NVIDIA CUDA
Compute Unified
Device Architecture

プログラミング・ガイド（日本語版）

Version 1.1

3/2/2008

目 次

Chapter 1. CUDAの紹介 1

1.1 データ並列演算デバイスとしてのグラフィック・プロセッサ・ユニット 1

1.2 CUDA: GPU での演算のための新しいアーキテクチャ 3

1.3 本書の構成 6

Chapter 2. プログラミング・モデル 7

2.1 高度なマルチスレッド・コプロセッサ 7

2.2 スレッドの集合 .... エラー! ブックマークが定義されていません。

2.2.1 スレッド・ビロック 7

2.2.2 スレッド・ブロックのグリッド 8

2.3 メモリ・モデル 10

Chapter 3. ハードウェア実装 13

3.1 オンチップ・シェアード・メモリ付 SIMD マルチ・プロセッサのセット 13

3.2 実行モデル 14

3.3 演算能力 15

3.4 マルチ・デバイス 16

3.5 モード・スイッチ 16

Chapter 4. アプリケーション・プログラミング・インターフェイス(API) 17

4.1 C 言語での拡張 17

4.2 言語の拡張 17

4.2.1 関数型修飾子 18

4.2.1.1 __device__ 18

4.2.1.2 __global__ 18

4.2.1.3 __host__ 18

4.2.1.4 制限 .... エラー! ブックマークが定義されていません。

4.2.2 修飾子の変数型 19

4.2.2.1 __device__ 19

4.2.2.2 __constant__ 19
4.2.2.3 __shared__ 19
4.2.2.4 Restrictions .... エラー! ブックマークが定義されていません。
4.2.3 実行コンフィグレーション .... エラー! ブックマークが定義されていません。
4.2.4 組み込み変数 21
4.2.4.1 gridDim 21
4.2.4.2 blockIdx 22
4.2.4.3 blockDim 22
4.2.4.4 threadIdx 22
4.2.4.5 制限 .... エラー! ブックマークが定義されていません。
4.2.5 NVCC を伴うコンパイル 22
4.2.5.1 __noinline__ 22
4.2.5.2 #pragma unroll 23
4.3 共通ランタイム・コンポーネント 23
4.3.1 組み込みベクター型 23
4.3.1.1 char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, float1, float2, float3, float4 23
4.3.1.2 dim3 型 23
4.3.2 数学的関数 .... エラー! ブックマークが定義されていません。
4.3.3 時間関数 24
4.3.4 テクスチャ型 24
4.3.4.1 テクスチャ・レファレンスの宣言 24
4.3.4.2 ランタイム・テクスチャ参照属性 25
4.3.4.3 リニア・メモリ対 CUDA 行列によるテクスチャ 25
4.4 デバイス・ランタイム・コンポーネント 26
4.4.1 数学関数 26
4.4.2 同期の関数 26
4.4.3 型変換関数 27
4.4.4 型キャスト関数 27
4.4.5 テクスチャ関数 27

4.4.5.1 デバイス・メモリからのテクスチャリング 27
4.4.5.2 CUDA 行列からのテクスチャリング 28
4.4.6 原子関数 28
4.5 ホスト・ランタイム・コンポーネント .... エラー! ブックマークが定義されていません。
4.5.1 共通概念 29
4.5.1.1 デバイス .... エラー! ブックマークが定義されていません。
4.5.1.2 メモリ .... エラー! ブックマークが定義されていません。
4.5.1.3 OpenGL 相互運用性 30
4.5.1.4 Direct3D 相互運用性 30
4.5.1.5 コンカレント実行の非同期 31
4.5.2 ランタイム API 32
4.5.2.1 初期化 .... エラー! ブックマークが定義されていません。
4.5.2.2 デバイス管理 32
4.5.2.3 メモリ管理 32
4.5.2.4 ストリーム管理 34
4.5.2.5 イベント管理 34
4.5.2.6 テクスチャ参照管理 35
4.5.2.7 OpenGL 相互運用性 37
4.5.2.8 Direct3D 相互運用性 37
4.5.2.9 デバイス・エミュレーション・モードを使ったデバッグ 37
4.5.3 ドライバ API 39
4.5.3.1 初期化 .... エラー! ブックマークが定義されていません。
4.5.3.2 デバイス管理 39
4.5.3.3 コンテクスト管理 .... エラー! ブックマークが定義されていません。
4.5.3.4 モジュール管理 40
4.5.3.5 実行制御 40
4.5.3.6 メモリ管理 .... エラー! ブックマークが定義されていません。
4.5.3.7 ストリーム管理 42
4.5.3.8 イベント管理 43
4.5.3.9 テクスチャ参照管理 44
4.5.3.10 OpenGL 相互運用性 44

4.5.3.11 Direct3D 相互運用性 44
Chapter 5. 性能ガイドライン 47
5.1 性能命令 47
5.1.1 命令スループット 47
5.1.1.1 演算命令 47
5.1.1.2 フロー命令の制御 48
5.1.1.3 メモリ命令 49
5.1.1.4 同期命令 49
5.1.2 メモリ帯域幅 49
5.1.2.1 グローバル・メモリ .... **エラー! ブックマークが定義されていません。**
5.1.2.2 定数メモリ 55
5.1.2.3 テクスチャ・メモリ 55
5.1.2.4 シェアード・メモリ .... **エラー! ブックマークが定義されていません。**
5.1.2.5 レジスタ .... **エラー! ブックマークが定義されていません。**
5.2 ブロックあたりのスレッドの数 62
5.3 ホストとデバイス間のデータ転送 63
5.4 テクスチャ・フェッチ対グローバルまたは定数メモリ読出し 63
5.5 総合的な性能の最適化戦略 64
Chapter 6. 行列乗算の例 67
6.1 概要 .... **エラー! ブックマークが定義されていません。**
6.2 ソース・コードのリスト 69
6.3 ソース・コード・ウォークスルー 71
6.3.1 Mul() 71
6.3.2 Muld() 71
Appendix A. 技術仕様 .... エラー! ブックマークが定義されていません。
A.1 一般仕様 .... **エラー! ブックマークが定義されていません。**
A.2 標準浮動小数点 74
Appendix B. 数学関数 77
B.1 共通ランタイム・コンポーネント 77
B.2 デバイス・ランタイム・コンポーネント 80
Appendix C. 原子関数 83

C.1 算術関数 エラー! ブックマークが定義されていません。
C.1.1 atomicAdd() 83
C.1.2 atomicSub() 83
C.1.3 atomicExch() 83
C.1.4 atomicMin() 84
C.1.5 atomicMax() 84
C.1.6 atomicInc() 84
C.1.7 atomicDec() 84
C.1.8 atomicCAS() 84
C.2 ビット単位関数 85
C.2.1 atomicAnd() 85
C.2.2 atomicOr() 85
C.2.3 atomicXor() 85
Appendix D. ランタイム API 参照 87
D.1 デバイス管理 87
D.1.1 cudaGetDeviceCount() 87
D.1.2 cudaSetDevice() 87
D.1.3 cudaGetDevice() 87
D.1.4 cudaGetDeviceProperties() 88
D.1.5 cudaChooseDevice() 89
D.2 スレッド管理 89
D.2.1 cudaThreadSynchronize() 89
D.2.2 cudaThreadExit() 89
D.3 ストリーム管理 89
D.3.1 cudaStreamCreate() 89
D.3.2 cudaStreamQuery() 89
D.3.3 cudaStreamSynchronize() 89
D.3.4 cudaStreamDestroy() 89
D.4 イベント管理 エラー! ブックマークが定義されていません。
D.4.1 cudaEventCreate() 90
D.4.2 cudaEventRecord() 90

D.4.3 cudaEventQuery() ... 90
D.4.4 cudaEventSynchronize() ... 90
D.4.5 cudaEventDestroy() ... 90
D.4.6 cudaEventElapsedTime() ... 90
D.5 メモリ管理 ... エラー! ブックマークが定義されていません。
D.5.1 cudaMalloc() ... 91
D.5.2 cudaMallocPitch() ... 91
D.5.3 cudaFree() ... 91
D.5.4 cudaMallocArray() ... 92
D.5.5 cudaFreeArray() ... 92
D.5.6 cudaMallocHost() ... 92
D.5.7 cudaFreeHost() ... 92
D.5.8 cudaMemset() ... 92
D.5.9 cudaMemset2D() ... 92
D.5.10 cudaMemcpy() ... 93
D.5.11 cudaMemcpy2D() ... 93
D.5.12 cudaMemcpyToArray() ... 94
D.5.13 cudaMemcpy2DToArray() ... 94
D.5.14 cudaMemcpyFromArray() ... 95
D.5.15 cudaMemcpy2DFromArray() ... 95
D.5.16 cudaMemcpyArrayToArray() ... 96
D.5.17 cudaMemcpy2DArrayToArray() ... 96
D.5.18 cudaMemcpyToSymbol() ... 96
D.5.19 cudaMemcpyFromSymbol() ... 96
D.5.20 cudaGetSymbolAddress() ... 97
D.5.21 cudaGetSymbolSize() ... 97
D.6 テクスチャ参照管理 ... 97
D.6.1 低レベル API ... 97
D.6.1.1 cudaCreateChannelDesc() ... 97
D.6.1.2 cudaGetChannelDesc() ... 97
D.6.1.3 cudaGetTextureReference() ... 97

D.6.1.4 cudaBindTexture() ... 98
D.6.1.5 cudaBindTextureToArray() ... 98
D.6.1.6 cudaUnbindTexture() ... 98
D.6.1.7 cudaGetTextureAlignmentOffset() ... 98
D.6.2 高レベル API ... 98
D.6.2.1 cudaCreateChannelDesc() ... 98
D.6.2.2 cudaBindTexture() ... 99
D.6.2.3 cudaBindTextureToArray() ... 99
D.6.2.4 cudaUnbindTexture() ... 99
D.7 実行制御 ... エラー! ブックマークが定義されていません。
D.7.1 cudaConfigureCall() ... 100
D.7.2 cudaLaunch() ... 100
D.7.3 cudaSetupArgument() ... 100
D.8 OpenGL 相互運用性 ... 100
D.8.1 cudaGLRegisterBufferObject() ... 100
D.8.2 cudaGLMapBufferObject() ... 101
D.8.3 cudaGLUnmapBufferObject() ... 101
D.8.4 cudaGLUnregisterBufferObject() ... 101
D.9 Direct3D 相互運用性 ... 101
D.9.1 cudaD3D9Begin() ... 101
D.9.2 cudaD3D9End() ... 101
D.9.3 cudaD3D9RegisterVertexBuffer() ... 101
D.9.4 cudaD3D9MapVertexBuffer() ... 101
D.9.5 cudaD3D9UnmapVertexBuffer() ... 102
D.9.6 cudaD3D9UnregisterVertexBuffer() ... 102
D.9.7 cudaD3D9GetDevice() ... 102
D.10 エラーの取り扱い ... 102
D.10.1 cudaGetLastError() ... 102
D.10.2 cudaGetErrorString() ... 102
Appendix E. ドライバ API 参照 ... 103
E.1 初期化 ... エラー! ブックマークが定義されていません。

E.1.1 cuInit() 103
E.2 デバイス管理 103
E.2.1 cuDeviceGetCount() 103
E.2.2 cuDeviceGet() 103
E.2.3 cuDeviceGetName() 103
E.2.4 cuDeviceTotalMem() 104
E.2.5 cuDeviceComputeCapability() 104
E.2.6 cuDeviceGetAttribute() 104
E.2.7 cuDeviceGetProperties() 105
E.3 コンテクスト管理 エラー! ブックマークが定義されていません。
E.3.1 cuCtxCreate() 106
E.3.2 cuCtxAttach() 106
E.3.3 cuCtxDetach() 106
E.3.4 cuCtxGetDevice() 106
E.3.5 cuCtxSynchronize() 106
E.4 モジュール管理 エラー! ブックマークが定義されていません。
E.4.1 cuModuleLoad() 106
E.4.2 cuModuleLoadData() 107
E.4.3 cuModuleLoadFatBinary() 107
E.4.4 cuModuleUnload() 107
E.4.5 cuModuleGetFunction() 107
E.4.6 cuModuleGetGlobal() 107
E.4.7 cuModuleGetTexRef() 108
E.5 ストリーム管理 108
E.5.1 cuStreamCreate() 108
E.5.2 cuStreamQuery() 108
E.5.3 cuStreamSynchronize() 108
E.5.4 cuStreamDestroy() 108
E.6 イベント管理 エラー! ブックマークが定義されていません。
E.6.1 cuEventCreate() 108
E.6.2 cuEventRecord() 108

