X-Ability
クロスアビリティ

2011年3月3日

GPGPUハンズオンプログラミング演習

株式会社クロスアビリティ
rkoga@x-ability.jp

X-Ability クロスアビリティ

講師： 古賀良太／古川祐貴

- X-Ability クロスアビリティ 取締役
  計算化学ソルバー XA-CHEM-SUITEの開発
- HPC SYSTEMS コンサルティングパートナー
  並列ソフトウェアの開発・ビルド、サーバ販売
- allinea SCALE TO NEW HEIGHTS ソフトウェア代理店

X-Ability
クロスアビリティ

会社紹介

- 社名
  株式会社クロスアビリティ (X-Ability Co.,Ltd) ※役員３名のみ
- 業務内容
  計算科学関連ソフトウェアの開発、販売
  フィールドルータの開発、販売
- ビジネスモデル
  産学連携によるプロダクト開発、ベンダとの連携による販売
- 設立
  2008年1月
- 主な製品
  XA-CHEM-SUITE (XA-CUDA-QM etc.), Field Router

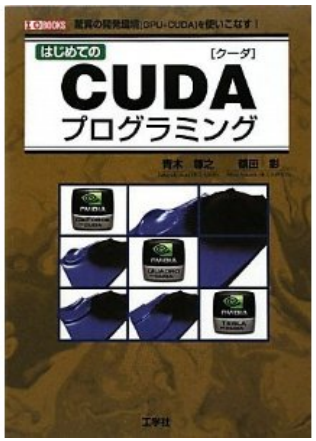
X-Ability
クロスアビリティ

前座

- 対象
  C/C++は一通り理解しているがGPGPUは初めての方
  ※対象でない方は“できる限り”個別に対応します
- 実践前提
  初プログラミング言語の学習で座学は意味がない
  少し書いてみてから各種ツールの有用性がわかる
- 絵は基本的に使わない
  ただしグラフィック処理の出力は別（今回はなし）
- 参考書籍に書いてあることは出来る限り書かない
  GPGPUの障壁をさげることが本講習会の目的
  コーディング＆コンパイルの体験と導入Tipsが重要

X-Ability
クロスアビリティ

- 参考書籍



- 参考URL
  - CUDA Programming Guide
  - CUDA Occupancy Calculator
    http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

X-Ability
クロスアビリティ

GPGPUとは？

- GPUを用いて汎用計算（科学技術計算など）を行うことをさす。
- 基本PCI Express x16バスにGPUを挿すだけだが、、、
  - デバイスドライバ（無料）のインストール要
  - 容量の大きい電源に交換要な場合が多い（電気食い）
  - マルチGPUの場合、host-device転送バスが同一バンド幅かチェック要
- 1つの命令で多数複数スレッドの同時演算が可能
  - 理論的には相当に高速だが（Fermi coreは価格性能比でCore i7の10倍）、Host(CPU+mem+HDD)-GPU転送コストがかかる + 特別な言語(CUDA)でコーディングしなおす必要がある。
- 高速だが容量が小さいon-chipメモリと低速だが容量が大きい（といっても数GB）external memoryによる構成
- 倍精度演算が単精度演算よりダイブ遅い
  - 倍精度ユニットが高速になったが、ソフトウェアが活用してない場合がある

