NVIDIA®

CUDA実践エクササイズ

CUDAエクササイズ

- 6つのCUDA実践エクササイズ用に、スケルトンと解答を用意しました。
- 各エクササイズ(5を除く)で、コードの欠けている部分を補ってください。
  - コンパイルし、プログラムを実行して「Correct!」と出力されたら、完了です。
- 解答は、各エクササイズの「solution」フォルダにあります。

コードのコンパイル: Windows

- Microsoft Visual Studioで<プロジェクト>.slnファイルを開く
  - プロジェクトをビルドする
  - 4つの構成オプション
    - Release、Debug、EmuRelease、EmuDebug
- コードをデバッグする場合には、EmuDebug構成でビルド
  - カーネル内にブレイクポイントを設定可能 (__global__または __device__ functions)
  - 通常どおりに、printfでもコードをデバッグ可能
  - 1つのGPUスレッドに対して1つのCPUスレッド
  - 実際にはGPU上ではスレッドは並列でない

コードのコンパイル: Linux

`nvcc <filename>.cu [-o <executable>]`

- リリースモードでビルド

`nvcc -g <filename>.cu`

- デバッグ (デバイス) モードでビルド
- ホストコードはデバックできるが、デバイスコード (GPUで実行) はデバッグできない

`nvcc -deviceemu <filename>.cu`

- デバイスエミュレーションモードでビルド
- デバッグシンボルなし、CPUですべてのコードを実行

`nvcc -deviceemu -g <filename>.cu`

- デバッグデバイスエミュレーションモードでビルド
- デバッグシンボル付き、CPUですべてのコードを実行
- gdbまたはその他のlinuxデバッガを使用してデバッグ

1: ホストとデバイス間でのコピー

- 「cudaMallocAndMemcpy」テンプレートから開始
- Part1: デバイス上にポインタd_aとd_bのメモリを割り当てる
- Part2: ホスト上のh_aをデバイス上のd_aにコピーする
- Part3: d_aからd_bへデバイス間のコピーを実行する
- Part4: デバイス上のd_bをホスト上のh_aにコピーバックする
- Part5: ホスト上のd_aとd_bを解放する
- **ボーナス**: h_aの割り当てに、mallocの代わりにcudaMallocHostを使って実行してみましょう

2: カーネルの起動

- 「myFirstKernel」テンプレートから開始
- Part1: ポインタd_aを使用して、カーネルの結果に対応するデバイスメモリを割り当てる
- Part2: 1-Dスレッドブロックの1-Dグリッドを使用してカーネルを構成して起動する
- Part3: 各スレッドで以下のようにd_aの要素を設定する

```
idx = blockIdx.x*blockDim.x + threadIdx.x
d_a[idx] = 1000*blockIdx.x + threadIdx.x
```

- Part4: d_aの結果をホストのポインタh_aにコピーバックする
- Part5: 結果が正しいことを検証する

3: 配列の反転(単一ブロック)

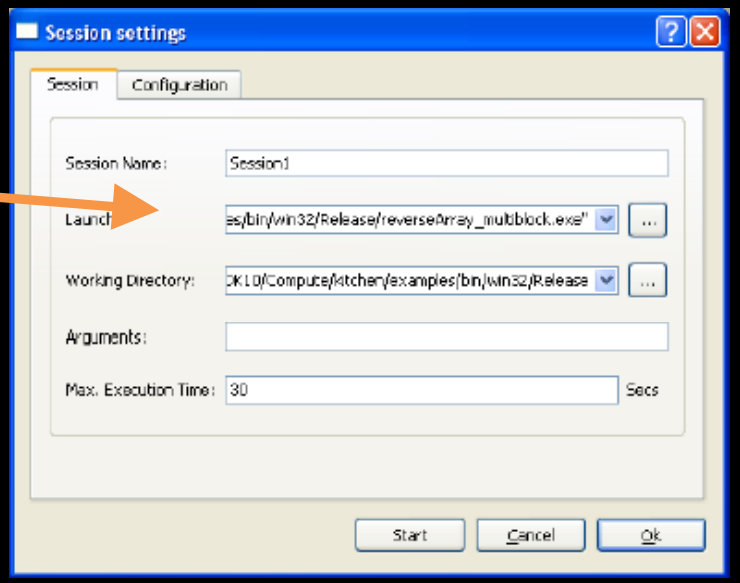
- ポインタd_aに入力配列{a_0, a_1, …, a_{n-1}}がある場合に、ポインタd_bに反転した配列{a_{n-1}, a_{n-2}, …, a_0}を保存する
- 「reverseArray_singleblock」テンプレートから開始
- スレッドブロックを1つだけ起動し、配列のサイズを反転させる
  N = numThreads = 256要素
- Part1：カーネル「greverseArrayBlock()」の本体を実装するのみ
- 各スレッドで単一の要素を反転の位置に移動する
  - d_aポインタから入力を読み込む
  - d_bポインタに位置を反転した出力を保存する

