Empowered by Innovation NEC

CUDA Fortran チュートリアル

2010年9月29日

NEC

概要

目的

- CUDA Fortranの利用に関する基本的なノウハウを提供する
  - 本チュートリアル受講後は、Web上で公開されている資料等を参照しながら独力でCUDA Fortranが利用できることが目標

対象

- CUDA Fortranの利用に興味を抱いている方

前提とする知識

- Fortranを用いたLinuxシステム上でのプログラミングに関する基本的な知識
- CUDA Cに関する知識が非常に有用

スケジュール

スケジュール

時間	内容
10:00 ～ 10:30 (30分)	第1部 GPGPU概要
10:30 ～ 10:40 (10分)	第2部 CUDA Fortranの概要
10:40 ～ 11:30 (50分)	第3部 シンプルなコードによる利用例
11:30 ～ 11:50 (20分)	第4部 姫野ベンチマークのCUDA化

第1部
GPGPU概要

第1部の内容

- GPGPUとは？
- 用語集
- GPGPUのHW構造
- GPUの特長
- GPUは何故速いのか？
- 性能を引き出すためのポイント
- 克服すべき課題
- 最新アーキテクチャFermi
- 第1部のまとめ

GPGPUとは？

GPGPU (General Purpose computing on Graphic Processing Unit)

- GPUを汎用計算に用いる技術

CUDA (Compute Unified Device Architecture)

- C言語の拡張およびruntimeライブラリで構成された、GPUで汎用計算を行なうための並列プログラミングモデルおよびソフトウェア環境
- グラフィックAPIを使用せず、C言語およびその拡張での開発が可能
- CUDA Fortranと区別する場合にCUDA Cと呼ぶことがある

CUDA Fortran

- PGI Fortran compilerに含まれる機能であり、Fortranおよびその拡張でのGPUプログラム開発が可能

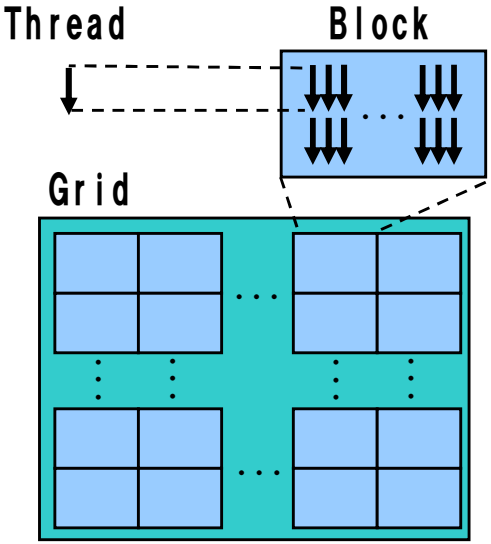
用語集

Host, Device

- CPUが処理を行うサーバ側をhost、GPU側をdeviceとよぶ

Thread, Block, Grid

- CUDAにおける並列処理の階層構造
  - Thread: 最小の処理単位
  - Block: Threadの集合
  - Grid: Blockの集合



Warp

- スケジューリングにおける最小の処理単位
- 1つのstreaming multiprocessor中の8コアでの処理、4 cycle分で32 threadからなる

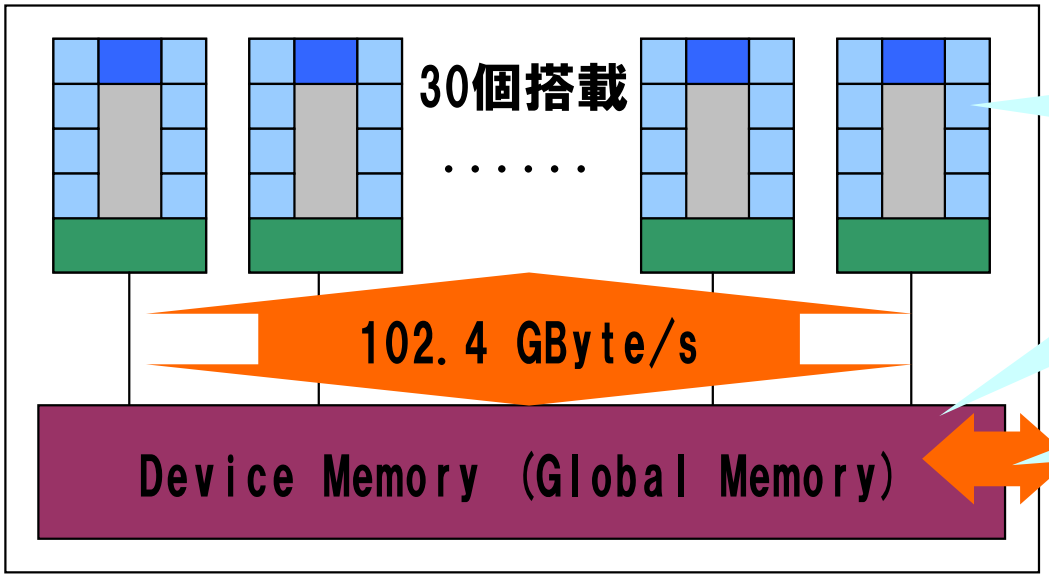
Kernel

- GPU上で動作する関数をkernelとよぶ

GPUのHW構造

最適化の際にはHW構造の把握が必須

- GPUカード全体の構造（Tesla S1070の例）



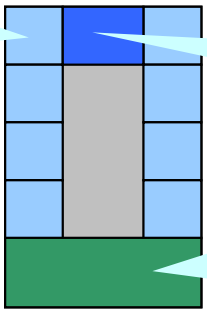
Streaming Multiprocessor (SM)
Tesla S1070では1 GPUあたり
30個搭載されている