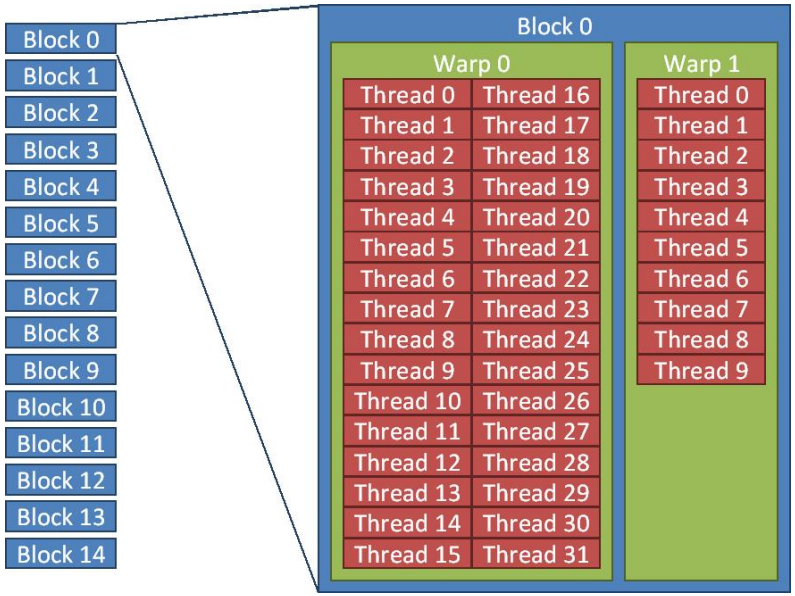
X-Ability
クロスアビリティ

CUDAプログラミングモデル

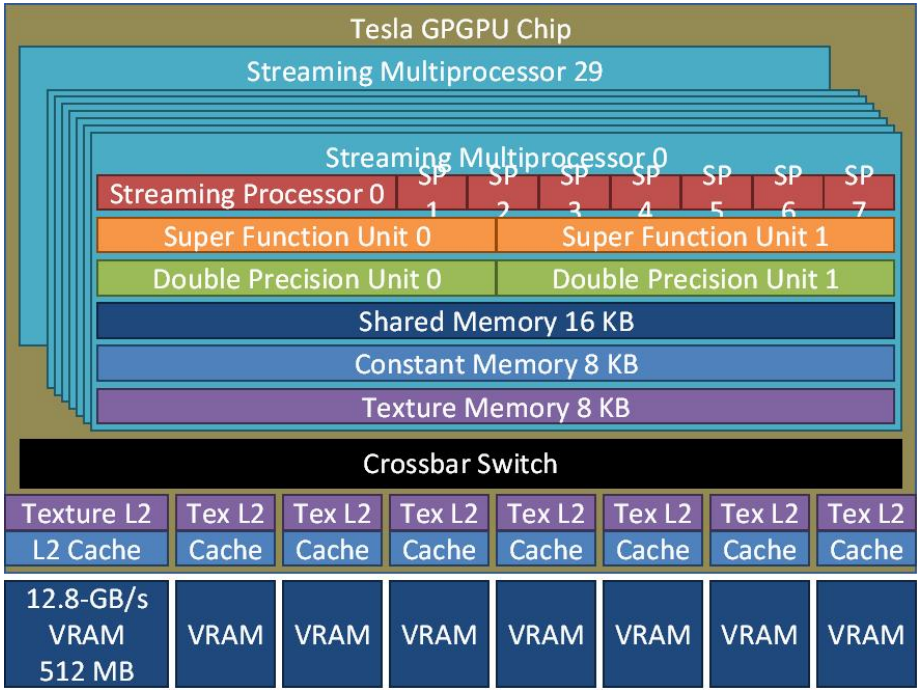
- GPUは多数のスレッドを並列実行できる外部演算デバイスとして扱われる
- GPUで走るプログラムをカーネルと呼び、カーネルを実行すると、同一のカーネルを実行するスレッドが多数走る
- 複数のスレッド群をまとめて、ブロックと呼ぶ。各ブロックは、ブロック内のスレッドのみがアクセスできる共有メモリを持つ
- カーネル実行時に、ブロック数及び各ブロックごとのスレッド数を指定する。概念的には、GPU上でブロック数×スレッド数の個数のスレッドが走ることになる
  - 実際には、Tesla C2050には448個しかCUDAコア（GPU演算コア）がないので、何千・何万のスレッドが同時に走るわけではない
  - 最適なブロック数・スレッド数というのはケースバイケース
- 各スレッドには、ブロックIDとスレッドIDという固有のIDが振られる。これらは3次元配列のindex

X-Ability
クロスアビリティ

CUDAプログラミングとGPUの関係



CPUスレッドから起動するkernel(grid, block)の中のblockの説明。GPU ThreadはblockのWarpという単位でスケジューリングされ、Warp内のthreadは同じ命令を実行する(SIMT)。多くのWarpを使うことでメモリ遅延を隠蔽できる。32threads単位で動く。



GlobalメモリはOff-chipなので、Global memoryからchipに転送する(ホスト機-GPU転送コストよりはダイブ低い)。

FermiはL1/L2 cacheが追加され、SMの数が16, SPの数が32になっている。

X-Ability
クロスアビリティ

実習内容

下記、行列乗算のコードを書くことで、徐々に高速化を実感してもらいます。

1. 普通に書くC++のコード
2. 普通に書くCUDAのコード
3. チューニングしたCUDAのコード
4. CUDAのライブラリを使ったコード

時間があれば、、、

5. チューニングしたOpenCLのコードを3と比較

X-Ability
クロスアビリティ

マシンアクセス方法

- Tesla C1060のマシン (kai)

  $ ssh gpuschoolXXX@133.30.112.247

  GPUのメモリキャッシュが効かないマシン

- Tesla C2050のマシン (ise)