4: 配列の反転(マルチブロック)

- ポインタd_aに入力配列{a_0, a_1,…, a_{n-1}}がある場合に、ポインタd_bに反転した配列{a_{n-1}, a_{n-2}, …, a_0}を保存する
- 「reverseArray_multiblock」テンプレートから開始
- 256スレッドのブロックを複数起動する
  - サイズNの配列を反転するには、N/256ブロック
- Part1: 起動するブロック数を計算する
- Part2: カーネルのreverseArrayBlock()を実装する
- 以下の両方を計算する必要がある
  - ブロック内で反転した位置
  - ブロックの先頭に対する、反転されたオフセット

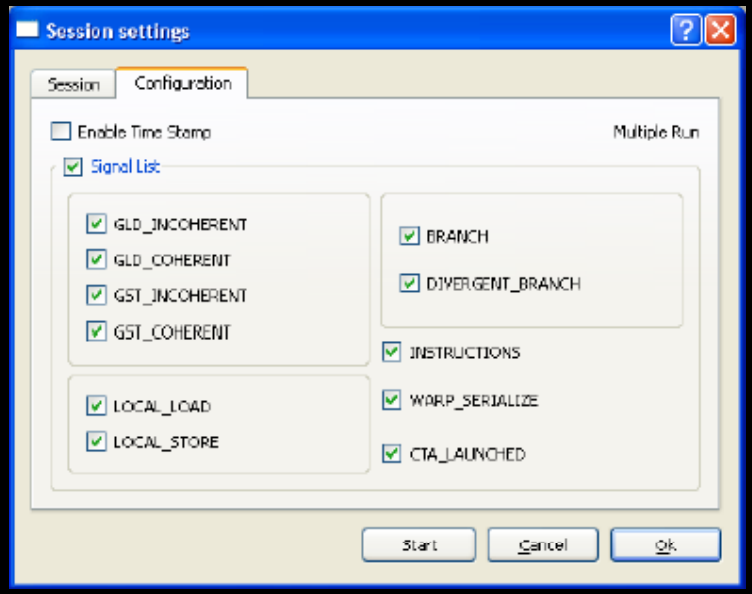
5: 配列の反転のプロファイリング

- 配列の反転にパフォーマンス上の問題がある
- CUDA Visual Profilerを使用して、コンパイル済みのプログラムを実行する
  - リリースモードでコンパイルし、「cudaprof」を実行し、新しいプロジェクトを作成する
  - [Session settings] ダイアログの [Launch] 入力ボックスで、実行可能ファイルを参照する



5: 配列の反転のプロファイリング

- [Configuration] タブをクリックする
- [Signal List] の横にあるチェックボックスをオンにする
- [OK] > [Start] の順にクリックする
- 以下の値をチェックし、0以外でないかを確認する
  - GLD_INCOHERENT
  - GST_INCOHERENT
  - WARP_SERIALIZE
- [GPU Time] を書き留める



6: 配列の反転の最適化

- 目標: インコヒーレントなロード／保存の問題を解決し、パフォーマンスを改善する
  - 共有メモリを使用して各ブロックを反転する
- Part1: 共有メモリのバイト数を計算する
  - 1スレッドあたり1要素
- Part2: カーネルを実装する
  - コメントをヒントに
  - 正しいブロックオフセットの計算を忘れないこと！
- Part3: 作業コードをプロファイリングする
  - GLD/GST_INCOHERENTの値を前と比較する
  - GPU時間を前と比較する

共有メモリでのデータの反転