Device Memory (Global Memory)
GDDR3 4 GByte
全SPからアクセス可能
PCI経由でここにデータをコピーする

PCI-Express 2.0 x16でhostと接続
バンド幅8 GByte/s

- Streaming Multiprocessorの構造

Streaming Processor (SP)
単精度演算ユニット
SMあたり8個搭載されている



倍精度演算ユニット
SMあたり1個搭載されている

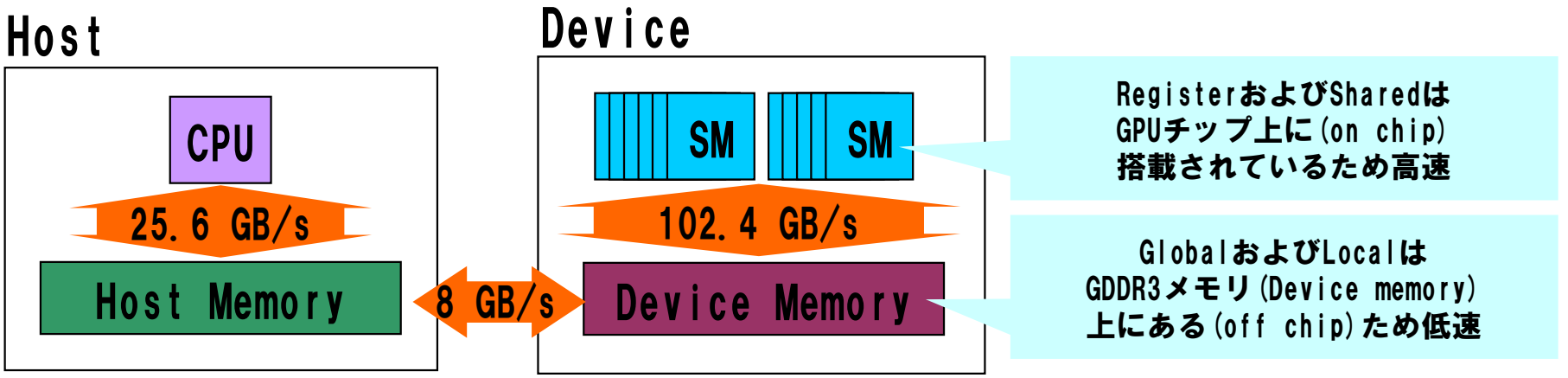
Shared Memory
SM内で共有
レジスタ並みに高速
サイズはSMあたり16 kByte

GPUのHW構造

メモリの階層構造に注意！

- 領域ごとにアクセス速度、scope、lifetimeが異なる

Memory	Location on/off chip	Scope	Lifetime
Register	On	1 thread	Thread
Local	Off		
Shared	On	All threads in block	Block
Global	Off	All threads + host	Host allocation



[出典]NVIDIA CUDA C Programming Best Practice Guide
Table 3.1 Salient features of device memory

GPUの特長

単精度演算のピーク性能が著しく高い！

- CPUとの比較で約11倍
  - NVIDIA Tesla S1070 1 GPU:
    1036.8 GFLOPS (1.04 TFLOPS)
  - Intel Xeon X5570 (2.93 GHz, 4 cores, SSE命令):
    93.76 GFLOPS

Device memoryのバンド幅が大きい

- Hostとの比較で4倍
  - NVIDIA Tesla S1070 1 GPU:
    102.4 GByte/s
  - Intel Xeon X5570, DDR3 1066 MHz:
    25.6 GByte/s

GPUは何故速いのか？

多数の演算コアによる超並列処理

- CPUと比較してより多くのトランジスタを演算器に割り当てている
  - Cacheや制御に割り当てられるトランジスタは少ない
- 多数の演算コアを持つ
  - 例えば、Tesla S1070は1 GPUあたり240個のSPを持つ
  - これらを超並列動作させ高い演算性能を実現
- CUDAは多数のthreadによる並列化を前提としたプログラミングモデルであり、通常とは異なる考え方が必要

[参考]NVIDIA CUDA C Programming Guide
1.1 From Graphics Processing to General-Purpose Parallel Computing

性能を引き出すためのポイント

演算器(SP)を効率的に使用するために多数のthreadを生成する

- CPUにおけるOpenMP等のthread並列化ではコア数と同程度のthread数が妥当だが、GPUではさらに多くのthreadを生成
  - 1つのSMに～1024個のactive threadが生成可能
  - Threadの切り替え時間が極めて短い
  - SMに多数のthreadを割り当てることでメモリアクセス時間を隠蔽できる

メモリの階層構造を意識

- Device memory (global memory), shared memory, registerと、異なる性質を持つメモリ領域を適切に使うことが重要
- Host-device間コピーはPCI経由となり相対的に遅い
  - Host-device間データコピーは最小限に抑える

克服すべき課題

倍精度ではピーク性能が1/12に

- コア(streaming processor)数1/8、コアあたりの演算数2/3で1/12に
  - NVIDIA Tesla S1070 1 GPU: 86.4 GFLOPS

プログラミングが複雑

- 性能を引き出すためにはメモリの階層構造等を意識したプログラミングが必要

Host-device間コピーはPCI経由となり遅い

- 計算に必要なデータをdevice memoryにコピーした後は、データの出し入れなしで計算を行なえるのが理想的
- そのためにはコード全域のCUDA化が必要な場合も

ECCがサポートされていない

- メモリエラー発生による検出困難な結果不正が生じうる
  - Bit errorで結果が変わってしまうことがあるため注意！

克服すべき課題

倍精度ではピーク性能が1/12に

- コア(streaming processor)数1/8、コアあたりの演算数2/3で1/12に
  - NVIDIA Tesla S1070 1 GPU: 86.4 GFLOPS

プログラミング上の難

- 性

最新アーキテクチャFermiでは単精度比1/2に改善
[参考]Next Generation CUDA Architecture. Code Named Fermi
http://www.nvidia.com/object/fermi_architecture.html

Host

- 計算に必要なデータをdevice memoryにコピーした後は、データの出し入れなしで計算を行なえるのが理想的
- そのためにはコード全域のCUDA化が必要な場合も

ECCがサポートされていない

- メモリエラー発生による検出困難な結果不正が生じうる
  - Bit errorで結果が変わってしまうことがあるため注意！

克服すべき課題

倍精度ではピーク性能が1/12に

- コア(streaming processor)数1/8、コアあたりの演算数2/3で1/12に
  - NVIDIA Tesla S1070 1 GPU: 86.4 GFLOPS

プログラミングが複雑

- 性能を引き出すためにはメモリの階層構造等を意識したプログラミングが必要

PGIコンパイラが指示行ベースでのCUDA化をサポートする等、状況は改善しつつある
[参考]PGI | Resources | Accelerator
http://www.pgroup.com/resources/accel.htm

Host ... はPCI経由となり遅い

- 計 ... 後は、
  デ ...
- そ ...