  $ ssh gpuschoolXXX@133.30.112.247

  GPUのメモリキャッシュが効くマシン

  ※ gpuschoolXXX はアカウント名です

X-Ability
クロスアビリティ

メイン関数(main.cu)

```
#include <sys/time.h>
#include <stdio.h>
const int N = 512;              //512 x 512の行列乗算
const int M = N * N;

//時間計測用コード
double gettimeofday_sec()
{
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return tv.tv_sec + (double)tv.tv_usec*1e-6;
}

void matmul(float* A, float* B, float* C, int N);
int main(void)
{
  // 以下3行はCUDAのコードのみで必要  ※CUDAランタイムライブラリの初期化する時間を節約する
  float* p;
  cudaMalloc((void**)&p, sizeof(float));
  cudaFree(p);

  float* A = new float[M];
  float* B = new float[M];
  float* C = new float[M];

  for(int i=0; i<M; i++) A[i] = 1.0f;
  for(int j=0; j<M; j++) B[j] = 2.0f;

  double t1 = gettimeofday_sec();
  matmul(A,B,C,N);                    //  この行で関数を呼ぶ
  double t2 = gettimeofday_sec();

  float fAns= 1.0f * 2.0f * N;
  float fDiff = 0.0f;
  for(int k=0; k<M; k++) {
    float f = C[k] - fAns;
    fDiff += f * f;
  }
  printf("Time = %10.30f¥n", t2 - t1);
  printf("Accuracy : %f¥n", sqrt( fDiff / M ) );
  return 0;
}
```

X-Ability
クロスアビリティ

普通に書くC++のコード(naive_cpu.cpp)

```
#include <omp.h>
void matmul(float *A, float *B, float *C, int N)
{
#pragma omp parallel for          //※OpenMPによるスレッド並列
  for(int i=0; i<N; i++){
    for(int j=0; j<N; j++){
      float sum = 0.0f;
      for(int k=0; k<N; k++){
        sum += A[i*N+k]*B[k*N+j];
      }
      C[i*N+j] = sum;
    }
  }
}
```

コンパイルと実行

```
$ nvcc –O3 -Xcompiler -fopenmp main.cu naive_cpu.cpp
$ export OMP_NUM_THREADS=4
$ ./a.out
```

X-Ability
クロスアビリティ

普通に書くCUDAのコード(naive_cuda.cu)

```
__global__ void _matmul(float *A, float *B, float *C, int N)
{
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  float sum = 0.0f;
  for(int k=0; k<N; k++){
    sum += A[y*N+k] * B[k*N+x];
  }
  C[y*N+x] = sum;
}

//wrapper for _matmul kernel
void matmul(float *A, float *B, float *C, int N)
{
  float *devA, *devB, *devC;
  cudaMalloc((void**)&devA, sizeof(float)*N*N);
  cudaMalloc((void**)&devB, sizeof(float)*N*N);
  cudaMalloc((void**)&devC, sizeof(float)*N*N);

  cudaMemcpy(devA, A, sizeof(float)*N*N,
cudaMemcpyHostToDevice);
  cudaMemcpy(devB, B, sizeof(float)*N*N,
cudaMemcpyHostToDevice);

  //kernel execution
  dim3 nThreads(16, 16);
  dim3 nBlocks(N/16, N/16);

  _matmul <<< nBlocks, nThreads >>> (devA, devB, devC, N);
  cudaMemcpy(C, devC, sizeof(float)*N*N,
cudaMemcpyDeviceToHost);

  cudaFree(devA);
  cudaFree(devB);
  cudaFree(devC);
}
```

コンパイルと実行

$ nvcc –O3 main.cu naive_cuda.cu

$ ./a.out

X-Ability
クロスアビリティ

CUDAの最適化

今回関係あるのは2.と4.のみ（共有メモリ使う＆#pragma unroll）

1. グローバルメモリアクセスはcoalesce（複数スレッドからのメモリアクセスが1回のフェッチになるように）
2. 共有メモリ使うときはバンクコンフリクトをしないように
   CUDA PROFILEのwarp_serializeで回数が見れる
3. 条件分岐は減らす
4. loop unrollingは地味に有効
5. __syncthreadsも減らす
   Block内のスレッド間を同期しないようにすれば必要なくなる
6. オフチップメモリのレイテンシの隠蔽
   warpを沢山使えば、特定のwarpが演算中に別のwarpが通信できる

etc

X-Ability
クロスアビリティ

チューニングしたCUDAのコード(cuda_opt.cu)

```
#define blockSize 16
__global__ void _matmul(float *A, float *B, float *C, int N)
{
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int a = N*blockSize*by;  //submatrix adress of Matrix A
  int b = blockSize*bx;
  float tmp = 0.0f;
  for(int i=0; i<N; i+=blockSize){
    __shared__ float As[blockSize][blockSize];
    __shared__ float Bs[blockSize][blockSize];
    As[ty][tx] = A[a + N*ty + tx];
    Bs[ty][tx] = B[b + N*ty + tx];
    __syncthreads();
#pragma unroll
    for(int k=0; k<blockSize; k++){
      tmp += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
    a += blockSize;
    b += blockSize*N;
  }
  int c = N*blockSize*by + blockSize*bx;
  C[c + N*ty + tx] = tmp;
}
```

```
//wrapper for _matmul kernel
void matmul(float *A, float *B, float *C, int N)
{
  float *devA, *devB, *devC;
  cudaMalloc((void**)&devA, sizeof(float)*N*N);
  cudaMalloc((void**)&devB, sizeof(float)*N*N);
  cudaMalloc((void**)&devC, sizeof(float)*N*N);
  cudaMemcpy(devA, A, sizeof(float)*N*N, cudaMemcpyHostToDevice);
  cudaMemcpy(devB, B, sizeof(float)*N*N, cudaMemcpyHostToDevice);

  //kernel execution
  dim3 nThreads(blockSize, blockSize);
  dim3 nBlocks(N/blockSize, N/blockSize);

  _matmul <<< nBlocks, nThreads >>> (devA, devB, devC, N);
  cudaMemcpy(C, devC, sizeof(float)*N*N, cudaMemcpyDeviceToHost);

  cudaFree(devA);
  cudaFree(devB);
  cudaFree(devC);
}
```

