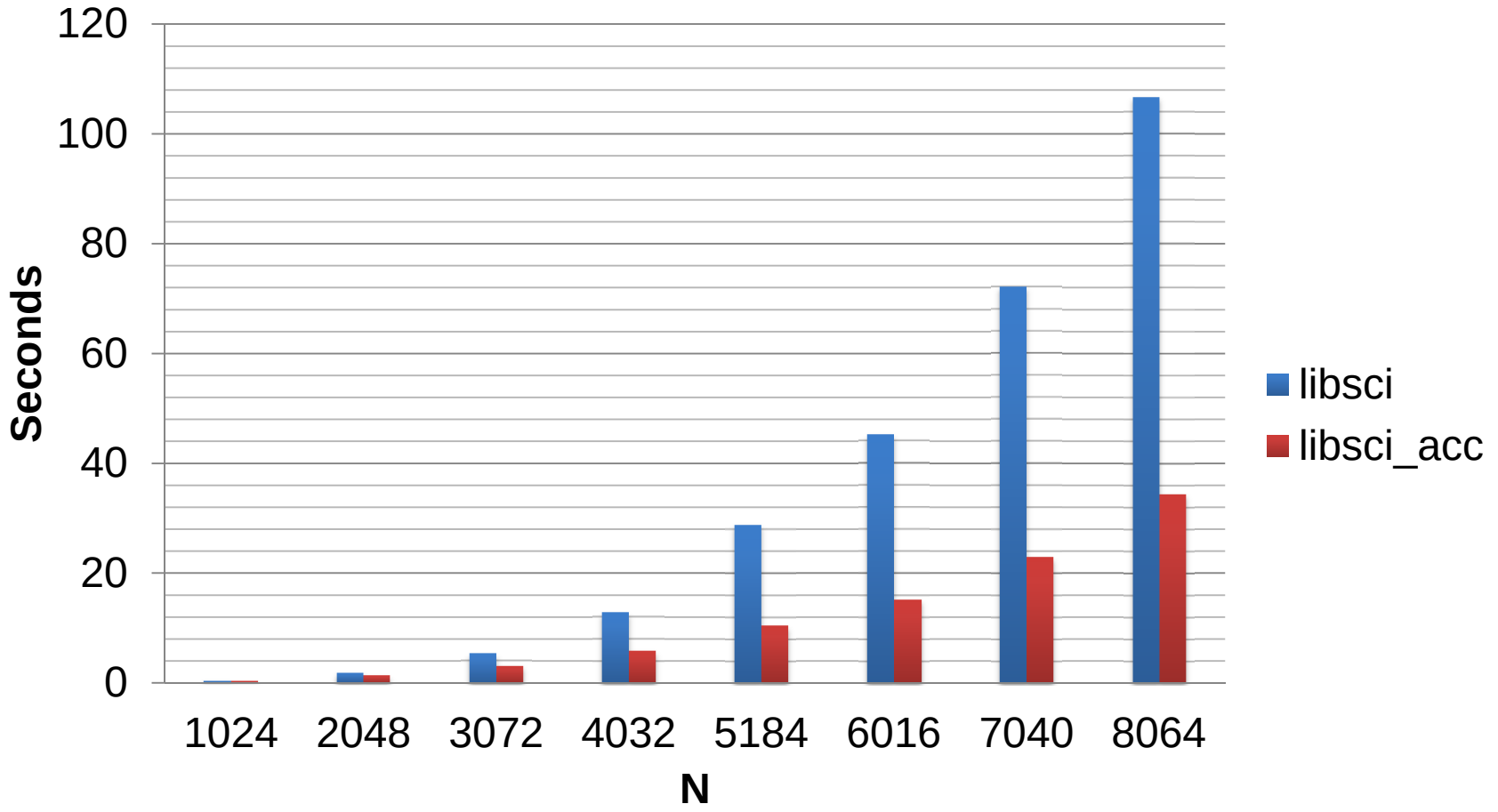


DGEQRF (QR Factorization)