ECCがサポートされていない

- メモリエラー発生による検出困難な結果不正が生じうる
  - Bit errorで結果が変わってしまうことがあるため注意！

克服すべき課題

倍精度ではピーク性能が1/12に

- コア(streaming processor)数1/8、コアあたりの演算数2/3で1/12に
  - NVIDIA Tesla S1070 1 GPU: 86.4 GFLOPS

プログ

- 性能 識した プログラミング

次世代PCIはまだ先.....
プログラミングの工夫で緩和可能な場合が多い

Host-device間コピーはPCI経由となり遅い

- 計算に必要なデータをdevice memoryにコピーした後は、データの出し入れなしで計算を行なえるのが理想的
- そのためにはコード全域のCUDA化が必要な場合も

ECCがサポートされていない

- メモリエラー発生による検出困難な結果不正が生じうる
  - Bit errorで結果が変わってしまうことがあるため注意！

克服すべき課題

- 倍精度ではピーク性能が1/12に
  - コア(streaming processor)数1/8、コアあたりの演算数2/3で1/12に
    - NVIDIA Tesla C1060 1 GPU: 86.4 GFLOPS
- プログラミングが複雑
  - 性能を引き出すためにはメモリの階層構造等を意識したプログラミングが必要
- Host-
  - 計算
    デ
  - そのためには 全域のCUDA化が必要な場合も
- **ECCがサポートされていない**
  - **メモリエラー発生による検出困難な結果不正が生じうる**
    - **Bit errorで結果が変わってしまうことがあるため注意！**

最新アーキテクチャFermiではECCをサポート
[参考]Next Generation CUDA Architecture. Code Named Fermi
http://www.nvidia.com/object/fermi_architecture.html

最新アーキテクチャFermi

NVIDIA GPUの最新アーキテクチャFermi

- これまで課題となっていた多くの事項について改善が行われた
  - 倍精度演算性能の改善
  - ECCのサポート
  - Cacheの追加
  - Shared memoryの増加
  - Etc...

[参考]Next Generation CUDA Architecture. Code Named Fermi
http://www.nvidia.com/object/fermi_architecture.html

第1部のまとめ

GPGPUに関する概要について説明した

- HWの構造について概観
  - Tesla S1070は1 GPUあたり240個の演算コア(SP)を持つ
  - メモリに階層構造があり、領域によってアクセス速度、scope、lifetimeが異なる
- GPUが高速なのは多数のSPによる超並列処理による
- 性能を引き出すためにはCPUとは異なる考え方が必要
  - SMあたり～1024 threadと多数のthreadを生成し、演算コアを効率的に使用する
  - メモリの階層構造を意識したコーディングが必要、特にhost-device間コピーは最小限に抑えなければならない
- 克服すべき課題はいくつかあるが、最新アーキテクチャFermiではそれらの多くが改善された

第2部ではCUDA Fortranの概要について説明する

第2部
CUDA Fortranの概要

第2部 CUDA Fortranの概要

- CUDA Fortran
- CUDA Cとの差異
- CUDA Fortranによるコーディングイメージ
- CUDA Fortranにおけるスレッド並列化
- 移植/最適化の流れ
- 第2部のまとめ

CUDA Fortran

PGI Fortran 10.0以降で利用可能

- 「PGIアクセラレータコンパイラ製品」でなければ利用できないことに注意

CUDA CがC言語の拡張であるのに対し、CUDA FortranはFortranの拡張である

- Fortranで記述されたプログラムをGPU化する際に有用

基本的な考え方はCUDA Cと同じ

- 「Device memoryのallocate」→「Host→Deviceデータコピー」→「Kernelの実行」→「Device→Hostデータコピー」という一連の流れは変わらない
- CとFortranの言語仕様における差異に拠るコーディングの差異があることに注意

CUDA Cとの差異

CUDA Cとの差異は主に以下の通り

- Kernel内で配列が利用可能
  - CUDA Cでは基本的にポインタでの参照のみ
- 属性の指定による記述の簡易化が可能
  - Device上の配列についてはその旨を明示するため、コンパイラがdevice上の配列であることを認識できる
- Fortranは参照渡しであるため、kernelにスカラ値を渡す場合は注意が必要
  - valueで修飾する
- Block IDおよびthread IDが1 origin
  - CUDA Cは0 origin
- APIのインターフェースが一部異なる
  - 省略可能な引数、データサイズの指定単位等
- Texture memoryが使用できない

CUDA Fortranによるコーディングのイメージ

Fortranとその拡張で
コード作成が可能！

通常のFortranコード

サイズNの配列a, bを加算し、配列cにストアする

```
[...]
  call sub(N, a, b, c);
[...]
subroutine sub(n,a,b,c)
  implicit none
  integer :: n
  real,dimension(n) :: a,b,c
  integer :: i
  do i=1,n
    a(i) = b(i) + c(i)
  enddo
  return
end subroutine sub
```

同様の計算をGPUで行なうためのCUDA Fortranコード

```
[...]
  stat = cudaMalloc(d_a,N)
  stat = cudaMalloc(d_b,N)                                    ①
  stat = cudaMalloc(d_c,N)
  stat = cudaMemcpy(d_b,b,N,cudaMemcpyHostToDevice)           ②
  stat = cudaMemcpy(d_c,c,N,cudaMemcpyHostToDevice)
  call sub<<<dimGrid,dimBlock>>>(N,d_a,d_b,d_c)               ③
[...]
  attributes(global) subroutine sub(n,a,b,c)
    implicit none
    integer,value :: n
    real,dimension(n),device :: a,b,c
    integer :: i
    i = (blockidx%x - 1) * blockdim%x + threadidx%x           ④
    if(i < n+1) a(i) = b(i) + c(i)
    return
  end subroutine sub
```

① Device memory（GPUボード上のメモリ）上に配列をallocateする
② Host memoryからdevice memoryにデータをコピーする
③ GPU上で動作する関数をcallする“<<<>>>”で生成するスレッド数等を指定
④ スレッドのIDがループ変数になるようなイメージ

CUDA Fortranによるコーディングイメージ

CUDA Fortranコード

```
[...]
  stat = cudaMalloc(d_a, N)
  stat = cudaMalloc(d_b, N)
  stat = cudaMalloc(d_c, N)
  stat = cudaMemcpy(d_b, b, N, cudaMemcpyHostToDevice)
  stat = cudaMemcpy(d_c, c, N, cudaMemcpyHostToDevice)
  call sub<<<dimGrid, dimBlock>>>(N, d_a, d_b, d_c)
[...]
  attributes(global) subroutine sub(n, a, b, c)
    implicit none
    integer, value :: n
    real, dimension(n), device :: a, b, c
    integer :: i
    i = (blockidx%x - 1) * blockdim%x + threadidx%x
    if(i < n+1) a(i) = b(i) + c(i)
    return
  end subroutine sub
```

device属性でdevice memory上の配列であることを明示するため、device memoryのallocate、単純なhost-Device間コピーを簡潔に記述できる