コンパイルと実行

$ nvcc –O3 main.cu cuda_opt.cu

$ ./a.out

X-Ability
クロスアビリティ

CUDAのライブラリを使ったコード(cublas.cu)

```
#include <cublas.h>

void matmul(float *A, float *B, float *C, int N)
{
  float *devA, *devB, *devC;

  //cublasInit();
  //Allocate Memory
  cublasAlloc(N*N, sizeof(*A), (void**)&devA);
  cublasAlloc(N*N, sizeof(*B), (void**)&devB);
  cublasAlloc(N*N, sizeof(*C), (void**)&devC);

  //set matrix
  cublasSetMatrix(N, N, sizeof(*A), A, N, devA, N);
  cublasSetMatrix(N, N, sizeof(*B), B, N, devB, N);
```

```
//CALL SGEMM
  cublasSgemm('N', 'N', N, N, N, 1.0f, devA, N,
devB, N, 0.0f, devC, N);

  cublasGetMatrix(N, N, sizeof(*C), devC, N, C, N);

  cublasFree(devA);
  cublasFree(devB);
  cublasFree(devC);
}
```

コンパイルと実行

```
$ nvcc –O3 main.cu cublas.cu -lcublas
$ ./a.out
```

X-Ability
クロスアビリティ

事前計測タイム

	512 x 512	1024 x 1024	2048 x 2048
naïve CPU	121.0	2086	2.37×10^5
naïve CUDA	4.83	30.4	232
CUDA opt	2.69	12.3	76.8
CUBLAS	2.22	7.75	34.8

unit : msec

CPU : Intel Core i7 860 @ 2.80GHz

GPU : NVIDIA Geforce GTX580 (Fermi Core)

naïve CPU : OpenMP 4threads, CUDA opt : 16 x 16 blocked

結論：cublasのようなライブラリがあればそれを使った方がいいが、ない場合は頑張ってチューニングしましょう

X-Ability
クロスアビリティ

チューニングしたOpenCLのコード(ocl_MatMul.cl)

- サンプル“oclMatrixMul”を改良
  - 時間の関係で、あらかじめ置いてあるコードを実行して、「チューニングしたCUDAのコード」と時間比較する
    - h,cpp,cl,Makefileが全て必要です
    - matrixMul_gold.cppはnaïve_cpu.ccと同様、oclMatrixMulの比較用コードです