E.6.3 cuEventQuery() 109
E.6.4 cuEventSynchronize() 109
E.6.5 cuEventDestroy() 109
E.6.6 cuEventElapsedTime() 109
E.7 実行制御 ..... エラー! ブックマークが定義されていません。
E.7.1 cuFuncSetBlockShape() 109
E.7.2 cuFuncSetSharedSize() 110
E.7.3 cuParamSetSize() 110
E.7.4 cuParamSeti() 110
E.7.5 cuParamSetf() 110
E.7.6 cuParamSetv() 110
E.7.7 cuParamSetTexRef() 110
E.7.8 cuLaunch() 111
E.7.9 cuLaunchGrid() 111
E.8 メモリ管理 111
E.8.1 cuMemGetInfo() 111
E.8.2 cuMemAlloc() 111
E.8.3 cuMemAllocPitch() 111
E.8.4 cuMemFree() 112
E.8.5 cuMemAllocHost() 112
E.8.6 cuMemFreeHost() 112
E.8.7 cuMemGetAddressRange() 112
E.8.8 cuArrayCreate() 113
E.8.9 cuArrayGetDescriptor() 114
E.8.10 cuArrayDestroy() 114
E.8.11 cuMemset() 114
E.8.12 cuMemset2D() 114
E.8.13 cuMemcpyHtoD() 115
E.8.14 cuMemcpyDtoH() 115
E.8.15 cuMemcpyDtoD() 115
E.8.16 cuMemcpyDtoA() 116

E.8.17 cuMemcpyAtoD() ... 116
E.8.18 cuMemcpyAtoH() ... 116
E.8.19 cuMemcpyHtoA() ... 116
E.8.20 cuMemcpyAtoA() ... 117
E.8.21 cuMemcpy2D() ... 117
E.9 テクスチ参照管理 ... 119
E.9.1 cuTexRefCreate() ... 119
E.9.2 cuTexRefDestroy() ... 119
E.9.3 cuTexRefSetArray() ... 119
E.9.4 cuTexRefSetAddress() ... 120
E.9.5 cuTexRefSetFormat() ... 120
E.9.6 cuTexRefSetAddressMode() ... 120
E.9.7 cuTexRefSetFilterMode() ... 120
E.9.8 cuTexRefSetFlags() ... 121
E.9.9 cuTexRefGetAddress() ... 121
E.9.10 cuTexRefGetArray() ... 121
E.9.11 cuTexRefGetAddressMode() ... 121
E.9.12 cuTexRefGetFilterMode() ... 121
E.9.13 cuTexRefGetFormat() ... 122
E.9.14 cuTexRefGetFlags() ... 122
E.10 OpenGL 相互運用性 ... 122
E.10.1 cuGLInit() ... 122
E.10.2 cuGLRegisterBufferObject() ... 122
E.10.3 cuGLMapBufferObject() ... 122
E.10.4 cuGLUnmapBufferObject() ... 122
E.10.5 cuGLUnregisterBufferObject() ... 123
E.11 Direct3D 相互運用性 ... 123
E.11.1 cuD3D9Begin() ... 123
E.11.2 cuD3D9End() ... 123
E.11.3 cuD3D9RegisterVertexBuffer() ... 123
E.11.4 cuD3D9MapVertexBuffer() ... 123

E.11.5 cuD3D9UnmapVertexBuffer() 123
E.11.6 cuD3D9UnregisterVertexBuffer() 123
E.11.7 cuD3D9GetDevice() 124
Appendix F. テクスチャ・フェッチ 125
F.1 直近ポイントのサンプリング 126
F.2 リニア・フィルタリング エラー! ブックマークが定義されていません。
F.3 参照テーブル 128

図表リスト

Figure 1-1.　CPUとGPUの浮動小数点演算能力 1
Figure 1-2.　GPUはデータ処理用に多くのトランジスタを割当てられる 2
Figure 1-3.　CUDAのソフトウェア・スタック 3
Figure 1-4.　*ギャザーとスキャッター*のメモリ動作 4
Figure 1-5.　シェアード・メモリはALUにより緊密にデータを持ち込む 5
Figure 2-1.　スレッドの集合 9
Figure 2-2.　メモリ・モデル 11
Figure 3-1.　ハードウェア・モデル 14
Figure 5-1.　結合したグローバル・メモリ・アクセス・パターンの例 52
Figure 5-2.　非結合グローバル・メモリ・パターンの例 53
Figure 5-3.　非結合グローバル・メモリ・アクセス・パターンの例. **エラー! ブックマークが定義されていません。**
Figure 5-4.　バンク競合のシェアード・メモリ・アクセスパターンの例 . **エラー! ブックマークが定義されていません。**
Figure 5-5.　バンク競合のないシェアード・メモリ・アクセスパターン例 59
Figure 5-6.　バンク競合のシェアード・メモリ・アクセス・パターンの例**エラー! ブックマークが定義されていません。**
Figure 5-7.　ブロードキャストのシェアード・メモリ読出しアクセス・パターンの例 61
Figure 6-1.　行列乗法 68

Chapter 1.
CUDA の紹介

1.1 データ並列演算デバイスとしてのグラフィック・プロセッサ・ユニット

わずか数年間の事態で、プログラマブルグラフィックプロセッサユニットはFigure 1-1によって示すように、明確にコンピューティングの主力製品に発展しました。マルチ・コアが非常に高いメモリ帯域幅によって動作されている状態で、今日のGPUはグラフィックスと非グラフィックス処理の両方のための信じられないリソースを提案します。

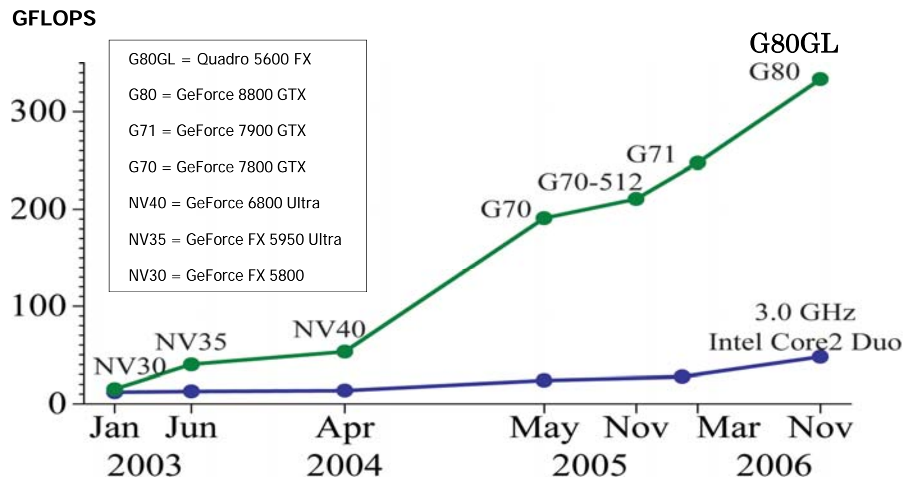


Figure 1-1. CPUとGPUの浮動小数点演算能力

そのような発展の主な背景はGPUが演算集約のために特化されるということです。高度並列演算、ーまさにグラフィックス・レンダリングに対するようにー のために設計されたようなものです。より多くのトランジスタがデータ・キャッシュやフロー制御よりも、図表1-2のようにデータ処理用に専念されています。

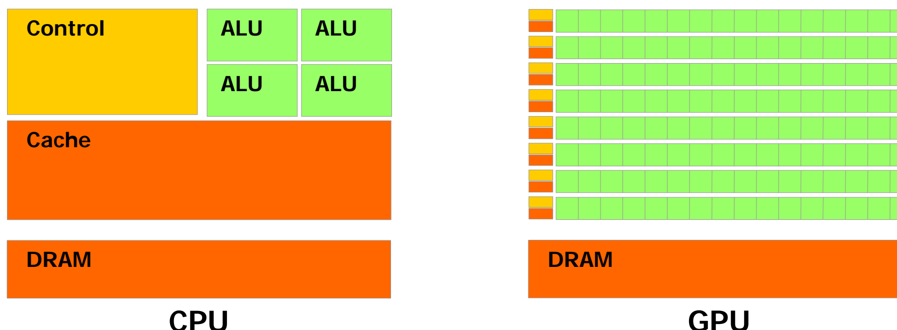


Figure 1-2.　GPUはデータ処理用に多くのトランジスタを割当てられる

GPU のメモリ処理用計算命令に使える面積比率は、特にデータ並列演算を表現するアドレスの問題に適合します。（同一のプログラムは高強度の計算を伴う並列内の多くのデータ要素を実行します）　なぜなら同一プログラムは、高度で洗練されたフロー制御を必要としない各データ要素を実行します。GPUは強力な計算力を有し、処理は多くのデータ要素上で実行されます。メモリ・アクセスの遅延は大きなデータ・キャッシュの代わりの演算能力により表面化しません。

データ並列処理は並列処理スレッドへデータ要素を割り当てます。多くのアプリケーションは行列のような大きなデータ・セットを処理し、演算処理向上のためのデータ並列プログラミング・モデルを活用できます。3 次元のレンダリング用の大きなピクセルと頂点セットは並列スレッドに割り当てられます。レンダリングした画像のポスト・プロセス、ビデオ・エンコード、画像スケーリングや立体視のような、画像とメディア処理アプリケーションに似ています。パターン認識では画像ブロックとピクセルを並列処理スレッドに割り当てることができます。つまり、画像レンダリング・フィールドの外部の多くのアルゴリズムと処理は、一般的な信号処理、物理シミュレーション、財務予測や生物学的計算などをデータ並列処理化により加速されるのです。

ところで、かねてより演算能力はGPU内に潜在的にあったのですが、非グラフィックス・アプリケーションのために効率的に、その演算パワーを扱うのが困難だったのです。

- GPUは初心者への高い学習カーブと不十分なAPIしかない非グラフィックス・アプリケーショを使ってしかプログラミングできなかったのです。
- GPU DRAM の一般的読み込み方法として、GPU プログラムは DRAM のあらゆる部分からデータ要素を集めることができましたが、GPU プログラムは一般的に書き出すkとができませんでした。GPU プログラムは如何なる DRAM の部分へもスキャッタできないなどの、CPU で容易に利用可能な多くのプログラミングの柔軟性が欠けていました。
- いくつかのアプリケーションがGPUの演算能力を利用している時に、DRAM メモリの帯域幅がボトルネックとなっていました。

このドキュメントは、これらの課題についての直接的な回答をすべく、真のジェネリック・データ並列演算デバイスとしての目新しいGPUハードウェアとプログラミングモデルについて記述しています。

1.2 CUDA: GPUでの演算のための新しいアーキテクチャ

CUDA は Compute Unified Device Architecture の省略で、データ並列処理デバイスとして、画像データ割り当てのためのAPIを除く、GPU での演算管理をするための新しいハードウェアとソフトウェア・アーキテクチャです。それは GeForce8 Series、Tesla 及び Quadro で利用可能です(詳細に関しては Appendix A を参照ください)。オペレーティング・システムの多重タスキング・メカニズムは、同時に稼働するいくつかの CUDA とグラフィックス・アプリケーションが GPU へアクセスするのを管理します。

CUDA ソフトウェア・スタックは図表 1-3 のように複数レイヤーから成ります：ハードウェア・ドライバ、API、そのランタイムと2つの上位のレイヤーで共通に使う数学ライブラリ、CUFFT、CUBLAS 等については別のドキュメントに記述しています。このハードウェアは高性能をもたらす軽量なドライバーやランタイム・レイヤーをサポートするように設計されてきました。

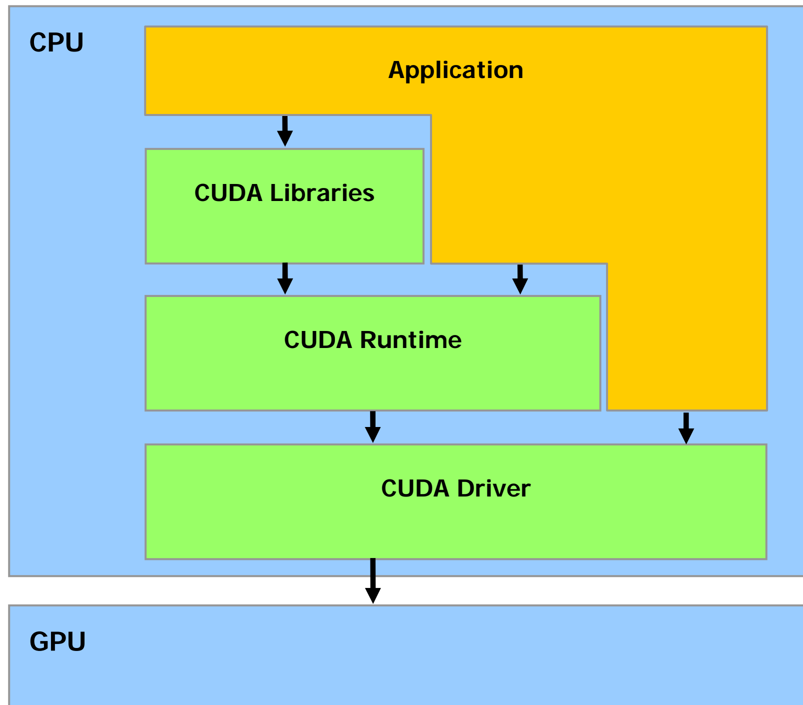


Figure 1-3. CUDAのソフトウェア・スタック

CUDA API は最小のラーニングカーブのために拡張 C プログラミング言語を包含しています(第 4 章を参照してください)。

CUDA は、より多くの柔軟なプログラミングのために図表 1-4 に示すような一般的なスキャッタやギャザー両方の DRAM メモリ・アドレッシングを提供します。プログラミングの見地からは、まさしく CPU などのように、DRAM のどんな位置でもデータを読み書きする転送できます。