※ CUDA Cではポインタがhost, device memoryのどちらを指しているかがわからない

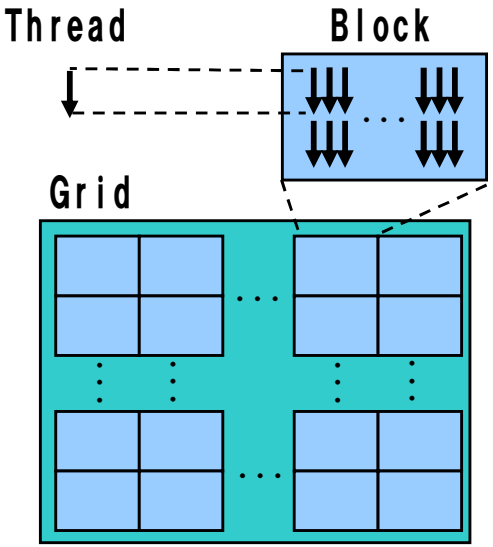
簡潔な記述が可能

```
[...]
  allocate(d_a(N))
  allocate(d_b(N))
  allocate(d_c(N))
  d_b = b
  d_c = c
  call sub<<<dimGrid, dimBlock>>>(N, d_a, d_b, d_c)
[...]
  attributes(global) subroutine sub(n, a, b, c)
    implicit none
    integer, value :: n
    real, dimension(n), device :: a, b, c
    integer :: i
    i = (blockidx%x - 1) * blockdim%x + threadidx%x
    if(i < n+1) a(i) = b(i) + c(i)
    return
  end subroutine sub
```

CUDA Fortranにおけるスレッド並列化

CUDA Cと同様に以下のような階層構造を持つ

- Threadの集合 → Block
  - 3次元配列で表現される
  - 1 block内のthread数には制限がある
    - 最大512 threads
  - BlockがSMに割り当てられる
    - Block内ではthread間の同期が可能
    - Block間の同期はkernel内では不可能
    - Registerおよびshared memory等のリソースに注意



- Blockの集合 → Grid
  - 1つのgridが1つのkernelに対応
  - 2次元配列で表現される
  - 1 Grid内のblock数には制限がある
    - 最大65535x65535x1 blocks

移植/最適化の流れ

CUDA FortranによるGPGPU利用の流れ

- CUDA化の対象とするルーチンを決定する
  - gprof等による高コストルーチンの特定
- 移植
  - Device memoryのallocate
  - Host-device間コピー
  - Gridおよびblock形状の指定
  - Kernelの作成
  - Makeおよび実行
- 性能測定
  - CUDAプロファイラを利用して性能を測定する
- 最適化

第3部ではごくシンプルなコードについて移植～性能測定の実例を示す

第2部のまとめ

- CUDA Fortranの概要について説明した
  - CUDA CがC言語の拡張であるのに対して、CUDA FortranはFortranの拡張
    - Fortranで記述されたコードをGPU化する際に有用
  - プログラミングの方法はCUDA Cとほぼ同様だが、言語仕様に起因する様々な差異があることに注意
  - Thread-Block-Gridという階層構造を意識する
    - Threadが最小単位
    - Threadの集合がBlock、Block単位でSMによる処理が行われる
    - Blockの集合がGrid、GridがKernelに対応
- 第3部ではごくシンプルなコードについて移植～性能測定の実例を示す

第3部
シンプルなコードによる利用例

第3部 シンプルなコードによる利用例

例：配列の加算

- CUDA化後のコード test.cuf
- Device MemoryのAllocate
- API関数のエラーチェック
- Host-Device間コピー
- GridおよびBlock形状の指定
- Kernelの動作確認
- Makeおよび実行

CUDA Profilerによる性能測定

Debug手法

- Blockあたりの使用リソース量確認

第3部のまとめ

例：配列の加算

まずはごく単純なプログラムをCUDA化

- 要素数N(=1000)の配列b, cを加算し配列aにストアする

[Fortranコード test.f90]

```
 1 subroutine sub(n,a,b,c)
 2   implicit none
 3   integer :: n
 4   real,dimension(n) :: a,b,c
 5   integer :: i
 6   do i=1,n
 7     a(i) = b(i) + c(i)
 8   enddo
 9   return
10 end subroutine sub
11
12 program test
13   implicit none
14   integer,parameter :: N = 1000
15   real,dimension(N) :: a,b,c
16   integer :: i
17   b = 1.0e0
18   c = 2.0e0
19   call sub(N,a,b,c)
20   do i=1,N
21     print *, 'i=', i, ',a(', i, ')=', a(i)
22   enddo
23   stop
24 end program test
```

[Makefile]

```
1 FC=pgf90
2 FFLAGS=
3 test: test.f90
4         $(FC) -o $@ $(FFLAGS) $?
5 clean:
6         rm -f test
```

[実行結果]

```
 i=           1 ,a(           1 )=   3.000000
 i=           2 ,a(           2 )=   3.000000
 i=           3 ,a(           3 )=   3.000000
[...]
 i=         998 ,a(         998 )=   3.000000
 i=         999 ,a(         999 )=   3.000000
 i=        1000 ,a(        1000 )=   3.000000
```

サブルーチンsubを
CUDA化する

CUDA化後のコード test.cuf

朱書き部分が変更箇所

[Fortranコード test.cuf]

```
 1 #define NUM_THREADS 512
 2
 3 module cuda_kernel
 4   contains
 5   attributes(global) subroutine sub(n,a,b,c)
 6     implicit none
 7     integer,value :: n
 8     real,dimension(n),device :: a,b,c
 9     integer :: i
10     i = (blockidx%x - 1) * blockdim%x + threadidx%x
11     if(i < n+1) a(i) = b(i) + c(i)
12     return
13   end subroutine sub
14 end module cuda_kernel
15
16 program test
17   use cudafor
18   use cuda_kernel
19   implicit none
20   integer,parameter :: N = 1000
21   real,dimension(N) :: a,b,c
22   real,dimension(:),allocatable,device :: d_a,d_b,d_c
23   integer :: i
24   integer :: stat
25
26   type(dim3) :: dimGrid,dimBlock
27
28   dimGrid = dim3((N-1)/NUM_THREADS+1,1,1)
29   dimBlock = dim3(NUM_THREADS,1,1)
30
31   b = 1.0e0
32   c = 2.0e0
33
34   stat = cudaMalloc(d_a,N)
35   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
36   stat = cudaMalloc(d_b,N)
37   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
38   stat = cudaMalloc(d_c,N)
39   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
40
41   stat = cudaMemcpy(d_b,b,N,cudaMemcpyHostToDevice)
42   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
43   stat = cudaMemcpy(d_c,c,N,cudaMemcpyHostToDevice)
44   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
45
46   call sub<<<dimGrid,dimBlock>>>(N,d_a,d_b,d_c)
47   print *, trim(cudaGetErrorString(cudaGetLastError()))
48
49   stat = cudaMemcpy(a,d_a,N,cudaMemcpyDeviceToHost)
50   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
51
52   do i=1,N
53     print *, 'i=', i, ',a(', i, ')=', a(i)
54   enddo
55
56   stat = cudaFree(d_a)
57   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
58   stat = cudaFree(d_b)
59   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
60   stat = cudaFree(d_c)
61   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
62
63   stop
64 end program test
```