デモで説明します

X-Ability
クロスアビリティ

ここから座学メイン

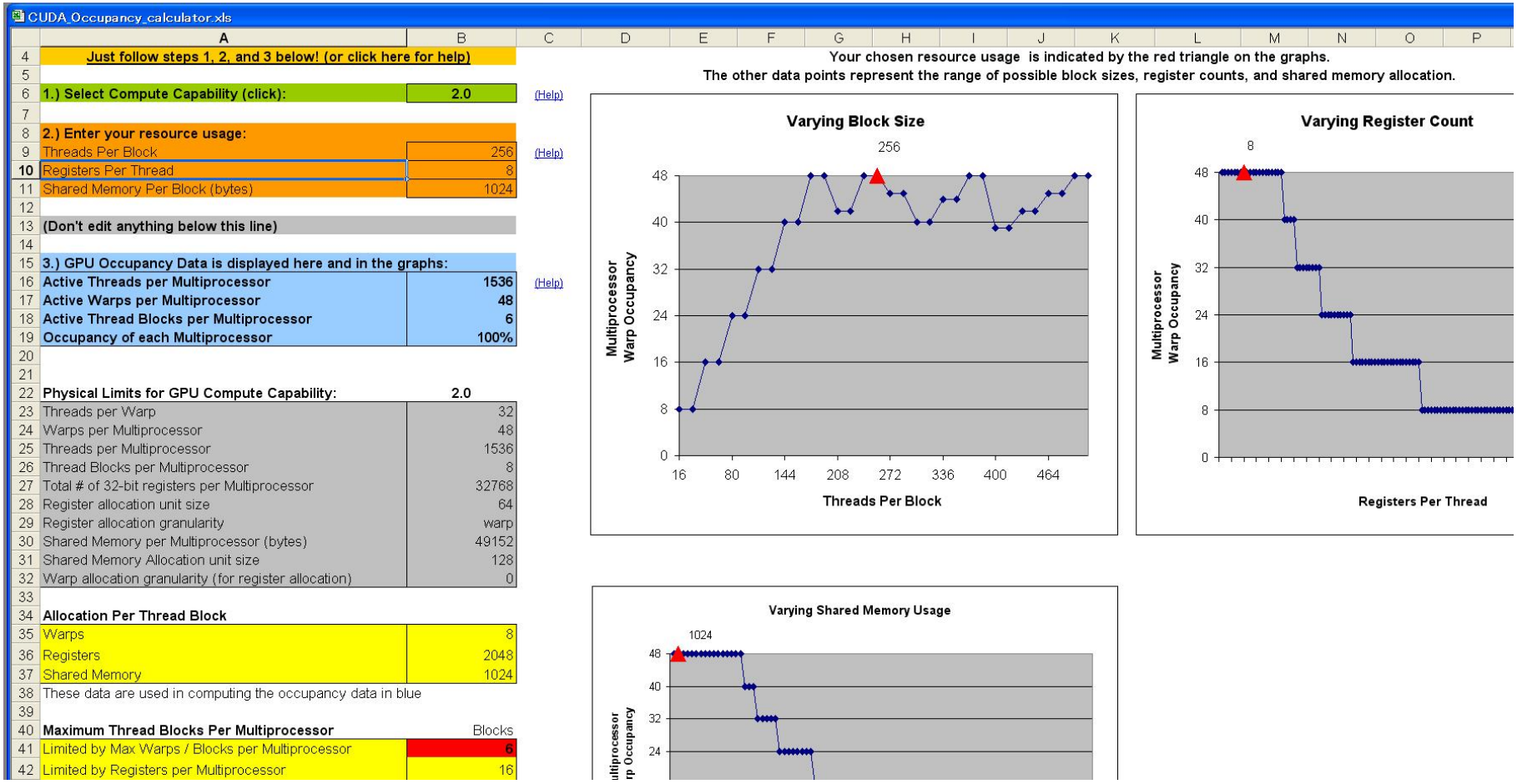
1. CUDA_Occupancy_calculator
2. OpenCLのTips
3. Allinea DDT (debugger)

   GPGPUコーディング入門マシン
4. 応用例：Amber11(MD)、XA-CHEM-SUITE

   XA-CHEM-SUITEの中のXA-CUDA-QMはCUDAで量子化学計算を加速するモジュール

X-Ability
クロスアビリティ

CUDA_Occupancy_calculator（1）



CUDA_Occupancy_calculator.xls

	A	B	C
4	Just follow steps 1, 2, and 3 below! (or click here for help)		
5			
6	1.) Select Compute Capability (click):	2.0	(Help)
7			
8	2.) Enter your resource usage:		
9	Threads Per Block	256	(Help)
10	Registers Per Thread	8	
11	Shared Memory Per Block (bytes)	1024	
12			
13	(Don't edit anything below this line)		
14			
15	3.) GPU Occupancy Data is displayed here and in the graphs:		
16	Active Threads per Multiprocessor	1536	(Help)
17	Active Warps per Multiprocessor	48	
18	Active Thread Blocks per Multiprocessor	6	
19	Occupancy of each Multiprocessor	100%	
20			
21			
22	Physical Limits for GPU Compute Capability:	2.0	
23	Threads per Warp	32	
24	Warps per Multiprocessor	48	
25	Threads per Multiprocessor	1536	
26	Thread Blocks per Multiprocessor	8	
27	Total # of 32-bit registers per Multiprocessor	32768	
28	Register allocation unit size	64	
29	Register allocation granularity	warp	
30	Shared Memory per Multiprocessor (bytes)	49152	
31	Shared Memory Allocation unit size	128	
32	Warp allocation granularity (for register allocation)	0	
33			
34	Allocation Per Thread Block		
35	Warps	8	
36	Registers	2048	
37	Shared Memory	1024	
38	These data are used in computing the occupancy data in blue		
39			
40	Maximum Thread Blocks Per Multiprocessor	Blocks	
41	Limited by Max Warps / Blocks per Multiprocessor	6	
42	Limited by Registers per Multiprocessor	16	

Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

X-Ability
クロスアビリティ

CUDA_Occupancy_calculator（2）

- 以下の3つをいじると何がLimitになっているかを示してくれるエクセルマクロ
  - Threads Per Block
  - Registers Per Thread
  - Shared Memory Per Block (bytes)

  3つのバランスが重要
  - 3つのパラメータはnvcc --ptxas-option=-vでも見れる
- 100%だから最高の速度が出ているとは限らない
  - 対象となるアルゴリズムによる律速があるため（レジスタ使用量が異常に多い、共有メモリを多数必要とする、etc）Occupancyだけで判断はできないが、参考にはなる。