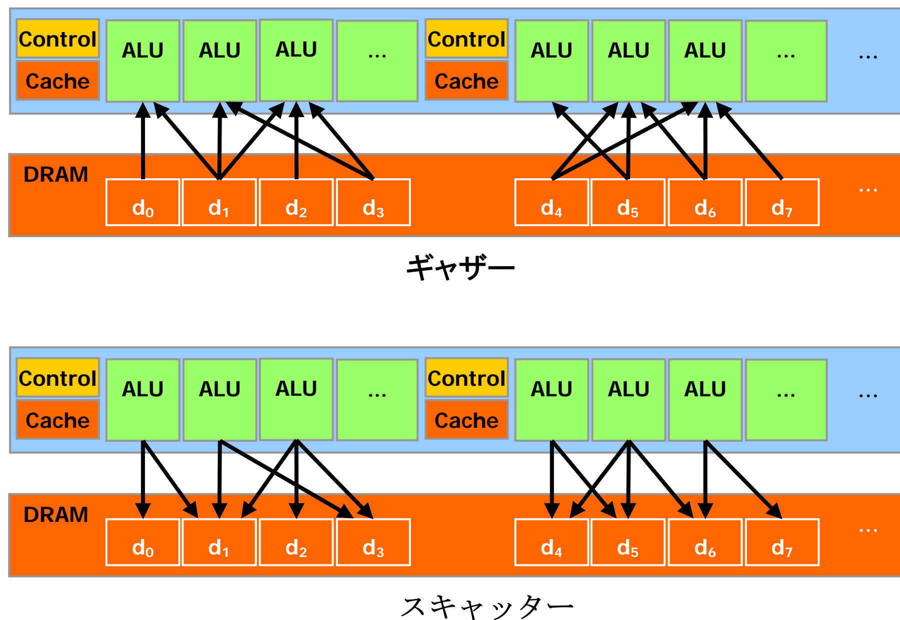
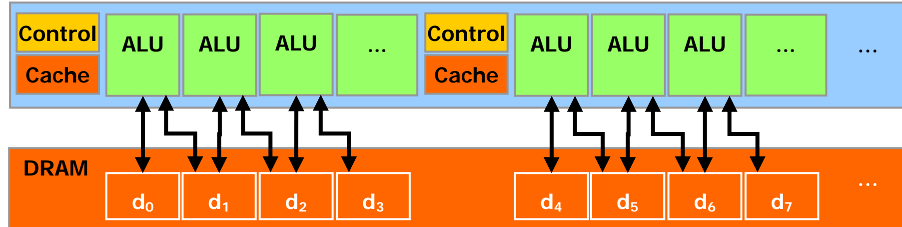
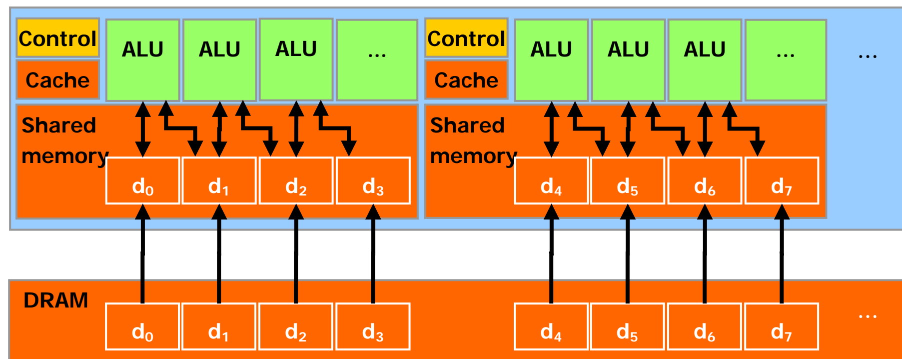


Figure 1-4. *ギャザーとスキャッターのメモリ動作*

CUDAは互いのシェアー・データを使うスレッドが並列データ・キャッシュか一般的オン・チップ・シェアード・メモリをとても高速に読み書きすることを特色とします(第 3 章を参照してください)。図表 1-5 に示すように、アプリケーションはオーバー・フェッチとラウンド・トリップを DRAM 用に最小化します。したがって、DRAM メモリ帯域幅に依存せずに利用することができます。



シェアード・メモリを伴わずに



シェアード・メモリを伴って

Figure 1-5. シェアード・メモリはALUにより緊密にデータを持ち込む

1.3 本書の構成

本書は下記の章から成り立っています

- 第1章 CUDA の紹介.
- 第2章 プログラミング・モデルの概要
- 第3章 ハードウェアの実装の記述
- 第4章 CUDA APIとランタイムの記述
- 第5章 どのように最高性能を引き出すかのガイダンス
- 第6章 幾つかの簡単なサンプル・コードでのウォーキング・スルーによる前章の図解
- **エラー! 参照元が見つかりません。**幾つかのデバイスの技術仕様を示します
- Appendix B CUDA でサポートしている数学演算子のリスト
- Appendix C CUDA でサポートしている原子演算子のリスト
- Appendix D CUDA ランタイム API レファレンス
- Appendix E CUDA ドライバーAPIレファレンス
- Appendix F より詳細なテクスチャ・フェッチ

Chapter 2.
プログラミング・モデル

2.1 高度なマルチスレッド・コプロセッサ

CUDA を通してプログラムされると、GPUは主CPUのコプロセッサ並列スレッドのとても多い数を実行する演算デバイスとみなします。それはメイン CPU かホストのコプロセッサとして作動します。 言い換えれば、ホストで動くアプリケーションが使う並列データ、演算集約的な部分は GPU へ任せます。

幾度も繰り返し実行したアプリケーションの部分で、しかし異なるデータ上の独立した1つの機能に分離できたものは、幾つかの異なるスレッドのように、このデバイス上で実行できる。その趣旨で、デバイスの命令セットにそのような機能をコンパイルします、そして、カーネルと呼ばれる結果としてのプログラムをデバイスにダウンロードします。

ホストとそのデバイスの両方は、それぞれホストメモリとデバイスメモリと呼ばれたそれ自身の DRAM を維持します。 あるデータはデバイスの高性能 Direct Memory Access(DMA)エンジンを活用した API の呼び出しで、1 つの DRAM から他方へデータをコピーすることができます。

.

2.2 スレッドの集合

スレッドの集まりは、Section2.2.1 と 2.2.2 で記述し、また図表 2-1 に表しているスレッド・ブロックのグリッドとして整理されたカーネルを実行します。

2.2.1 スレッド・ブロック

スレッド・ブロックはメモリ・アクセスを調整するために、ある速い共有メモリを通して効率的にデータを共有して、それらの実行を同時にさせることによって同期できるスレッドの集まりです。あるものはカーネルで同期ポイントを指定することができます。そこでは、同期ポイントに達するまで、ブロックのスレッドがサスペンドしています。

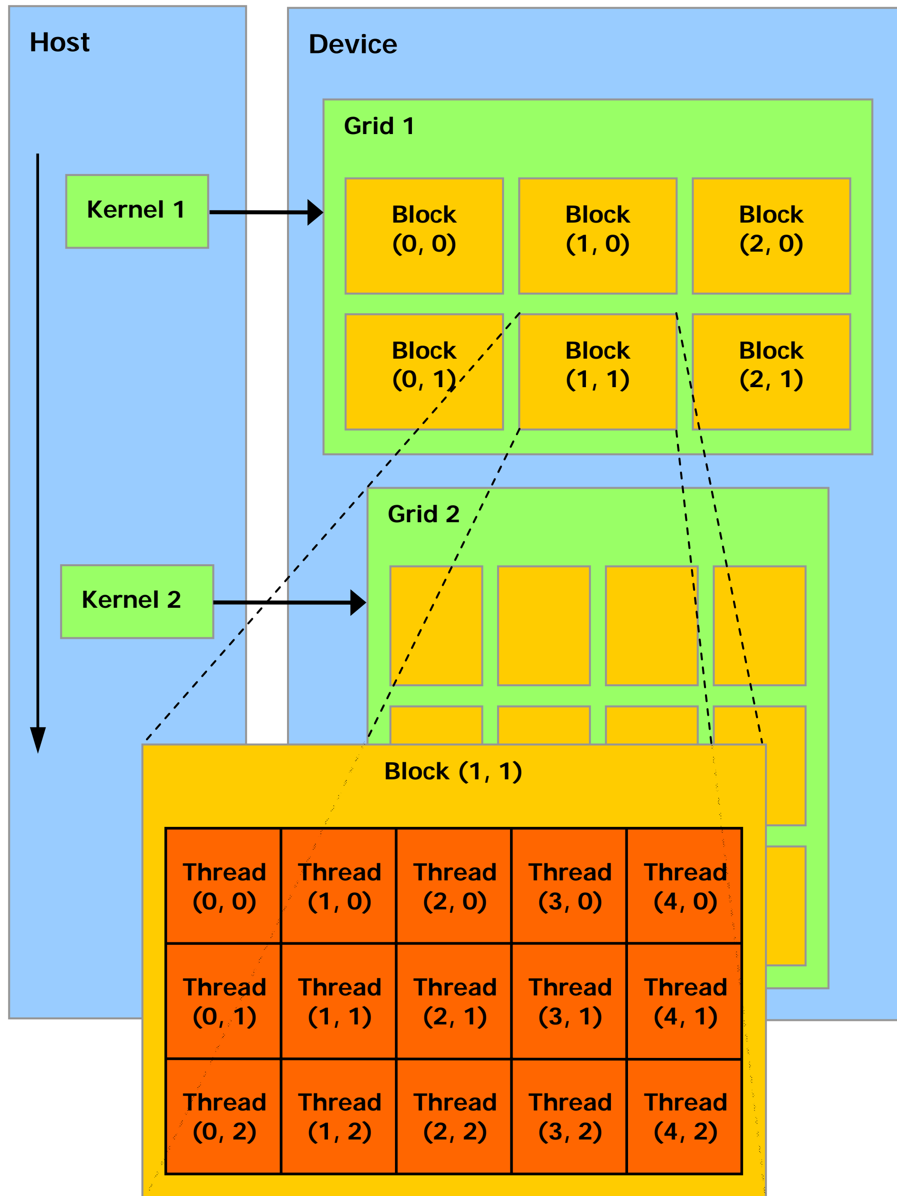
各スレッドはスレッド ID によって特定されます。(それは、ブロックの中のスレッド番号です)。スレッドIDに基づく複雑なアドレッシングを助けるために、アプリケーションは任意にサイズの2か3次元行列としてブロックを指定でき、２－３のコンポーネント・インデックスに代えて使った各スレッドを認識します。 サイズ (D_x, D_y)の 2 次元ブロック用のインデックス(x, y)のスレッドのスレッドIDは

$(x + y D_x)$ですし、サイズ(D_x, D_y, D_z)の３次元ブロック用のインデックス(x, y, z)のスレッドのスレッドIDは$(x + y D_x + z D_x D_y)$です。

2.2.2 スレッド・ブロックのグリッド

ひとつのブロックが持てる最大のスレッド数には制限があります。ところで、同じ次数とサイズのブロックの場合は、複数のブロックから一つのブロックに集めた同じカーネルで実行します。これによる一つのカーネル呼び出しで立ち上げることができるスレッドの総数はとても大きいのです。これは減少したスレッドの協力を犠牲にしています。なぜなら、同じグリッドからの異なるスレッドのブロック内のスレッドは相互の交信と同期ができないからです。このモデルでは異なる並列能力を伴う様々なデバイス上で、リコンパイルなしにカーネルが効率的に動作ができます：そのデバイスに、もしほんの少しか膨大な並列能力、あるいは、その両方があれば通常はグリッドの全てのブロックを連続稼動するかも知れません。

各ブロックはグリッド内のブロック番号であり、それ自身のブロックIDにより認識されます。ブロックIDに基づく複雑なアドレッシングを助けるため、アプリケーションは任意のサイズの 2 次元行列としての一つのグリッドを特定できます。また、2コンポーネント・インデックスに代えて使った各ブロックを認識します。サイズ (D_x, D_y)の 2 次元ブロック用のインデックス(x, y)のブロックのブロックIDは $(x + y D_x)$です。



ホストはデバイスにカーネル呼び出しの継続を発行します。 スレッドの集まりが幾つかのスレッド・ブロックを一つのグリッドとして整理されたので、各カーネルは実行されます。