CUDA化後のコード test.cuf

[Fortranコード test.cuf]

```
  1 #define NUM_THREADS 512
  2
  3 module cuda_kernel
  4   contains
  5   attributes(global) subroutine sub(n, a, b, c)
  6     implicit none
  7     integer, value :: n
  8     real, dimensio(n), device ::
  9     integer :: i
 10     i = (blockI      1) * bloc
 11     if(i < n+1)      (i) + c(i)
 12     return
 13   end subrouti
 14 end module cu
 15
 16 p
 17
 18
 19
 20
 21   real, dimension(N) :: a, b, c
 22   real, dimension(:), allocatable, device :: d_a, d_b, d_c
 23   integer :: i
 24   integer :: stat
 25
 26   type(dim3) :: dimGrid, dimBlock
 27
 28   dimGrid = dim3((N-1)/NUM_THREADS+1, 1, 1)
 29   dimBlock = dim3(NUM_THREADS, 1, 1)
 30
```

```
      = 1.0e0
      = 2.0e0
                                                    tring(stat))
                                                    tring(stat))
 38   stat = cudaMalloc(d_c, N)
                                                    n(cudaGetErrorString(stat))
                                                    ostToDevice)
 42   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 43   stat = cudaMemcpy(d_c, c, N, cudaMemcpyHostToDevice)
 44   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 45
      sub<<<dimGrid, dimBlock>>>(N, d_a, d_b, d_c)
       *, trim(cudaGetErrorString(cudaGetLastError()))
      = cudaMemcpy(a, d_a, N, cudaMemcpyDeviceToHost)
      at /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 52   do i=1,N
 53     print *, 'i=', i, ',a(', i, ')=', a(i)
 54   enddo
 55
 56   stat = cudaFree(d_a)
 57   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 58   stat = cudaFree(d_b)
 59   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 60   stat = cudaFree(d_c)
 61   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 62
 63   stop
 64 end program test
```

スレッド数の指定

Kernelはmodule内になければならない

Kernelにはattributes(global)を付加

Fortranは参照渡しであるため
スカラ変数の値を受け取りたい場合は
valueで修飾する

CUDA化後のコード test.cuf

[Fortranコード test.cuf]

```
 1 #define NUM_THREADS 512
 2
 3 module cuda_kernel
 4   contains
 5   attributes(global) subroutine sub
 6     implicit none
 7     integer,value :: n
 8     real,dimension(n),device :: a,b,c
 9     integer :: i
10     i = (blockidx%x - 1) * blockdim%x + threadidx%x
11     if(i < n+1) a(i) = b(i) + c(i)
12     return
13   end subroutine sub
14 end module cuda_kernel
15
16 program test
17   use cudafor
18   use cuda_kernel
19   implicit none
20   integer,parameter :: N = 1000
21   real,dimension(N) :: a,b,c
22   real,dimension(:),allocatable,device :: d_a,d_b,d_c
23   integer :: i
24   integer :: stat
25
26   type(dim3) :: dimGrid,dimBlock
27
28   dimGrid = dim3((N-1)/NUM_THREADS+1,1,1)
29   dimBlock = dim3(NUM_THREADS,1,1)
30
```

```
31   b = 1.0e0
35   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
36   stat = cudaMalloc(d_b,N)
                                                              ))
                                                              ))
41   stat = cudaMemcpy(d_b,b,N,cudaMemcpyHostToDevice)
42   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
                                                    tToDevice)
                                                    udaGetErrorString(stat))
                                                    b,d_c)
47   print *, trim(cudaGetErrorString(cudaGetLastError()))
48
49   stat = cudaMemcpy(a,d_a,N,cudaMemcpyDeviceToHost)
50   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
51
52   do i=1,N
53     print *, 'i=', i, ',a(', i, ')=', a(i)
54   enddo
55
56   stat = cudaFree(d_a)
57   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
58   stat = cudaFree(d_b)
59   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
60   stat = cudaFree(d_c)
61   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
62
63   stop
64 end program test
```

Device memory上の配列はdeviceで修飾

Block ID, thread IDからindexを算出
IDが1 originであることに注意！

配列外(i>n)にアクセスしないようifで制御

CUDA化後のコード test.cuf

[Fortranコード test.cuf]

```
 1 #define NUM_THREADS 512
 2
 3 module cuda_kernel
 4   contains
 5   attributes(global) subroutine sub(n,a,b,c)
 6     implicit none
 7     integer,value :: n
 8     real,dimension(n),device :: a,b,c
 9     integer :: i
10     i = (blockidx%x - 1) * blockdim%x + threadidx%x
11     if(i <= n) c(i) = b(i) + a(i)
12   end subroutine sub
13 end module cuda_kernel
14
15
16 program test
17   use cudafor
18   use cuda_kernel
19   implicit none
20   integer,parameter :: N = 1000
21   real,dimension(N) :: a,b,c
22   real,dimension(:),allocatable,device :: d_a,d_b,d_c
23   integer :: i
24   integer :: stat
25
26   type(dim3) :: dimGrid,dimBlock
27
28   dimGrid = dim3((N-1)/NUM_THREADS+1,1,1)
29   dimBlock = dim3(NUM_THREADS,1,1)
30
31   b = 1.0e0
32   c = 2.0e0
33
34   stat = cudaMalloc(d_a,N)
35   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
36   stat = cudaMalloc(d_b,N)
37   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
38   stat = cudaMalloc(d_c,N)
39   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
40
41   stat = cudaMemcpy(d_b,b,N,cudaMemcpyHostToDevice)
42   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
43   stat = cudaMemcpy(d_c,c,N,cudaMemcpyHostToDevice)
44   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
45
46   call sub<<<dimGrid,dimBlock>>>(N,d_a,d_b,d_c)
47   stat = cudaGetLastError()
48   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
49
50   stat = cudaMemcpy(a,d_a,N,cudaMemcpyDeviceToHost)
51   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
52
53   do i=1,N
54     print *, 'i=', i, ',a(', i, ')=', a(i)
55   enddo
56
57   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
58
59   stat = cudaFree(d_a)
60   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
61   stat = cudaFree(d_b)
62   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
63   stop
64 end program test
```

Moduleを使用するためのuse文
APIを使用するためにcudaforを使用

Device memory上の配列を定義
deviceで修飾する

API関数の返り値を受けるための変数

Grid, blockの形状を指定する際は
定義型dim3を使用する

CUDA化後のコード test.cuf

cudaMallocでdevice memory上に
配列a, b, cの領域をallocate
サイズの指定は要素数で行う
※ CUDA Cではbyte数で指定