X-Ability
クロスアビリティ

OpenCLの始め方

- CUDA3.1以上のsdkを入れれば入ってる
  - NVIDIAサイトのOpenCL driver & sdkは不要
- 動作Tips
  - まずliboclUtil.a(liboclUtil_x86_64.a)を作成
    - これがないとSDK内のサンプルコードがビルドできない

    cd $OPENCL_SDK/OpenCL/common
    ($OPENCL_SDK : デフォルトで/usr/local/cuda/sdk/OpenCL)
    make
  - その後、例えばoclDeviceQueryを実行

    cd $OPENCL_SDK/OpenCL/src/oclDeviceQuery
    make
    cd $OPENCL_SDK/OpenCL/bin/linux/release
    ./oclDeviceQuery

X-Ability
クロスアビリティ

OpenCLの流れ（１）

1. プラットフォーム取得　clGetPlatFormIDs()
2. デバイス取得　clGetDeviceIDs()
3. コンテキスト作成　clCreateContext()
4. コマンドキュー作成
   clCreateCommandQueue()
5. プログラム作成
   clCreateProgramWithBinary(), or
   clCreateProgramWithSource()
6. カーネル作成　clCreateKernel()

X-Ability
クロスアビリティ

OpenCLの流れ（２）

7. バッファオブジェクト作成　clCreateBuffer()
8. バッファ書込　clEnqueueWriteBuffer()
9. カーネル実行　clEnqueueNDRangeKernal()
10. バッファ読込　clEnqueueReadBuffer()
11. OpenCLオブジェクトリリース

clReleaseKernel(kernel), clReleaseProgram(program),
clReleaseMemObject(memobj),
clReleaseCommandQueue(command_queue),
clReleaseContext(context)