Figure 2-1.　　スレッドの集合

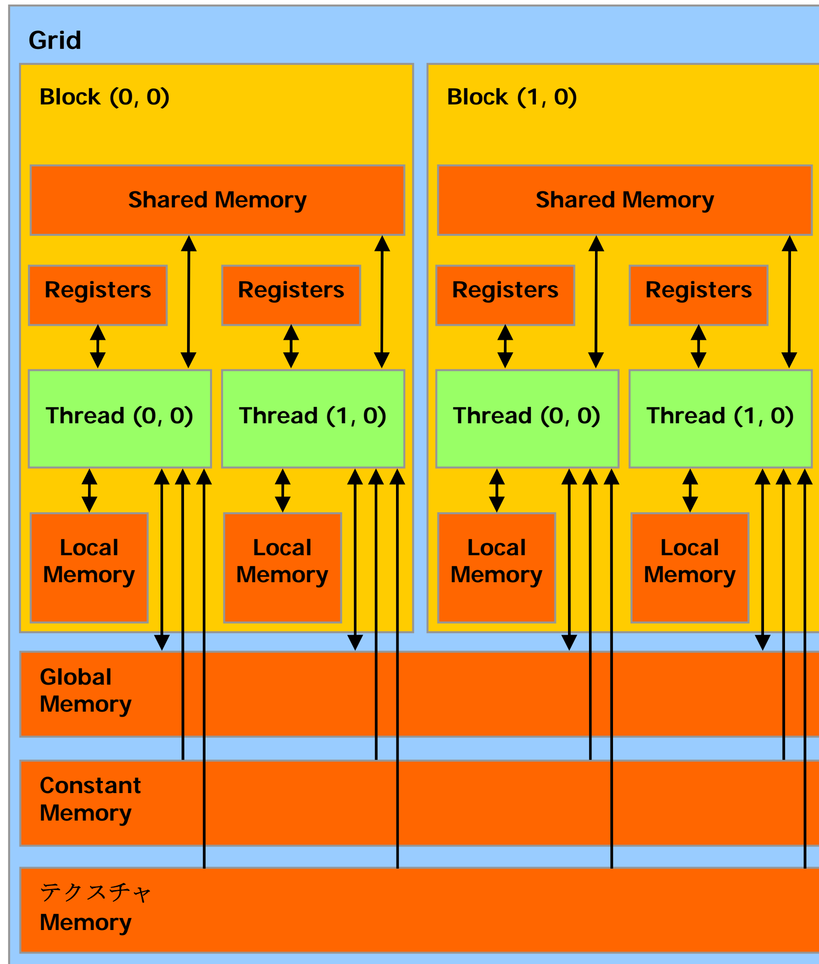
2.3 メモリ・モデル

スレッドは図表 2-2 に示した、デバイス上のそのデバイスが持つDRAMと下記のメモリ空間を経由したオンチップ・メモリへアクセスする手段のみを持っています：

- スレッド・レジスタ毎の読み込み／書き出し
- スレッド・ローカル・メモリ毎の読み込み／書き出し
- シェアード・メモリ・ブロック毎の読み込み／書き出し
- グローバル・メモリ・グリッド毎の読み込み／書き出し
- コンスタント・メモリ・グリッド毎の読み込み／書き出し
- テクスチャ・メモリ・グリッド毎の読み込み／書き出し

グローバル、コンスタント及びテクスチャ・メモリ空間はホストによる読み込み或いは書き出しができます。また、これらは同じアプリケーションにより、永続的にいたるところのカーネルが起動します。

グローバル、コンスタント及びテクスチャ・メモリ空間は異なるメモリ使用量のために最適化されます（図表 5.1.2.1、5.1.2.2 及び 5.1.2.3 を参照下さい）。テクスチャ・メモリはまた、幾つかの特定データ・フォーマット用にデータ・フィルタリングや異なるアドレッシング・モデルを提供します（Section 4.3.4 を参照下さい）。



1つのスレッドは様々な範囲のメモリ空間を経由して、そのデバイス自身ののDRAMとオンチップ・メモリへアクセスします。

Figure 2-2. メモリ・モデル I

Chapter 3.
ハードウェア実装

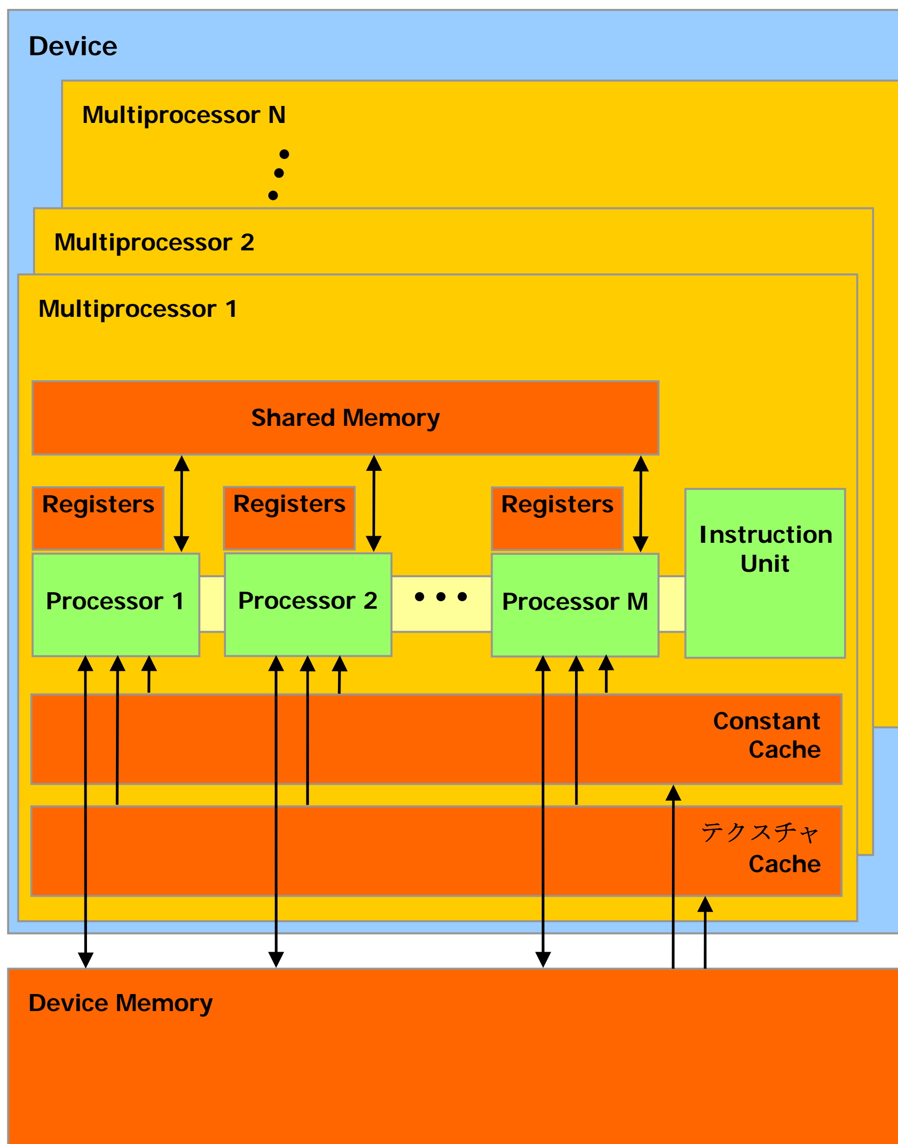
3.1 オンチップ・シェアード・メモリ付きSIMDマルチプロセッサのセット

デバイスは Figure 3-1 に示す *マルチプロセッサ*のセットとして実装されます。各マルチプロセッサは単一命令複データ・アーキテクチャ（SIMD）を持ちます：あらゆる与えられたクロック周期、マルチプロセッサの各プロセッサは、同じ指示を実行しますが、異なったデータを操作します。

各マルチプロセッサは次の４つの型のオン・チップ・メモリを持ちます：

- プロセッサあたりのローカル 32 ビット・*レジスタ*を１セット持ちます、
- 全てのプロセッサでシェアーされ、シェアード・メモリ空間で実装されるパラレル・データ・キャッシュまたは*シェアード・メモリ*
- 全てのプロセッサによりシェアーされ、定数メモリ空間から読出しが高速になるリード・オンリー*定数キャッシュ* でこれはデバイス・メモリのリード・オンリー区域として実装されています、
- 全てのプロセッサによりシェアーされ、テクスチャ・メモリ空間から読出しが高速になるリード・オンリー*テクスチャ・キャッシュ* でこれはデバイス・メモリのリード・オンリー区域として実装されています、

ローカル及びグローバル・メモリ・空間はデバイス・メモリのリード–ライト区域として実装され、キャッシュされません。Section2.3 で言及した様々なアドレッシング・モードとデータ・フィルタリングを与える*テクスチャ・ユニット*で各マルチプロセッサはテクスチャ・キャッシュにアクセスします。



オンチップ・メモリ付SIMDマルチプロセッサのセット

Figure 3-1. ハードウェア・モデル

3.2 実行モデル

スレッド・ブロックの1つのグリッドは、マルチプロセッサ上の実行用スケジューリング・ブロックによりデバイス上で実行されます。

各マルチプロセッサはブロックの集まりの1つのブロックを次々と処理します。1つのブロックは、ただ1つのマルチプロセッサにより処理されます。そしてこれは、とても高速なメモリへ読み込むオンチップ・シェアード・メモリ内にシェアード・メモリ空間を備えています。

各マルチプロセッサが 1 つの集まりで幾つのブロックを処理することができるかは、マルチプロセッサのレジスタと共有メモリがブロックの集まりの全てのスレッドの中で分けられるのでいくつの 1 スレッドあたりのレジスタと、どのくらいの 1 ブロックあたりの共有メモリが与えられたカーネルに必要であるかよります。少なくとも 1 ブロックを処理するために十分なレジスタか、1 マルチプロセッサあたり利用可能な共有メモリがないと、カーネルは起動しないでしょう。

1 つのマルチプロセッサによって 1 つの集まりで処理されるブロックは *アクティブ*であると呼ばれます。それぞれのアクティブなブロックは *ワープ*と呼ばれるスレッドの SIMD グループに分けられます：それぞれのこれらのワープは、*ワープ・サイズ*と呼ばれるスレッドの同じ数を含んでいて、マルチプロセッサにより SIMD 方式で実行されます。

アクティブ・ワープ－例えば、全てのアクティブ・ブロックからの全てのワープ－はタイム・スライスされます：*スレッド・スケジューラ*は、マルチプロセッサのコンピュータのリソースの使用を最大にするために、定期的に 1 ヶ所のワープから別のものに切り替わります。*ハーフ・ワープ*はワープの前半か後半のどちらかです。

ブロックがワープに分けられる方法はいつも同じです； 各ワープは、スレッド0を含む最初のワープを伴うスレッドIDが、インクリメントし、連続するスレッドを含みます。Section2.2 .1 ではスレッドの ID がブロックでどうスレッドのインデックス・リストに関連するかを説明します。

ブロックの中のワープの発行順序は未定義ですが、それらの実行は同時にすることができます、グローバルか共有メモリ・アクセスを調整するために Section2.2.1 で言及します。

スレッド・ブロックのグリッドの中のブロックの発行順序は未定義であり、ブロック間の同期メカニズムが全くないので、同じグリッドの 2 つの異なったブロックからのスレッドはグリッドの実行の間、グローバルなメモリを通して安全に互いに通信することができません。

もし、非アトミック命令が、ワープの1つ以上のスレッド用の、グローバルかシェアード・メモリ内の、同じロケーションに書き込むワープにより実行された時は、それらの出現は未定義ですが、書き込みの1つは成功するように保証されていて、ロケーションと順序へ出現する順番を書き込みます。

もし、アトミック命令（Section 1.11.6 を参照下さい）がグローバル・メモリ内（このグローバル・メモリとは全て順序付けされ、出現したロケーションへワープ、各読み込み、編集、書き込みのうちの1つ以上のスレッドのためのもの）へのワープ読み込み、編集、書き込みにより実行されたら、そこに出現したその順序は未定義になります。

3.3 演算能力

デバイスの演算能力はメジャー・レビジョン番号とマイナー・レビジョン番号により定義されます。

メジャー・レビジョン番号付きデバイスは同じコア・アーキテクチャです。追補Aに掲載したデバイスは全て演算能力 1.x です（それらのメジャー・レビジョン番号は1ですから）。

マイナー・レビジョン番号はコア・アーキテクチャの改訂番号や新機能を含む可能性のあるものに対応しています。

様々な演算能力の技術仕様は追補Aに説明ある方式で与えられます。

3.4 マルチ・デバイス

複数のGPUの使用はマルチプルGPUシステム上の稼動アプリケーションによるCUDAデバイスとして、それらのGPUが同じタイプで動作する時だけ保証されます。

もし、そのシステムがSLIモードの場合は、全てのGPUはドライバー・スタック内の最下位でフューズしますので、1つのGPUしかCUDAデバイスとして使えません。

各GPUを独立したものとして見えるためには、SLIモードをCUDAのためにコントロール・パネルを「オフ」にしておく必要があります。

3.5 モード・スイッチ

GPUは*プライマリ・サーフェス*と呼ばれる幾つかのDRAMメモリに専念します。プライマリ・サーフェスは、ユーザーによる表示出力している際に表示装置のリフレッシュに使います。