定義時にdevice memory上の
配列であることを明示しているため
allocate(d_a(N))
という記述も許される

関数の返り値でエラー検出
返り値がcudaSuccess以外なら
cudaGetErrorStringで
エラー内容を表示

```
28    dimGrid = dim3((N-1)/NUM_THREADS+1, 1, 1)
29    dimBlock = dim3(NUM_THREADS, 1, 1)
30
```

```
1    b = 1.0e0
2    c = 2.0e0
3
4    stat = cudaMalloc(d_a, N)
5    if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
     stat = cudaMalloc(d_b, N)
     if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
     stat = cudaMalloc(d_c, N)
     if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))

     stat = cudaMemcpy(d_b, b, N, cudaMemcpyHostToDevice)
2    if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
3    stat = cudaMemcpy(d_c, c, N, cudaMemcpyHostToDevice)
4    if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
5
6    call sub<<<d                Block>>>(N, d_a, d_b, d_c)
7    print *, t                  String(cudaGetLastError()))
8
9
0
1
2
3
4
5
6
7
58
59
60
61
62
63
64   end program test
```

cudaMemcpyで配列b, cのデータを
device memoryにコピーする
サイズの指定は要素数で行う
※ CUDA Cではbyte数で指定

コピーの方向を指定する
cudaMemcpyHostToDeviceは省略可能

また、d_b=bという記述も許される

CUDA化後のコード test.cuf

[Fortranコード test.cuf]

```
 1 #define NUM_THREADS 512
 2
 3 module cuda_kernel
 4   contains
 5   attributes(global) subroutine sub(n, a, b, c)
 6     implicit none
 7     integer, value :: n
14 end module cuda_kernel
15
16 program test
17   use cudafor
18   use cuda_kernel
19   implicit none
20   integer, parameter :: N = 1000
21   real, dimension(N) :: a, b, c
29   dimBlock = dim3(NUM_THREADS, 1, 1)
30
31   b = 1.0e0
32   c = 2.0e0
33
34   stat = cudaMalloc(d_a, N)
35   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
36   stat = cudaMalloc(d_b, N)
37   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
38   stat = cudaMalloc(d_c, N)
45
46   call sub<<<dimGrid, dimBlock>>>(N, d_a, d_b, d_c)
47   print *, trim(cudaGetErrorString(cudaGetLastError()))
48
49   stat = cudaMemcpy(a, d_a, N, cudaMemcpyDeviceToHost)
50   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
51
52   do i=1,N
53     print *, 'i=', i, ',a(', i, ')=', a(i)
54   enddo
55
56   stat = cudaFree(d_a)
57   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
58   stat = cudaFree(d_b)
59   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
60   stat = cudaFree(d_c)
61   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
62
63   stop
64 end program test
```

cudaGetLastErrorでkernelが正常に実行されたかどうかをチェックする

<<<◇>>>でgrid, block形状を指定しkernelを呼び出す

cudaFreeでdevice memoryを解放
deallocate(d_a)
という記述も許される

Device MemoryのAllocate

朱書き部分が関連箇所

- Deviceメモリ上に配列a, b, cを格納する領域をallocate

```
[...]
 22   real,dimension(:),allocatable,device :: d_a,d_b,d_c
[...]
 34   stat = cudaMalloc(d_a,N)
 35   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 36   stat = cudaMalloc(d_b,N)
 37   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 38   stat = cudaMalloc(d_c,N)
 39   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
[...]
 56   stat = cudaFree(d_a)
 57   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 58   stat = cudaFree(d_b)
 59   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 60   stat = cudaFree(d_c)
 61   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
[...]
```

- integer function cudaMalloc(devptr, count)
  - devptr: 1次元のallocatableなdevice配列
  - count: 配列の要素数
  - ここでは配列a, b, cを格納する領域として実数型N要素の領域を確保
- integer function cudaFree(devptr)
  - devPtr: allocatableなdevice配列

Device MemoryのAllocate

- allocate, deallocateによるdevice memoryの確保、解放が可能

```
[...]
 22   real,dimension(:),allocatable,device :: d_a,d_b,d_c
[...]
 34   stat = cudaMalloc(d_a,N)
 35   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 36   stat = cudaMalloc(d_b,N)
 37   if(stat /= cudaSu                    t *, trim(cudaGetErrorString(stat))
 38   stat = cudaMalloc
 39   if(stat /= cudaSu                    t *, trim(cudaGetErr
[...]
      stat = cu      ee(d
 57                                        trim(cudaGetErr
 58
 60
 61
[...]
```

簡潔な記述が可能

```
[...]
 22   real,dimension(:),allocatable,device :: d_a,d_b,d_c
[...]
 33   allocate(d_a(N))
 34   allocate(d_b(N))
 35   allocate(d_c(N))
[...]
 49   deallocate(d_a)
 50   deallocate(d_b)
 51   deallocate(d_c)
[...]
```

- また、以下のように返り値を取ることもできる

```
allocate(d_a(N),stat=stat)
```

API関数のエラーチェック

朱書き部分が関連箇所

- API関数の返り値（整数型）でエラーチェックを行う

```
[...]
 24   integer :: stat
[...]
 34   stat = cudaMalloc(d_a, N)
 35   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 36   stat = cudaMalloc(d_b, N)
 37   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 38   stat = cudaMalloc(d_c, N)
 39   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
[...]
 56   stat = cudaFree(d_a)
 57   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 58   stat = cudaFree(d_b)
 59   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 60   stat = cudaFree(d_c)
 61   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
[...]
```

- ここでは、正常終了を表すcudaSuccess以外の値が返ってきた場合、cudaGetErrorStringでエラー内容を表示

Host-Device間コピー

朱書き部分が関連箇所

- Device memory上の領域に配列b, cのデータをhost→deviceコピー、配列aの計算結果をdevice→hostコピー

```
[...]
 41   stat = cudaMemcpy(d_b, b, N, cudaMemcpyHostToDevice)
 42   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 43   stat = cudaMemcpy(d_c, c, N, cudaMemcpyHostToDevice)
 44   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 45
 46   call sub<<<dimGrid, dimBlock>>>(N, d_a, d_b, d_c)
 47   print *, trim(cudaGetErrorString(cudaGetLastError()))
 48
 49   stat = cudaMemcpy(a, d_a, N, cudaMemcpyDeviceToHost)
 50   if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
[...]
```

- Integer function cudaMemcpy(dst, src, count, kdir)
  - srcが指す領域からdstが指す領域にcount要素のコピーを行なう
    - countは要素数であることに注意！（CUDA Cではbyte数）
  - kdirでコピーの方向を指定(省略可能)
    - cudaMemcpyHostToDevice: host→deviceのコピー
    - cudaMemcpyDeviceToHost: device→hostのコピー
    - CUDA Fortranでは定義時にdeviceで修飾しdevice memory上の領域であること明示するため、コピーの方向は省略できる