X-Ability
クロスアビリティ

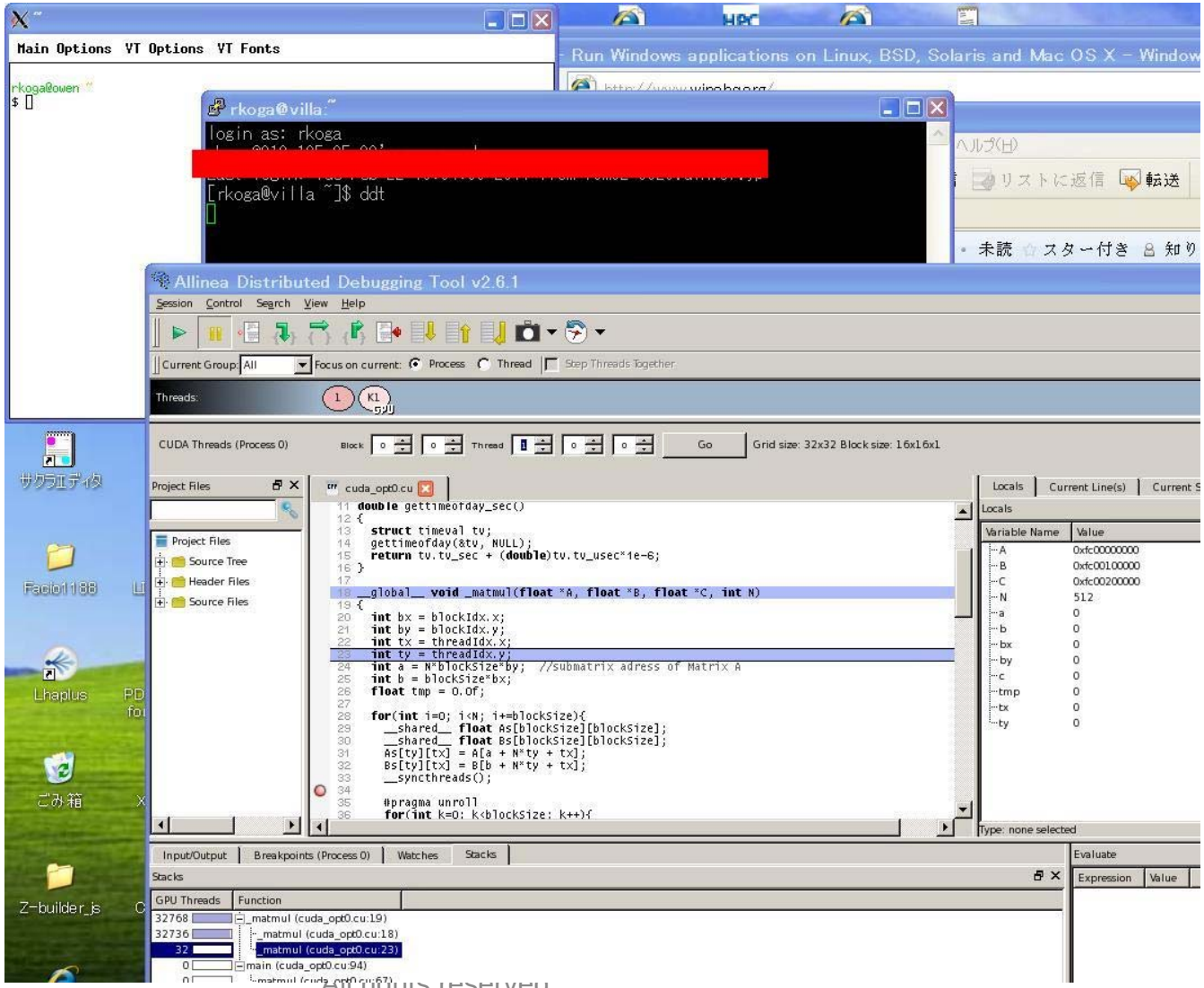
Allinea DDT【デモ】

nvcc –O3 cuda_opt.cu –g -G

直感的なインターフェース

-g -Gのコンパイルオプション必要

※代理店始めました



X-Ability
クロスアビリティ

GPGPUコーディング入門マシン

1. Intel SSE/AVXとCUDAの併用による最高クラスのSIMD高速化コーディングが可能
2. OpenMP/MPIローカル並列マシン環境にてX-terminalによる直感的かつシー ムレスなデバッグが可能
3. 次世代GPGPU言語であるOpenCLによる開発も可能

皆様に実習でお使い頂いたマシンとほぼ同じ（例）

CPU : Intel Zeon 3.33GHz (X5680 6Core)

GPGPU : NVIDIA Tesla C2050 x 2

Compiler : Intel Cluster Studio 2011

Intel Composer Xe, MKL, Intel MPI

CentOS 5.5 + GUI Debugger : Allinea DDT (X-terminal)

X-Ability
クロスアビリティ

XA-CHEM-SUITE

- XA-CUDA-QM

  CUDAで量子化学計算(Quantum Mechanics : QM)を加速するモジュール

- XA-SSE-QM

  SSEで量子化学計算を高速化するモジュール

- XA-AVX-QM

  AVXで量子化学計算を高速化するモジュール

X-Ability
クロスアビリティ

ハートリー・フォック法のプロセス

$$F_{\mu\nu} = H_{\mu\nu}^{core} + \sum_{a}^{2/N}\sum_{\lambda\sigma} C_{\lambda a} C_{\sigma a}^{*} \left[2(\mu\nu|\sigma\lambda) - (\mu\lambda|\sigma\nu) \right]$$

$$= H_{\mu\nu}^{core} + \sum_{\lambda\sigma} P_{\lambda\sigma} \left[(\mu\nu|\sigma\lambda) - \frac{1}{2}(\mu\lambda|\sigma\nu) \right] \quad \text{SCF}$$

$$= H_{\mu\nu}^{core} + G_{\mu\nu}$$

$$(ab|cd) = \int \frac{\chi_a(r)\chi_b(r)\chi_c(r')\chi_d(r')}{|r-r'|} dr' dr$$

Density Matrix : Initial Guessで初期値を作った後、非線形方程式を解いている間（SCFサイクル）アップデートされ続ける。 $F(C)C = SC\varepsilon$

Electron Replusion Integral J-matrix : Coulomb Potential J。理屈では一度計算してメインメモリにおけばいいのだが、$O(N^2)$ のため少し基底Nが大きくなると置けなくなる。ディスクI/Oは時間がかかり毎回演算するのがリーズナブルとなるが大変。

Electron Replusion Integral K-matrix : $O(N^4)$、HF exchange K。J-matrixと同様の問題を抱えるが、使用レジスタの量が多く、メモリ少のSIMD型への実装が難しい。

X-Ability
クロスアビリティ

そもそもGPGPUを使って良い場合

- 下記条件が揃っていて初めて意味がある
  - C/CUDAプログラミングに抵抗がなければハードを買うだけでよい
  - 理論、アルゴリズム、CPUソフトが成熟して高速化が見込めない
    - 苦労して並列化したが最新理論で楽々抜かれた、、、
  - 高速で通信が少ないアルゴリズムが確立されている
    - 計算量が通信量より1～2桁大きい
  - 類似問題がGPUで加速できると分かっている

量子化学計算はこれらの条件を満たしている

X-Ability
クロスアビリティ

実アプリの高速化Tips

先の条件を満たした上で、凄く頑張るのは前提として、、、

- CUDAだけだと対応部分の速度は出るが精度が足りない、メモリ制約があると言われる
  - 結局、倍精度＆メモリ沢山のCPU演算とのハイブリッド
  - ホスト側のプログラムの高速化が重要
    - SSE, AVXのintrinsicを使ってSIMD実装
- CUDAで機能をFull実装するのは、ハードウェアの制約もあり新アルゴリズムを生み出す必要がありかなり大変
  - 結局、これもホスト側とのハイブリッド
- アプリケーションによってはhost-device通信が律速
  - 結局、これもCPU演算とのハイブリッドで、CPU-GPUが同一ダイ上にあるプロセッサ（sandy-bridgeなど）が有効