ユーザーがディスプレイの解像度やビットの深さの切り替えによりディスプレイの *モード・スイッチ* を起動した際に（NVIDIA コントロール・パネルや Windows のディスプレイ・コントロール・パネルを使って）、相当なメモリをプライマリ・サーフェスの変更用に必要とします。例えば、ユーザーがディスプレイの解像度を from 1280x1024x32-bit to 1600x1200x32-bit へ変更した場合、システムはプライマリ・サーフェス用に 5.24MB より多くの 7.68MB を専念させなくてはなりません。（アンチ・エイリアシングを伴うフル・スクリーン・グラフィックス・アプリケーションの場合は更に多くのディスプレイ・メモリをプライマリ・サーフェスに必要とします。）Windows において、フル・スクリーン DirectX アプリケーションの起動や、コンピュータをロックするための Ctrl+Alt+Del 操作を含むディスプレイ・モードの切り替えの起動という他のイベントをする場合も同様です。

もし、モード・スイッチがプライマリ・サーフェス用に必要な相当量のメモリを増加させるなら、システムはCUDAアプリケーションに専念しているメモリを奪い合わなければなりませんので、結果としてそれらのアプリケーションはクラッシュするかも知れません。

Chapter 4.
アプリケーション・プログラミング・インターフェイス（API）

4.1 C言語での拡張

CUDAプログラミング・インターフェイスの目的は、ユーザーが容易にCに近いプログラミング言語で、デバイスによる実行用のプログラミング記述のための比較的簡単なパスを提供することです。

これは下記から成り立っています：

- 最小限のC言語への拡張セットは Section4.2 に述べています：プログラマはデバイス上で実行するためのソース・コードの部分を対象とします；
- ランタイム・ライブラリは以下へ分割します：
  - ホスト・コンポーネント（Section4.5 に述べています）はホストで稼動し、ホストからの１つ以上の演算デバイスにアクセスし制御する機能を提供します；
  - デバイス・コンポーネント（Section4.4.で述べています）はデバイスで稼動し、デバイスに特化した機能を提供します；
  - 共通コンポーネント（Section4.3 に述べています）はビルト・イン・ベクター型とホストとデバイス・コードの両方でサポートしているC標準ライブラリのサブセットを提供します。

これらは共通ランタイム・コンポーネントにより提供された関数であり、デバイスで稼動するのにサポートしている、C標準ライブラリからの関数のみに重視されるべきです。

4.2 言語の拡張

Cプログラミング言語の拡張は下記の４つです：

- ホストまたはデバイスで実行するか無関係に、さらにホストまたはデバイスから呼び出せるかに関係なく指定する関数型修飾詞（Section4.2.1）；
- 変数のデバイス上でのメモリ・ロケーションを指定する変数型（Section4.2.2）；

- カーネルがホストからのデバイス上でどのように稼動するかを指定する新ディテクティブ（Section4.2.3）;
- グリッド、ブロックの次数、ブロック及びスレッド・インデックスの4つの組み込み変数（Section4.2.4）

これらの拡張子を含む各ソース・ファイルは、Section4.2.5 に概略の述べている CUDA コンパイラ nvcc を伴ってコンパイルしなければなりません。nvcc の詳細な記述は別のドキュメントで読むことができます。

これらの拡張子のそれぞれは、以下の Section 毎に制限を記述しています。nvcc はこれらの制限の同じ警告上のエラーまたはワーニングを与えるでしょうが、それらのいくつかは発覚できません。

4.2.1 関数型修飾子

4.2.1.1 __device__

device 修飾子は次の機能を宣言します:

- デバイスでの実行、
- デバイスからのみ呼び出し可能。

4.2.1.2 __global__

global 修飾子は存在としてのカーネルの機能を宣言します。その機能とは;

- デバイスでの実行、
- ホストからのみ呼び出し可能。

4.2.1.3 __host__

host 修飾子は次の機能を宣言します;

- デバイスでの実行、
- ホストからのみ呼び出し可能。

それは_host_ 修飾子のみを伴う機能を宣言するか、またはあらゆる_host_, _device_か _global_修飾子を伴わないのと等価なものです。いずれの場合の機能もホストだけのためにコンパイルされます。

ところで、_host_修飾子は_device_修飾子と組み合わせで使うことができます。この場合の関数はホストとデバイスの両方に対してコンパイルされます。

4.2.1.4 制限

device と _global_ 関数は帰納をサポートしません。

device と _global_ 関数はそれらの本体内の静的変数を宣言できません。

device と _global_関数は引数の変数番号を保有することはできません。

device 関数はそれらのアドレスを持つことはできません。他方で_global_ 関数への関数ポインターはサポートされます。

global と _host_ 修飾子は一緒に使用できません。

global 関数はボイド復帰型を持っていなければなりません。

global 関数を呼ぶあらゆるものは、Section4.2.3 に述べている実行コンフィギュレーションを指定しなければなりません。

global 関数を呼ぶものは、デバイスが実行の完了を意味する非同期です。.

global 関数パラメータは現在デバイスへのシェアード・メモリを経由して送られ、256 バイトの制限があります。

4.2.2 修飾子の変数型

4.2.2.1 __device__

device 修飾子は変数がデバイスに存在することを宣言します。

他の型の修飾子は、変数が属するメモリ空間で更に指定し_device_を伴って使う際に、次の3つの Section で定義します。もし、それらの表示がないなら、変数は:

- グローバル・メモリ空間に存在しています、
- アプリケーションのライフタイムを保有しています、
- グリッド内部の全てのスレッドとランタイム・ライブラリを経由してホストからからアクセスできます。

4.2.2.2 __constant__

constant 修飾子は_device_をオプションとして一緒に使って変数を宣言します:

- メモリ空間定数に存在しています、
- アプリケーションのライフタイムを保有しています、
- グリッド内部の全てのスレッドとランタイム・ライブラリを経由してホストからからアクセスできます。

4.2.2.3 __shared__

shared 修飾子は_device_をオプションとして一緒に使って変数を宣言します:

- スレッド・ブロックのシェアード・メモリ空間に存在します、
- ブロックのライフ・タイムを保有します、
- ブロック内部の全てのスレッドからのみアクセスできます。

これらはスレッドを渡る弛緩順序付けであっても、スレッド内部のシェアード変数の完全順次整合性です。_syncthreads()(Section4.4.2)の実行後でのみ、見えるように保証した他のスレッドから書き込みます。変数が揮発性として宣言でもしない限り、コンパイラは、読み出しを最適化するのに自由であり、前の宣言が満たされる限りシェアード・メモリに書き込みます:

変数を次の外部行列のようにシェアード・メモリ内で宣言している際に

```
extern _shared_ float shared[];
```

行列サイズは起動時に決定しています(Section4.2.3)。全ての変数はメモリの同じアドレスで開始し、この方式で宣言します。

行列内の変数の配置はオフセットを経由して明示的に管理しなければなりません。例えば、もしそれが以下の方程式を望んだ場合

```
short array0[128];
float array1[64];
int   array2[256];
```

ダイナミックに割り当てたシェアード・メモリでは、それは以下の方法で行列を宣言して初期化するかも知れません。

```
extern _shared_ char array[];
_device_ void func()      // _device_ or _global_ function
{
    short* array0 = (short*)array;
    float* array1 = (float*)&array0[128];
    int*   array2 =  (int*)&array1[64];
}
```

4.2.2.4 制限

これらの修飾子は **struct** と **union** メンバーで、正式なパラメータで、そしてホストで実行する関数内部のローカル変数では許可されません。

shared と **_constant_** 変数は暗黙の静的ストレージを保有します。

device, **_shared_** 及び **_constant_** 変数は **extern** キーワードを使った外部として宣言できません。

device と **_constant_** 変数はファイル有効範囲でのみ許可されます。

constant 変数はホスト・ランタイム関数を経由してのホストからのみとデバイスから割り当てできません（Section4.5.2.3 と 4.5.3.6）。

shared 変数はそれらの宣言子の一部として初期化を持つことができません。

一般的にこれらのあらゆる修飾子がなくとも、デバイス・コードで宣言された自動変数は、レジスタにあります。また一方、幾つかの場合コンパイラはローカル・メモリにそれを置くかも知れません。これはしばしば、余りに多くのレジスタ空間を費やす大きな構造か行列やコンパイラが定数量付きインデックスを決定できない行列を示します。

ptx アッセンブリ・コードの検査（**t-ptx** か**-keep** 付きコンパイリングにより取得したもの）は、**ld.local** と **st.local** ニーモニックを使用して変数が宣言されますから、最初のコンパイル段階の間 **local** ニーモニックを使用することでローカル・メモリに置かれ、アクセスできるかどうかを伝えるでしょう。

もっとも、それが対象としたアーキテクチャのために、余りに多くのレジスター空間を費やすことが判明したなら、その後のコンパイル段階はそのまま他の方法を決めるかも知れません。これはローカル・メモリ使用量（**lmem**）を報告する**--ptxas-options=-v** オプションでコンパイルすることによりチェックできます。

デバイスで実行するコードのポインターは、コンパイラがそれらをシェアード・メモリ空間をしているか否かに関係なく、解決することができ、グローバルなメモリ空間であるかぎりサポートされます。さもなければ、それらはグローバル・メモリ空間で割り当てるか、または宣言するメモリを示すだけのために制限されます。

コード内のグローバルかシェアード・メモリに修飾子参照するポインターは、大抵は分離の失敗かアプリケーションの終了で、未定義の挙動内のデバイス結果を実行した、ホストかコード内のホスト・メモリで実行します。a **_device_**, **_shared_** か **_constant_**変数のアドレスで取得したアドレスはデバイス・コードでのみ使用することができます。その**_device_** か **_constant_**変数は

Section4.5.2.3 で述べている **cudaGetSymbolAddress()**経由して取得したアドレスはホスト・コードでのみ使用できます。

4.2.3 実行コンフィグレーション

__global__関数を呼ぶすべてのものは、それを呼び出すために*実行コンフィグレーション*を指定しなければなりません。

実行コンフィグレーションは、デバイスで実行する関数を使うのにグリッドとブロックの次数を定義します。同様にや関連するストリーム（Section4.5.1.5 にストリームについて述べています）でも。それは、挿入している<<< **Dg, Db, Ns, S** >>>の関数名から括弧内の引数の間からくる式によって指定されます。ここで：

- **Dim3**（Section4.3.1.2 を参照下さい）型には **Dg** があり、起動開始ブロックの数と **Dg.x * Dg.y** が同等なようにグリッドのサイズと次数を指定します。**Dg.z** は未使用です；
- **Dim3**（Section4.3.1.2 を参照下さい）型には **Db** があり、ブロック当りのスレッド数と **Db.x * Db..y * Db.z** が同等なように各ブロックのサイズと次数を指定します；
- **size_t** 型には **Ns** があり、静的に割り当てられたメモリに加えるこの呼び出しのために、ブロック毎に動的に割り当てるシェアード・メモリ内のバイト数を指定します。この動的に割り当てられたメモリは、Section4.2.2.3 に言及している外部行列として宣言した、変数の全てによって使用されたものです；**Ns** は0をデフォルトとするオプション引数です；
- **cudaStream_t** には **S** があり、関連するストリームを指定します。**S** は0をデフォルトとするオプション引数です。

宣言された関数の例

```
__global__ void Func(float* parameter);
```

このように呼び出さなくてはなりません：

```
Func<<< Dg, Db, Ns >>>(parameter);
```

実行コンフィグレーションのための引数は実関数引数の前に関数引数のように評価され、現状ではそれはデバイスへのシェアード・メモリ経由でパスされます。

関数呼び出しは、もし **Dg** か **Db** が Appendix A.1 にて指定したデバイス用に許された最大サイズより大きいか、または **Ns** が静的割り当て、関数引数や実行コンフィグレーションに必要なシェアード・メモリの容量を差し引いたデバイスで可能なシェアード・メモリの最大容量より大きいと失敗します。

4.2.4 組み込み変数

4.2.4.1 gridDim

この変数は **dim3** 型（Section4.3.1.2 を参照）で、グリッドの次数を含んでいます。

.