Host-Device間コピー

- 単純なデータコピーについては以下のような簡潔な記述が可能
  - ただし、device→deviceのデータコピーはcudaMemcpyを使う必要がある

```
[...]
 41    stat = cudaMemcpy(d_b, b, N, cudaMemcpyHostToDevice)
 42    if(stat /= cudaSuccess) print *, trim(cudaGetErrorString(stat))
 43    stat = cudaMem      , N, cudaMemcpyHostToDevice)
 44    if(stat /= cud        print *, trim(cudaGetErrorString(stat))
 45
 46    call sub<<<dim        lock>>>(N, d_a, d_
 47    print *    im(        rorString(cudaGe
 48
 50
[
```

簡潔な記述が可能

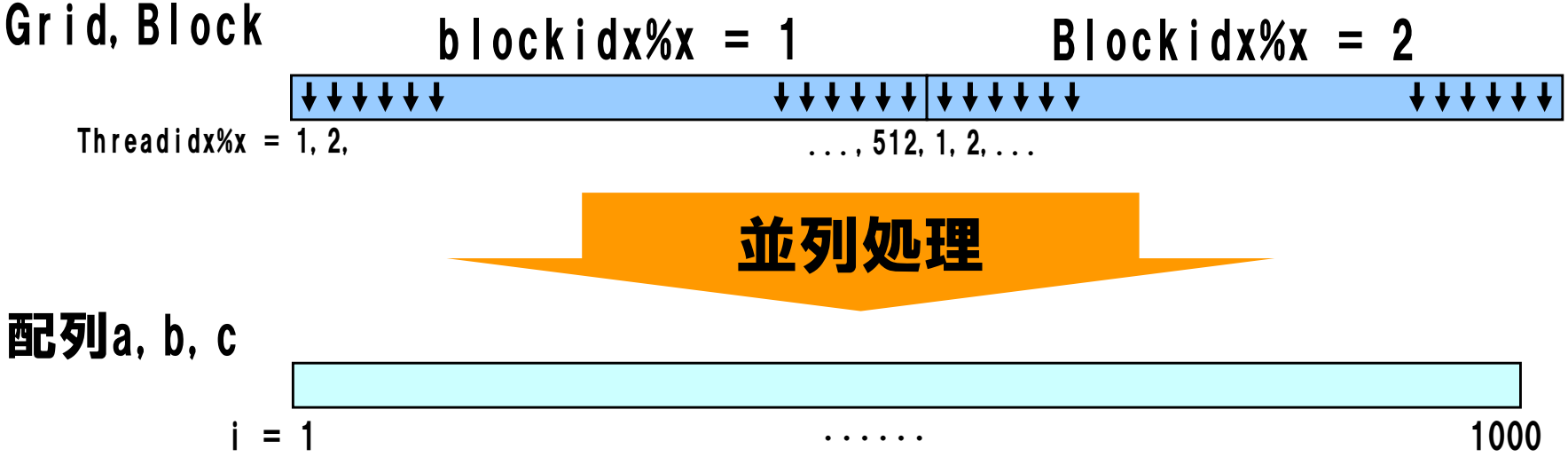
```
[...]
 37    d_b = b
 38    d_c = c
 39
 40    call sub<<<dimGrid, dimBlock>>>(N, d_a, d_b, d_c)
 41    print *, trim(cudaGetErrorString(cudaGetLastError()))
 42
 43    a = d_a
[...]
```

- 詳細は、CUDA Fortran Programming Guide and Reference Chapter 3. Implicit Data Transfer in Expressins参照

GridおよびBlock形状の指定

Gridおよびblock形状のイメージ図

- 512 threadからなる2つのブロックで1000要素の配列を処理する
  - 各blockが順次SMで実行される
  - 各スレッドがwarp(32 thread)単位で実行される
  - 生成されるthread数が要素数より多いので、配列外アクセスを防ぐためのif文が必要
- CUDA Fortranでは、配列のindex、gridおよびblockのIDは1 originであることに注意



GridおよびBlock形状の指定

朱書き部分が関連箇所

- 今回は長さN(=1000)のループを(BLOCK_SIZE,1,1)のblock((N-1)/BLOCK_SIZE+1,1,1)で実行
  - 512 threadsのblock 2つで実行

```
  1 #define NUM_THREADS 512
[...]
 26   type(dim3) :: dimGrid,dimBlock
 27
 28   dimGrid = dim3((N-1)/NUM_THREADS+1,1,1)
 29   dimBlock = dim3(NUM_THREADS,1,1)
[...]
 46   call sub<<<dimGrid,dimBlock>>>(N,d_a,d_b,d_c)
 47   print *, trim(cudaGetErrorString(cudaGetLastError()))
[...]
```

- Gridおよびblockの形状は定義型dim3で指定
  - Block
    - 最大512個のthreadからなる、形状は3次元
  - Grid
    - Blockの集合、形状は実質2次元（3次元目は1しか取れない）
- Kernel呼び出し時に<<<>>>内で指定する

Kernelの動作確認

- Kernelはサブルーチンであるため返り値はないが、以下のようにしてエラーコードを確認できる
  - cudaGetLastErrorで最後のエラーコードを取得
  - cudaGetErrorStringでエラーコードに対応した文字列を取得
    - 正常終了ならno errorが返される

```
  1 #define NUM_THREADS 512
[...]
 26    type(dim3) :: dimGrid,dimBlock
 27
 28    dimGrid = dim3((N-1)/NUM_THREADS+1,1,1)
 29    dimBlock = dim3(NUM_THREADS,1,1)
[...]
 46    call sub<<<dimGrid,dimBlock>>>(N,d_a,d_b,d_c)
 47    print *, trim(cudaGetErrorString(cudaGetLastError()))
[...]
```

Kernelの呼び出し直後にチェックする

Kernelの作成

朱書き部分が関連箇所

- Kernelはmodule内になければならない
- ループ変数iをblock ID (blockidx%x)、thread ID (threadidx%x)およびblockのサイズ(blockdim%x)から算出

[変更前]

```
 1 subroutine sub(n, a, b, c)
 2   implicit none
 3   integer :: n
 4   real, dimension(n) :: a, b, c
 5   integer :: i
 6   do i=1, n
 7     a(i) = b(i) + c(i)
 8   enddo
 9   return
10 end subroutine sub
[...]
```

[変更後]

```
[...]
 3 module cuda_kernel
 4   contains
 5   attributes(global) subroutine sub(n, a, b, c)
 6     implicit none
 7     integer, value :: n
 8     real, dimension(n), device :: a, b, c
 9     integer :: i
10     i = (blockidx%x - 1) * blockdim%x + threadidx%x
11     if(i < n+1) a(i) = b(i) + c(i)
12     return
13   end subroutine sub
14 end module cuda_kernel
[...]
```

IDが1 originであるため、

i = (blockidx%x – 1) * blockdim%x + threadidx%x

となることに注意！

- Thread数512のblockが2個となるので
  - threadidx%x: 1～512
  - blockidx%x: 1～2
  - blockdim%x: 512
  - よってアクセス範囲（iの範囲）は1～1024となる
- 配列外参照を避けるためのif文を挿入