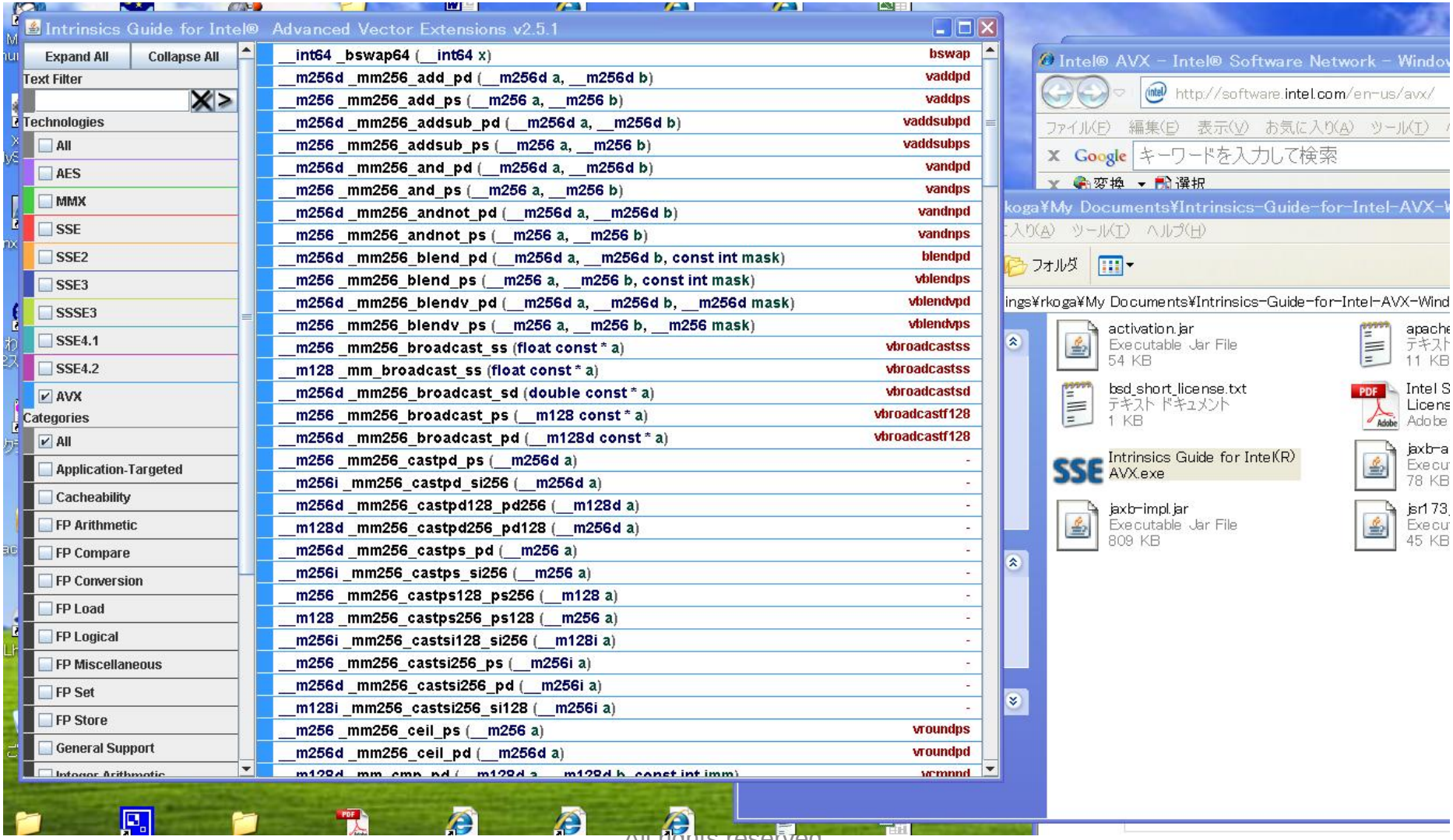
X-Ability
クロスアビリティ

Intel Sandy-bridge

- 2011年2月末現在、チップセットがリコール中（デスクトップは動いてますが、、、）のCPU-GPUハイブリッドプロセッサ
  - 現時点でIntel Composer XEでGPU部分はコンパイルできないようである。言語は未定。
- AVXで256bitの要素をdouble x 4 / float x 8にSIMD的に計算できる
  - SSEの延長
- Intel Intrinsics Guide
  - http://software.intel.com/en-us/avx/
  - Intrinsicを使わないとSSEオプションつけてコンパイルしても十分に最適化されない

X-Ability
クロスアビリティ

AVX Document



X-Ability
クロスアビリティ

受講ありがとうございました

3 Mar 2011