4.2.4.2 blockIdx

この変数は **uint3** 型（Section4.3.1.3 を参照）で、グリッド内部のブロック・インデックスを含んでいます。

4.2.4.3 blockDim

この変数は **dim3** 型（Section4.3.1.2 を参照）で、ブロックの次数を含んでいます。

4.2.4.4 threadIdx

この変数は **uint3** 型（Section4.3.1.3 を参照）で、ブロック内部のスレッド・インデックスを含んでいます。

4.2.4.5 制限

- あらゆる組み込み変数のアドレスの取得を許可しません。
- あらゆる組み込み変数への値の割り当てを許可しません。
-

4.2.5 NVCC を伴うコンパイル

nvcc はコンパイルしているCUDAコードの処理を単純にするコンパイラ・ドライバです：簡単で身近なコマンド・ライン・オプションを提供し、異なるコンパイル段階を実装するツールの収集を呼び出すことで実行します。

nvcc の基本的流れはホスト・コードからデバイス・コードを分離するのと、バイナリ・フォームまたは *cubin* オブジェクトへコンパイルしているデバイス・コードをから成り立ちます。生成したホスト・コードは、別のツールを使用してコンパイルした残りか、最終コンパイル段階にてホスト・コンパイラが直接呼び出したオブジェクト・コードとしての、Cコードとしての出力です。

アプリケーションは生成したホスト・コードを無視するか、CUDAドライバ API（Section4.5.3 を参照下さい）を使った、デバイス上の *cubin* オブジェクトのロードか実行のどちらもできます。または、それらはグローバルに初期化したデータ行列としての *cubin* オブジェクトを含んんでいたり、必要なCUDAランタイム・スタートアップ・コードからロード及び起動した各コンパイル・カーネル（Section4.5.2 を参照下さい）へ、Section4.2.3 に述べている実行コンフィグレーション構文の変換を含んで生成したホスト・コードにリンクすることができます。

C++の構文ルールによるとコンパイラのフロント・エンドは CUDA ソースファイルを処理します。フル C++はホスト・コード用にサポートされます。また一方、C++ののサブセット C だけはデバイス・コードを全てサポートします；C++の基本ブロック内部変数のクラス、継承または宣言子などの特定の機能は違います。C++構文ルールを使用した帰結として、無効ポインター（例；**malloc()**による返し）は型に嵌っていない非無効ポインターへの割り当てができません。

nvcc の流れの詳細な記述とコマンド・オプションはこれ以外のドキュメントで読めます。

nvcc は以下の Section で説明している2つのコンパイラ指示文で紹介しています。

4.2.5.1 __noinline__

デフォルトで_**device**_関数はいつもインラインされています。

__noinline__ 関数修飾子はできれば関数についてどんなインラインのためでなく、コンパイラのためならヒントとして使えます。関数本体はそれが呼ばれたのと同じファイル内に、まだあるに違いありません。

コンパイラは__noinline__ 修飾子をポインター・パラメータ付き関数用と大きなパラメータ・リスト付き関数用に引き受けません。

4.2.5.2 #pragma unroll

デフォルトでコンパイラは既知のトリップ・カウント付きの小さなループを展開します。すべての与えられたループを展開しながら制御するのに#pragma unroll 命令を使用できます。それはループの直前に置かれなくてはならなく、ループに対してだけ適用されます。その数字はオプションでループを何回展開しなければならないかをという指示することになります。

例として、このコード・サンプル内で：

```
#pragma unroll 5
for (int i = 0; i < n; ++i)
```

このループは 5 回展開します。それはプログラマー次第で、その展開はプログラムの正当性に影響を与えないでしょう（上記の例でもし n が 5 より小さい場合はそうかも知れません）。

#pragma unroll 1 はコンパイラがループを展開するのを防止するでしょう。

もしトリップ・カウンターが定数で数値が全く#pragma unroll の後に指定されないなら、ループは展開でき、さもなければ全く展開できません。

4.3 共通ランタイム・コンポーネント

共通ランタイム・コンポーネントはホストとデバイス関数の両方により使用できます。

4.3.1 組み込みベクター型

4.3.1.1 char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, float1, float2, float3, float4

これらはから基本整数と浮動小数点型から生成されたベクター型です。それらは構造体で、1 番目から 4 番目のコンポーネントはフィールド x,y,z と w それぞれを経由してアクセスできます。これらは全てフォーム make_<type name>の構造体関数とともに来ます。例として、

```
int2 make_int2(int x, int y);
```

値**(x, y)**を伴う **int2** 型のベクターを生成します。

4.3.1.2 dim3 型

この型は次数を指定するのに使用される uint3 に基づく整数ベクター型です。dim3 型の変数を定義する際に、不特定の状態で残っている、どんなコンポーネントも1に初期化します。

4.3.2 数学的関数

デバイスで実行されると、Table B-1 には各エラー領域と共に現在サポートしている、数学的関数 C/C++標準ライブラリの総覧を収めています。

ホスト・コードで実行されると、与えられた関数は可能ならば C ランタイム実装を使います。

4.3.3 時間関数

```
clock_t clock();
```

これは各クロック周期で増加されるカウンターの値を返します。

カーネルの最初と最後のカウンターを抽出し、2つのサンプルの違いを取得し、完全にスレッドを実行したデバイスにより取得した、クロック・サイクルの数の各スレッドあたりの測定結果を記録します。しかし、デバイスで実行したスレッド命令を費やしたクロック・サイクルの数ではありません。前の数はタイム・スライスした最後のスレッドよりも大きいです。

4.3.4 テクスチャ型

CUDAはテクスチャ・メモリにアクセスするグラフィックス用GPUのテクスチャリング・ハードウェアのサブセットをサポートします。

グローバル・メモリの代わりにテクスチャ・メモリから読み込んだデータは Section5.4 に記述した幾つかの性能利得を得ることができます。

テクスチャ・メモリは Section4.4.5 に記述した*テクスチャ・フェッチ*と呼ばれるデバイス関数を使ってカーネルから読み込みます。最初のテクスチャ・フェッチのパラメータは*テクスチャ・レファレンス*と呼ばれるオブジェクトを指定します。

テクスチャ・レファレンスはテクスチャメモリの部分をフェッチするか定義します。それはホスト・ランタイム関数（Section4.5.2.6 お呼び 4.5.3.9）を経て*テクスチャ*と呼んでいるメモリのある領域に結合されなければなりません。これ以前に、それはカーネルにより使用できます。幾つかの特殊テクスチャ・レファレンスは同じテクスチャかメモリ内の重複したテクスチへ結合されるかも知れません。テクスチャ・レファレンスは幾つかの属性を保有しています。それらの一つは、テクスチャが*テクスチャ座標*を使用する1次元行列か 2 次元行列として記述するかどうかを、2つのテクスチャを使用することで、指定する次元数が調整されます。行列の要素は「テクスチャ・エレメント」を短縮して*テクセル*と呼ばれます。他の属性は、どのように入力座標が割り込まれて処理されたかというのと同様に、テクスチャ・フェッチの入出力データ型を定義します。

4.3.4.1 テクスチャ・レファレンスの宣言

幾つかのテクスチャ・レファレンスの属性は不変で、コンパイルする時に分かっていなければならなく、それらはテクスチャ・レファレンスを宣言した時に指定されます。テクスチャ・レファレンスは**テクスチャ**型の変数としてファイル・スコープで宣言されます：

```
テクスチャ<Type, Dim, ReadMode> texRef;
```

ここで：

- **Type** はテクスチャをフェッチしたときに返されるデータ型です；**Type** は Section4.3.1.1 に記述している、基本的整数、浮動小数点型と 1-, 2-及び 4-コンンポーネント・ベクター型に制限します；
- **Dim** はテクスチャ参照の 1 か2の次数を指定します；**Dim** はデフォルトを1とするオプション引数です；
- **ReadMode** は **cudaReadModeNormalizedFloat** か **cudaReadModeElementType** と同等です；もし、それが **cudaReadModeNormalizerFloat** で、**Type** が16ビットか8ビット整数型で、その値は最大範囲の符号なし整数型用の[0.0, 1.0]と符号付整数型用の[-1.0, 1.0]にマップされたときに、実際に浮動小数点として値を返します； 例えば値 0xff を伴う符号なし8ビットテクスチャ要素を1と読む、もし、それが **cudaReadModeElementType** なら変換は実行しません。ReadMode は cudaReadElementType をデフォルトとするオプション引数です。

4.3.4.2 ランタイム・テクスチャ参照属性

テクスチャ参照の他の属性は易変で、ホストからホスト・ランタイムのときに変えることができます(ランタイム API については Section4.5.2.6、ドライバーAPI いついては 4.5.3.9)。それらはテクスチャ座標が正規化かどうか、アドレッシング・モード及びテクスチャ・フィルタリングを以下のように詳細に指定します。

デフォルトでテクスチャは[0, N]の範囲内の浮動小数点座標を使って参照されます。このとき、N は座標に対応する寸法のテクスチャのサイズです。 例えば、サイズの 64x32 にあるテクスチャは x と y 次数の座標が[0, 63]と[0, 31]に参照されます。 正規化テクスチャ座標で、[0, N]の代わりに[0.0, 1.0]の範囲に座標を指定します。そして、同じ 64x32 のテクスチャは x, y 両座標の[0, 1]範囲内に正規化してアドレスされます。 正規化テクスチャ座標は、もしそれが、テクスチャ・サイズ如何に関係なくテクスチャ座標が望ましいなら、幾つかのアプリケーションの要求に自然に適合します。

アドレッシング・モードはテクスチャ座標が範囲外のときにどうなるかを定義します。非正規化テクスチャ座標を使用するとき、[0, N]範囲外のテクスチャ座標はクランプされています。 値が 0 未満は 0 に設定され、N 以上は N-1 に設定されます。クランピングはまた、正規化テクスチャ座標を使用しているときはデフォルト・アドレッシング・モードです：0.0 未満か 1.0 を超える値は[0.0, 1.0]の範囲にクランプされます。正規化座標として、「ラップ」アドレッシング・モードも指定されるかも知れません。テクスチャが周期的信号を含むときに、通常はラップ・アドレッシングが使用されます。それはテクスチャ座標の断片的部分だけを使用します。 例えば、1.25 は 0.25 に、-1.25 は 0.75 と同じように扱われるということです。 リニア・テクスチャ・フィルタリングは浮動小数点・データを返すための構成されるテクスチャ用にだけ実行されるかも知れません。 それは隣接のテクセル間の低い精度の補間をします。 可能になると、テクスチャ・フェッチ位置を囲むテクセルが読まれ、テクスチャ・フェッチのリターン値はテクスチャ座標がテクセルの間で落下したところに基づいた状態で補間されます。簡単なリニア補間は一次元テクスチャのために実行されます、そして、バイ・リニアの補間は二次元テクスチャのために実行されます。

Appendix F はテクスチャ・フェッチングのより詳細なことを記述しています。

4.3.4.3 リニア・メモリ対 CUDA 行列によるテクスチャ

テクスチャはリニア・メモリか CUDA 行列のどんな領域にあるかもしれません（Section4.5.1.2 を参照下さい）。

テクスチャはリニア・メモリに割り当てられます：

- 次数＝1の時のみ実行できます：
- テクスチャ・フィルタリングはサポートしません；
- 非正規化整数テクスチャ座標をしたときのみアドレスできます；
- 前のアドレッシング・モードはサポートしません； 範囲外のときにテクスチャ・アクセスは0を返します。
- ハードウェアは整列要求にテクスチャ・ベース・アドレスで実行します。プログラマから行列要求を抽出するために、この関数はデバイス・メモリ上へテクスチャ参照を拘束します。デバイス・メモリにテクスチャ参照を拘束する関数は必要なメモリから読むために、テクスチャ・フェッチに適用するパス・バックされた 1 バイトを戻します。CUDA の配分ルーチンで返されたベース・ポインタはこの整列規制に一致しています。そしてアプリケーションは、割り当てられたポインタを cudaBind テクスチャ()/cuTexRefSetAddress() に通過することによって、全体でオフセットを避けることができます。

4.4 デバイス・ランタイム・コンポーネント

デバイス・ランタイム・コンポーネントはデバイス関数でのみ使用できます。

4.4.1 数学関数

Table B-1 の幾つかの関数は、それほど正確ではありませんが、より速いバージョンはデバイス・ランタイム・コンポーネントに存在しています; それで、_ (_sin(x)のような)と共に同じ名前を前に置いています。それらの組み込み関数は、それらの各エラー結合と共に Table B-2 に列記しています。コンパイラに存在しているなら、あらゆる関数にそれほど正確でないカウンターパートにコンパイルさせるオプション(-use_fast_math)があります。

4.4.2 関数の同期

```
void _syncthreads();
```

ブロック内のすべてのスレッドを同期します。すべてのスレッドが一旦このポイントに達すると、通常は実行を再開します。

_syncthreads()は、同じブロックのスレッドのコミュニケーションを調整するのに使用されます。ブロックの中のいくつかのスレッドが共有されたかグローバルなメモリの同じアドレスにアクセスするとき、それらは潜在的なリード-アフター-ライト、ライト-アフター-リードまたはライト-アフター-ライトのメモリ・アクセスの危険性があります。これらがアクセスする中間のスレッドを連動させることによって、これらのデータ危険を避けることができます。

_syncthreads() は条件が全体のスレッド・ブロックにわたり完全に同じと評価した時に、条件付コードを許可します。さもなければコード実行は、故意でない副作用に掛かるか、ハングを発生しそうです。

4.4.3 型変換関数

以下での関数における接尾語は IEEE-754 丸めモードを示します:

- **rn** は round-to-nearest-even のことです
- **rz** は round-towards-zero のことです
- **ru** は round-up のことです (正の無限大へ),
- **rd** は round-down のことです(負の無限大へ).

```
int __float2int_[rn,rz,ru,rd](float);
```

指定した丸めモードを使用して、浮動小数点の引数を整数に変換します。

```
unsigned int __float2uint_[rn,rz,ru,rd](float);
```

指定した丸めモードを使用して、浮動小数点の引数を符号なし整数に変換します。

```
float __int2float_[rn,rz,ru,rd](int);
```

指定した丸めモードを使用して、整数の引数を浮動小数点に変換します。

```
float __uint2float_[rn,rz,ru,rd](unsigned int);
```

指定した丸めモードを使用して、符号なし整数の引数を浮動小数点に変換します。

4.4.4 型キャスト関数

```
float __int_as_float(int);
```

値を変更せず、整数の引数に浮動小数点の型キャストを実行します。例えば、
__int_as_float(0xC0000000)は-2 と等しいです。

performs a floating-point type cast on the integer argument, leaving the value unchanged. For example, __int_as_float(0xC0000000) is equal to -2.

```
int __float_as_int(float);
```

値を変更せず、浮動小数点の引数に整数の型キャストを実行します。例えば、
__float_as_int(1.0f)は to 0x3f800000 と等しいです。

4.4.5 テクスチャ関数

4.4.5.1 デバイス・メモリからのテクスチャリング

デバイス・メモリからのテクスチャリングのときに、テクスチャは tex1Dfetch()関数群と共にアクセスされます; 例として:

```
template<class Type>
Type tex1Dfetch(
   texture<Type, 1, cudaReadModeElementType> texRef,
   int x);

float tex1Dfetch(
   texture<unsigned char, 1, cudaReadModeNormalizedFloat> texRef,
   int x);

float tex1Dfetch(
   texture<signed char, 1, cudaReadModeNormalizedFloat> texRef,
```

```
  int x);

float tex1Dfetch(
  texture<unsigned short, 1, cudaReadModeNormalizedFloat> texRef,
  int x);

float tex1Dfetch(
  texture<signed short, 1, cudaReadModeNormalizedFloat> texRef,
  int x);
```

これらの関数は、テクスチャ座標 x を使用することでテクスチャ参照 **texRef** に拘束されたリニア・メモリの範囲をとって来ます。テクスチャ・フィルタリングとアドレッシング・モードは全くサポートされません。整数型のために、これらの関数は整数を 32 ビットの浮動小数点に任意にプロモートするかもしれません。そのうえ、上に示された関数、2 と 4 倍はサポートされます; 例えば:

```
float4 tex1Dfetch(
  texture<uchar4, 1, cudaReadModeNormalizedFloat> texRef,
  int x);
```

テクスチャ座標 x を使用することでテクスチャ参照 **texRef** に拘束したリニア・メモリをとって来ます。

4.4.5.2 CUDA 行列からのテクスチャリング

CUDA 行列からテクスチャリングするとき、テクスチャは **tex1D()** か **tex2D()**

と共にアクセスされます:

```
template<class Type, enum cudatextureReadMode readMode>
Type tex1D(texture<Type, 1, readMode> texRef, float x);

template<class Type, enum cudatextureReadMode readMode>
Type tex2D(texture<Type, 2, readMode> texRef, float x, float y);
```

これらの関数は、テクスチャ座標 x と y を使用することでテクスチャ参照 **texRef** に縛られた CUDA 行列をとって来ます。テクスチャ参照の不変(コンパイル時)の、そして、可変(ランタイム)の属性の組み合わせは、座標がどんな処理がテクスチャ・フェッチの間で起こるか、返し値はテクスチャ・フェッチにより配信したか読み取るかを決定します (Section 4.3.4.1 と 4.3.4.2 を参照下さい)。

4.4.6 原子関数

原子関数は演算能力 1.1 のデバイス用でのみ可能です。それらは Appendix C にリストされています。原子関数はグローバル・メモリに存在する 1 つの 32 ビット・ワードの読込み−編集−書込み原子演算子を実行します。例えば、**atomicAdd()**はグローバル・メモリの同じアドレスにある 32 ビット・ワードを読込み、整数をアドします。そして、同じアドレスに結果を書込みます。その演算子はセンス内の原子で、他のスレッドから干渉なく実行することを保証されます。言い換えれば、演算子が完全になるまで、他のどんなスレッドもこのアドレスにアクセスすることができません。原子演算子は 32 ビット符号付か符号なし整数を伴うときのみ稼動します。

4.5 ホスト・ランタイム・コンポーネント

ホスト・ランタイム・コンポーネントはホスト関数によってのみ使用できます。それは操作するために下記の関数を提供します：

- デバイス管理
- コンテキスト管理
- メモリ管理
- コード・モジュール管理
- 実行制御
- テクスチャ参照管理
- OpenGL と Direct3D の可搬性

それは2つのAPIで構成されます：

- CUDA driver API と呼ばれている低レベルの API
- *CUDA runtime API* と呼ばれている CUDA ドライバ API の上の実装されている高レベルの API

それらの API は相互に排他的です：アプリケーションは1つか他方を使用しなければなりません。CUDA ランタイムは、暗黙の初期化、コンテキスト管理及びモジュール管理を提供することによって、デバイスコード管理を容易にします。**Nvcc** により生成されたCホスト・コードは CUDA ランタイムに依存します（Section4.2.5 を参照下さい）。そして、アプリケーションは CUDA ランタイム API を使用しなければならない、このコードにリンクする。コントラストに於いて、CUDA ドライバ API はさらにコードを要求します。このことはプログラミングやデバッギングには難しいが、*cubin* オブジェクトに対処するだけであるので、より良い管理水準を提供して、言語に依存していません（Section4.2.5 を参照下さい）。CUDA ドライバーAPI を使用するカーネルを構成して、始動するのは特に難しいです。明白なファンクション・コールが Section4.2.3 で説明された実行構成構文の代わりにある状態で、実行構成とカーネル・パラメタを指定しなければなりませんから。また、デバイス・エミュレーション（Section4.5.2.7 を参照下さい）は CUDA ドライバ API では動作しません。CUDA ドライバ API は **cuda** ダイナミック・ライブラリを経由して配信され、それらの全てのエントリー・ポイントは **cu** の接頭語です。CUDA ランタイム API は **cudart** ダイナミック・ライブラリを経由して配信され、それらの全てのエントリー・ポイントは **cuda** の接頭語です。

4.5.1 共通概念

4.5.1.1 デバイス

両方の API は、カーネル実行のためにシステムの上で利用可能なデバイスを列挙するために関数を提供して、彼らの特性について問い合わせし、それらの 1 つを選択します（ランタイム API のために Section4.5.2.2 とドライバ API のために Section4.5.3.2 を参照下さい）。

幾つかのホスト・スレッドは同一デバイスでデバイス・コードを実行できます。しかし設計により、1 つのホスト・スレッドはデバイス・コードを1つのデバイスでしか実行できません。結果として、複数のホストスレッドが、複数のデバイスでデバイス・コードを実行するのに必要となります。さらに、別のホスト・スレッドからのランタイムはランタイムを経由して 1 つのホスト・スレッドで作成されたどんな CUDA リソースも使用することはできません。

4.5.1.2 メモリ

デバイス・メモリは*リニア・メモリ*または *CUDA 行列* のいずれかとして割り当てできます。リニア・メモリは32ビット・アドレス空間のデバイスに存在します。例えば、別々に割り当てられたエンティティーは二分木のポインタ経由でお互いに参照できます。

CUDA 行列はテクスチャ・フェッチするために最適化された不透明なメモリ・レイアウトです（Section4.3.4 を参照下さい）。それらは1次元か2次元で、エレメントの集合です。それぞれは、1, 2 か 4 のコンポーネントを保有し、おそらく符号付か符号なし 8-, 16-または 32 ビット整数、16 ビット（現在ドライバ API を経由してのみサポートしています）か 32 ビット浮動です。

CUDA 行列は単にカーネルでテクスチャのフェッチすることで読込み可能であり、同じ数の詰まっているコンポーネントでテクスチャ参照に拘束されるだけかもしれません。リニア・メモリと CUDA 行列の両方は、Section4.5.2.3 及び 4.5.3.6 に記述していているメモリ・コピー関数経由でホストにより読込み可能で書込み可能です。また、ホスト・ランタイムは **malloc()**によって割り当てられた通常のページ-可能なホストメモリと、対照的に割り当てる関数とフリー・ページ -固定ホスト・メモリ- を提供します（ランタイム API のための SectionD.5.6 及び D.5.7 とドライバ API のための E8.5 及び E8.6 を参照下さい）。 ページ固定メモリの1つの優位点は、もしホスト・スレッドによるデータ交換を実行するためにのみ、ホスト・メモリに割り当てられたものがページ固定に割り当てられたら、ホスト・メモリとデバイス・メモリ間のバンド幅が高いことです。 ページ固定メモリは希少リソースです。それでページ固定されたメモリにおける配分はページ-可能なメモリにおける配分のずっと前に失敗し始めるでしょう。ページング用にオペレーティングシステムに利用可能な物理的なメモリの量を減少させることで、あまりに多くのページ固定メモリを割り当てると、総合システム性能は抑えられます。

4.5.1.3 OpenGL の相互運用性

OpenGL バッファ・オブジェクトは CUDA のアドレッシング内にマップされるでしょう。CUDA が、OpenGL によって記述されたデータを読出すか、または CUDA が OpenGL で費やされるためにデータを書込むのを可能にするどちらかで。Section4.5 .2 .7 でどのようにランタイム API で実行されるかを記述し、Section4.5.3.10 ではドライバ API について説明します。

4.5.1.4 Direct3D の相互運用性

Direct3D 9.0 頂点バッファは CUDA のアドレス空間内部へマップされるでしょう。Direct3D により記述された DUDA がデータを読出すか、Direct3D により費やされるために CUDA がデータを書き込むのを可能のするどちらかで。Section4.5 .2 .8 でどのようにランタイム API で実行されるかを記述し、Section4.5.2.8 ではドライバ API について説明します。

CUDA コンテクストは一度の1つだけの Direct3D デバイスを伴い相互運用するでしょう。最初／最後の関数を呼び出すことは Section4.5.2.8 及び 4.5.3.11 に記述しています。

CUDA コンテクストtpDirect3D デバイスは同じ GPU で生成されなければなりません。これはランタイム API 用の **cudaD3D9GetDevice()**（SectionD.9.7 を参照下さい）、またはドライバ API 用の **cuD3D9GetDevice()**（SectionE.11.7 を参照下さい）を使用して、Direct3D によって使用されるアダプターに対応する CUDA デバイスについて問い合わせすることで確実にすることができます。

Direct3D デバイスは **D3DCREATE_HARDWARE_VERTEXPROCESSING** フラッグ付で生成されなければなりません。

CUDA 以下を未だサポートしていません。

- Direct3D 9.0 以外のバージョン
- 頂点バッファ以外の Direct3D オブジェクト

また、Direct3D ドライバと CUDA コンテクストが異なったドライバで、Direct3D と CUDA の負荷バランスが相互運用性より好まれる場合に作成されるのを保証するために **cudaD3D9GetDevice()** か **cuD3D9GetDevice** を使用できます。

4.5.1.5 コンカレント実行の非同期

ホストとデバイス間のコンカレントの実行を容易にするための幾つかのランタイム関数は非同期です: デバイスが要求されたタスクを完了する前に制御をアプリケーションに返します。それらは:

- カーネルは_global_関数または cuGridLaunch()及び cuGridLaunchAsync()を経由して起動します;
- メモリコピーを実行して **Async** で接尾される関数;
- デバイスとデバイスの双方向でのメモリコピーを実行する関数;
- メモリをセットする関数;

また、幾つかのデバイスはページ固定したホスト・メモリとデバイス・メモリ間でカーネル実行を伴うコンカレントにコピーを実行できます。アプリケーションは **CU_DEVICE_ATTRIBUTE_GPU_OVERLAP** 付きの **cuDeviceGetAttribute()**を呼び出すことで、この機能を問い合わせするでしょう(それぞれ SectionE.2.6 を参照下さい)。この機能は現在では **cudaMallocPitch()** (Section4.5.2.3 を参照下さい)か **cuMemAllocPitch()**(Section4.5.3.6 を参照下さい)を経由して割り当てられた CUDA 行列か 2D 行列にかかわらないメモリコピーのためだけにサポートされます。

アプリケーションは*ストリーム*を経由してコンカレントに管理します。ストリームは、その命令で実行する関数の順序です。他方で異なったストリームは個別の順序外の関数をもう他方かコンカレントに実行するでしょう。

ストリームはストリーミング・オブジェクトを生成することで定義され、ストリーム・パラメータとしてカーネル起動の順序とホストとデバイスの双方向のコピーを指定します。Section4.5.2.4 ではこれをランタイム API と共に、Section4.5.3.7 ではドライバ API と共にどのように実行したかを記述しています。