Makeおよび実行

Makeファイルを適宜変更する

```
[Makefile]
  1 FC=pgf90
  2 FFLAGS=-Mpreprocess -Mcuda
  3 test: test.cuf
  4         $(FC) -o $@ $(FFLAGS) $?
  5 clean:
  6         rm -f test *.mod
```

コンパイラオプション
-Mcudaを付加

- コンパイラオプション-Mcudaを付加する

実行結果

cudaGetLastErrorの出力が
no errorだったことを示す

```
[実行結果]
 no error
  i=           1 ,a(           1 )=    3.000000
  i=           2 ,a(           2 )=    3.000000
  i=           3 ,a(           3 )=    3.000000
 [...]
  i=         998 ,a(         998 )=    3.000000
  i=         999 ,a(         999 )=    3.000000
  i=        1000 ,a(        1000 )=    3.000000
```

CUDA Profilerによる性能測定

- プログラム実行時に環境変数CUDA_PROFILE=1を指定することで性能情報が採取できる
  - デフォルトではcuda_profile_0.logに出力される
    - 環境変数CUDA_PROFILE_LOG=[ファイル名]で出力ファイルを指定可能

```
[cuda_profile_0.log]
 1 # CUDA_PROFILE_LOG_VERSION 2.0
 2 # CUDA_DEVICE 0 Tesla T10 Processor
 3 # TIMESTAMPFACTOR fffff719889c9340
 4 method,gputime,cputime,occupancy
 5 method=[ memcpyHtoD ] gputime=[ 6.208 ] cputime=[ 6.000 ]
 6 method=[ memcpyHtoD ] gputime=[ 5.472 ] cputime=[ 6.000 ]
 7 method=[ sub ] gputime=[ 3.200 ] cputime=[ 11.000 ] occupancy=[ 1.000 ]
 8 method=[ memcpyDtoH ] gputime=[ 5.952 ] cputime=[ 28.000 ]
```

- CUDA_PROFILE_CONFIG=[ファイル名]で指定したファイルで採取するデータを選択できる
  - 詳細は以下のURLを参照
    - URL: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/VisualProfiler/computeprof.html

Debug手法

うまく動作しない場合には以下の事項を確認する

- Kernelは動作しているか？
  - cudaGetLastErrorによる確認
- Deviceエミュレーションモードでの実行
  - KernelをCPU上で実行する
  - Make時に-Mcuda=emuオプションを付加する
  - 主にロジックのバグを検出する際に便利
    - print文の使用が可能
  - 実行時間が増大することに注意！
- リソースは不足していないか？
  - Global memoryのサイズ
    - S1070ではカードあたり4 GByteまで
  - Blockあたりのsmemおよびレジスタ数
  - attributes(global)関数の引数
    - 引数はshared memoryを介して渡される、最大256 Byte

Blockあたりの使用リソース量確認1

Kernelのコンパイル時に-Mcuda=keepbinオプションを付加すると foo.???.binというファイルが作成される

```
[test.003.bin]
  1 architecture  {sm_13}
  2 abiversion    {1}
  3 modname       {cubin}
  4 code  {
  5           name = sub
  6           lmem = 0
  7           smem = 48
  8           reg  = 4
  9           bar  = 0
 10           bincode  {
 11                   0x10000205 0x40004780 0xa0000005 0x04000780
[...]
```

- smemはblockあたりのshared memory使用量
  - 16,536 Byte以下でなければならない
- regはthreadあたりのregister使用量
  - Blockあたりの使用量が16,536以下でなければならない
    - ここではthreadあたり4なので512 threadで2,048となり問題なし
  - Make時に-Mcuda=maxregcount:[レジスタ数]を付加すると
    使用レジスタ数を制限できる（ただし、多くの場合性能が低下）

Blockあたりの使用リソース量確認2

Kernelのコンパイル時に-Mcuda=ptxinfoオプションを付加するとコンパイル時メッセージとしてregster, shared memory使用量が出力される

```
$ pgf90 -o test -Mpreprocess -Mcuda=ptxinfo test.cuf
ptxas info    : Compiling entry function 'sub'
ptxas info    : Used 4 registers, 32+16 bytes smem
ptxas info    : Compiling entry function 'sub'
ptxas info    : Used 4 registers, 32+16 bytes smem
```

- smemはblockあたりのshared memory使用量
  - 16,536 Byte以下でなければならない
- regはthreadあたりのregister使用量
  - Blockあたりの使用量が16,536以下でなければならない
    - ここではthreadあたり4なので512 threadで2,048となり問題なし
  - Make時に-Mcuda=maxregcount:[レジスタ数]を付加すると使用レジスタ数を制限できる（ただし、多くの場合性能が低下）

第3部のまとめ

- ごくシンプルなコードについて、CUDA FortranによるGPU利用の例を示した
  - APIを利用し、device memoryを適切に設定する
    - Device memoryのallocate（cudaMalloc）
    - Allocateされた領域に、GPU上で参照されるデータをコピー（cudaMemcpy）
    - 必要に応じてGPU上での計算結果をhost側にコピー（cudaMemcpy）
  - 適切な並列化でkernelを呼び出す
    - 並列化の方法はBlockおよびGridの形状で指定
  - うまく動作しない時はBlockあたりの使用リソース量に注意！
- 第4部では、より実際のコードに近い「姫野ベンチマーク」の移植、最適化について説明する

参考資料

CUDA Fortran関連

- PGI | Resources | CUDA Fortran
  - http://www.pgroup.com/resources/cudafortran.htm
  - CUDA Fortranに関する解説
- CUDA Fortran Programming Guide and Reference
  - http://www.pgroup.com/doc/pgicudaforug.pdf
  - CUDA Fortranに関するプログラミングガイド
    - CUDAに関する基本的な知識があることが前提？
    - CUDA Programming Guideと併せて読む必要がある

参考資料

NVIDIA CUDA関連

- CUDA Zone
  - http://www.nvidia.com/object/cuda_home_new.html
  - NVIDIAによる公式ページ
- NVIDIA GPU Computing Developer Home Page
  - http://developer.nvidia.com/object/gpucomputing.html
  - CUDA Cに関する各種ドキュメントが公開されている
    - CUDA Programming Guide: CUDAプログラミングに関する基本的なドキュメント
    - CUDA Best Practice Guide: CUDAプログラムの最適化手法に関するドキュメント、対象はCUDAで動作するコードを作成したことがある方
    - CUDA Reference Guide: APIに関する網羅的なドキュメント、リファレンスとして参照

その他

- 姫野ベンチマーク
  - http://accc.riken.jp/HPC/HimenoBMT/index.html

Empowered by Innovation