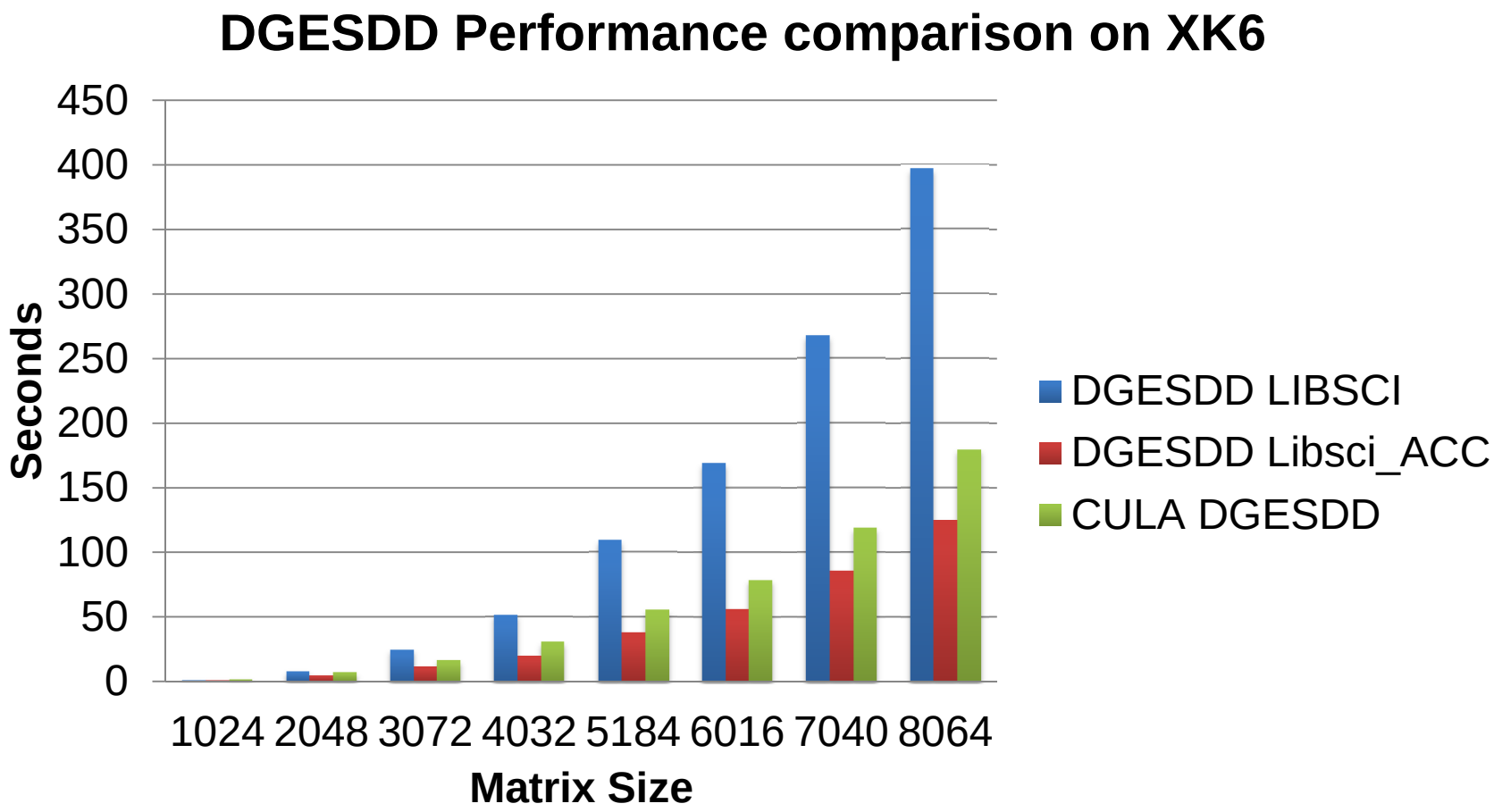


32 core case is slower than 16 core case.

DSYEVD：通常固有値ソルバ



DGESDD：特異値分解ソルバ



DGESVD uses an inefficient O(N^3) algorithm .
We recommend DGESDD for Libsci/Libsci_acc