すべての先行関数の後でのみ、ゼロ・ストリーム・パラメタで指定した、あらゆるカーネルの起動、メモリのセット、またはメモリーのコピーが始まります。ストリームの一部の関数を含み、後続でない関数はそれが完了するまで始まります。

ランタイム API 用の **cudaStreamQuery()**及びドライバ API 用の **cuStreamQuery()**(SectionD.3.2 及び E.5.2 のそれぞれを参照下さい)はストリーム内の全ての先行関数が完結しているなら、それを知るための方法をアプリケーションに提供します。

ランタイム API 用の **cudaStreamSynchronize()**とドライバ API 用の **cuStreamSynchronize()** (SectionE.5.2 及び E.5.3 のそれぞれを参照下さい)はストリーム内の全ての先行関数が完結まで待つ、ランタイムを明示的に強制するための方法を提供します。

ランタイム API 用の **cudaThreadSynchronize()**とドライバ API 用の **cuCtxSynchronize()** (SectionD.2.1 及び E.3.5 のそれぞれを参照下さい)アプリケーションはストリーム内の全ての先行タスクが完結するまで待つ、ランタイムを強制できます。不要なスローダウンを避けるために、タイミング目的や起動の隔離やメモリ・コピーが失敗しているときに、これらの関数を使用するのは最も良いです。

ランタイムもまたデバイスの進捗を密接にモニタして、アプリケーションでそれらのイベントが記録されたときに、プログラムとクエリーのあらゆるポイントの *イベント*を非同期に記録することを送出することで、正確なタイミングを実行する方法を提供します。イベントはイベントが完結する前なら、全てのタスク(または全ての関数はストリームを与えたもの)を記録します。Section4.5.2.5 はランタイム API で、そして Section4.5.3.8 はドライバ API で、これをどのように実行するかを記述しています。

異なるストリームからの2つの関数は、もし、ページ固定したホスト・メモリの割り当てか、デバイス・メモリ割り当てか、デバイス・メモリ・セットか、デバイス/デバイス間の双方向メモリ・コピーのいずれかはコンカレントに動作できません。また、イベントはそれらの間の記録を発生します。

プログラマは **CUDA_LAUNCH_BLOCKING** 環境変数を1に設定することにより、システムで動作する全ての CUDA アプリケーションのための非同期実行を、グローバルに無効にすることができます。この機能をデバッグ目的だけに供給すべきであり、決してプロダクション・ソフトウェアを確実に動作させる方法として使用するべきではありません。

4.5.2 ランタイム API

4.5.2.1 初期化

ランタイム API 用の明白な初期化関数はありません; ランタイム関数が呼ぶ 1 回目を初期化します。それはランタイム関数が呼んだタイミングと、最初のランタイムへの呼び出しエラー・コードを解釈したときを記憶しておく必要があります。

4.5.2.2 デバイス管理

SectionD.1 の関数はシステム内のデバイス・プレゼントを管理するのに使います。

cudaGetDeviceCount() 及び **cudaGetDeviceProperties()** はデバイスを数えて、それらの特性を検索するの方法を提供します:

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
}
```

cudaSetDevice() はホスト・スレッド関連のデバイスを選択するのに使います。

```
cudaSetDevice(device);
```

あらゆる**__global__**関数や Appendix D からのどんな関数も呼ばれる前に、デバイスを選択しなければなりません。**cudaSetDevice()**への明白なコールでこれをしないなら、自動的にデバイス 0 を選択します、そして、その後の **cudaSetDevice()**へのどんな明白なコールも効果はないでしょう。

4.5.2.3 メモリ管理

SectionD.5 の関数はデバイス・メモリの割り当てや開放とメモリがホストとデバイス・メモリの間でグローバルなメモリ空間および転送データで宣言された、あらゆる変数のためにも割り当てたアクセスに使われます。リニア・メモリは **cudaMalloc()** や **cudaMallocPitch()**を使った割り当てや

cudaFree()を使って開放されます。. 以下のコード・サンプルはリニア・メモリに 256 の浮動小数点の要素の行列を割り当てます:

```
float* devPtr;
cudaMalloc((void**)&devPtr, 256 * sizeof(float));
```

配分が整列要求を満たすため適切に冗長するのを確実にするので、2D 行列の配分は Section5.1.2.1 で記述した **cudaMallocPitch()**を推奨します。

ということで、最も良い性能を確実にするのは、2D 行列と他のデバイスメモリ（**cudaMemcpy2D()**関数を使って）領域の間で行アドレスにアクセスするときかコピーを実行するときです。行列の要素にアクセスするのは、返された間隔(スキャン幅)を使用しなければなりません。以下のコードのサンプルは、浮動小数点の値の **width×height** 2D 行列を割り当て、デバイス・コードの行列要素でどのようにループ・オーバーにするかを示します：

```
// host code
float* devPtr;
int pitch;
cudaMallocPitch((void**)&devPtr, &pitch,
                width * sizeof(float), height);
myKernel<<<100, 512>>>(devPtr, pitch);

// device code
_global_ void myKernel(float* devPtr, int pitch)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

CUDA 行列は **cudaMallocArray()**を使って割り当てし、**cudaFreeArray()**を使って開放します。**cudaMallocArray()**は **cudaCreateChannelDesc()**を使って生成されたフォーマットの記述を必要とします。

次のコード・サンプルは 32 ビット浮動小数点コンポーネントの **width×height**CUDA 行列を割り当てます。

```
cudaChannelFormatDesc channelDesc =
                        cudaCreateChannelDesc<float>();
cudaArray* cuArray;
cudaMallocArray(&cuArray, &channelDesc, width, height);
```

cudaGetSymbolAddress()はグローバル・メモリ空間で宣言された変数のために割り当てられた、メモリのアドレス・ポイントを取得するのに使用されます。**cudaGetSymbolSize()**を通して割り当てられたのがメモリのサイズです。SectionD.5 には、リニア・メモリが **cudaMalloc()**と **cudaMallocPitch()**の間に割り当てた、前の関数の全てを列挙します。CUDA 行列と変数のために割り当てられたメモリは、グローバルか定数のメモリ空間で宣言します。

次のコード・サンプルは 2D 行列から、前のコード・サンプルに割り当てられた CUDA 行列にをコピーします：

```
cudaMemcpy2DToArray(cuArray, 0, 0, devPtr, pitch,
```

```
                width * sizeof(float), height,
                cudaMemcpyDeviceToDevice);
```

次のコード・サンプルはホスト・メモリ行列からデバイス・メモリへコピーをします。

```
float data[256];
int size = sizeof(data);
float* devPtr;
cudaMalloc((void**)&devPtr, size);
cudaMemcpy(devPtr, data, size, cudaMemcpyHostToDevice);
```

次のコード・サンプルはホスト・メモリ行列から定数メモリへコピーをします：

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
```

4.5.2.4 ストリーム管理

SectionD.3 での関数はストリームを生成するか破棄するのに使われ、全てのストリームの操作が完成したか否かに関係なく決定します。

次のコード・サンプルは2つのストリームを生成します：

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
```

各ストリームは、下記のホスト・デバイスからの1つのメモリ・コピーの手順として、1つのカーネルが起動し、1つのメモリがデバイスからホストへコピーするコード・サンプルで定義されます：

```
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    myKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                size, cudaMemcpyDeviceToHost, stream[i]);
cudaThreadSynchronize();
```

各ストリームは入力行列 **hostPtr** の一部をデバイス・メモリ内の行列 **inputDevPtr** へコピーします。それは **myKernel()**と呼ばれるデバイス上で **inputDevPtr** を処理し、**hostPtr** の同じ部分に結果の **outputDevPtr** をコピーして戻します。 2 つのストリームを使用する処理 **hostPtr** は、1 つのストリームのメモリ・コピーが、もう片方のストリームのカーネルの実行と重なるのを許します。**hostPtr** はどんなオーバラップも起こるようにページ固定されたホスト・メモリをポイントしなければなりません：

```
float* hostPtr;
cudaMallocHost((void**)&hostPtr, 2 * size);
```

cudaThreadSynchronize()は処理が継続する前に、すべてのストリームが確実に終了するためにコールされます。

4.5.2.5 イベント管理

SectionD.4 の関数は生成、記録及び破棄イベントと2つのイベント間の経過時間の問い合わせに使われます：

次のコード・サンプルは2つのイベントを生成します：

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

次の方法で前の Section のコードのサンプルを調節するのに、これらのイベントを使用することができます:

```
cudaEventRecord(start, 0);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDev + i * size, inputHost + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    myKernel<<<100, 512, 0, stream[i]>>>
              (outputDev + i * size, inputDev + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(outputHost + i * size, outputDev + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

4.5.2.6 テクスチャ参照管理

SectionD.6 の関数はテクスチャ参照の管理に使われます。

ハイレベルな API によって定義された**テクスチャ**型は、次のように低レベルである API によって定義されたテクスチャ Reference タイプから配信された構造です:

```
struct textureReference
{
  int                            normalized;
  enum cudaTextureFilterMode     filterMode;
  enum cudaTextureAddressMode    addressMode[2];
  struct cudaChannelFormatDesc   channelDesc;
}
```

-
- **normalized** は、テクスチャ座標が正規化されるかどうかを指定します。それがもしゼロでないなら、テクスチャの全ての要素は[0,width-1] または[0,height-1]範囲よりむしろ[0,1]範囲のテクスチャ座標にアドレスされます。ここで **width** 及び **hight** はテクスチャのサイズです;
- **filterMod** はフィルタリング・モードを指定します、すなわち、テクスチャをフェッチするとき、値がどう戻ったかは入力テクスチャ座標に基づいて計算されます; **filterMode** は **cudaFilterModePoint** または **cudaFilterModeLinear** と同等です。もしそれが **cudaFilterModePoint** なら、戻り値はテクスチャ座標が入力テクスチャ座標の最も近くにあるテクセルです。もしそれが **cudaFilterModeLinear** なら、戻り値はテクスチャ座標が入力テクスチャ座標の最も近くにある 2 つ(一次元テクスチャのため)か 4 つ(二次元テクスチャのため)のテクセルの直線補間です;浮動小数点型の戻り値のときだけに、**cudaFilterModeLinear** は有効です;
- **addressMode** はアドレッシング・モードを指定して、それは範囲の外では、テクスチャ座標がどのように扱われるか指定します; **addressMode** はそれぞれ 1 番目と 2 番目のテクスチャ座標のために、1 番目と 2 番目の指定したアドレス・モードの2つの行列のサイズです; アドレッシング・モードが **cudaAddressModeClamp** と等しい場合は、範囲外のテクスチャ座標は有効枠へクランプされます。また **cudaAddressModeWrap** 等しい

場合は、範囲の外のテクスチャ座標は有効枠に返されます; **cudaAddressModeWrap** は正規化テクスチャ座標のためにサポートされます;

- **channelDesc** はテクスチャをフェッチした時に返された、値のフォーマットを記述します; 以下の型に **channelDesc** があります;

```
struct cudaChannelFormatDesc {
  int x, y, z, w;
  enum cudaChannelFormatKind f;
};
```

ここで x, y, z 及び w は返された値の各コンポーネントのビットの数と同等です。f は;

- cudaChannelFormatKindSigned もしこれらのコンポーネントが符号付整数型のとき;
- cudaChannelFormatKindUnsigned もしそれらが符号なし整数型のとき;
- cudaChannelFormatKindFloat もしそれらが浮動小数点型のとき。

normalized, **addressMode** 及び **filterMode** はホスト・コード内で直接編集されるでしょう。これらは CUDA 行列で結合するテクスチャ参照に適用するだけです。

カーネルの前はテクスチャ・メモリからの読出しのテクスチャ参照で使えます; テクスチャ参照は **cudaBindTexture()** か **cudaBindTextureToArray()**を使ってテクスチャに結合されなければなりません。

次のコード・サンプルはテクスチャ参照から **devPtr** によりポイントされたリニア・メモリへ結合するものです;

- 低レベル API の使用;

```
texture<float, 1, cudaReadModeElementType> texRef;
textureReference* texRefPtr;
cudaGetTextureReference(&texRefPtr, “texRef”);
cudaChannelFormatDesc channelDesc =
                              cudaCreateChannelDesc<float>();
cudaBindTexture(0, texRefPtr, devPtr, &channelDesc, size);
```

- 高レベル API の使用;

```
texture<float, 1, cudaReadModeElementType> texRef;
cudaBindTexture(0, texRef, devPtr, size);
```

次のコード・サンプルは CUDA 行列 **cuArray** へテクスチャ参照を結合するものです:

- 低レベル API の使用;

```
texture<float, 2, cudaReadModeElementType> texRef;
textureReference* texRefPtr;
cudaGetTextureReference(&texRefPtr, “texRef”);
cudaChannelFormatDesc channelDesc;
cudaGetChannelDesc(&channelDesc, cuArray);
cudaBindTextureToArray(texRef, cuArray, &channelDesc);
```

- 高レベル API の使用;

```
texture<float, 2, cudaReadModeElementType> texRef;
cudaBindTextureToArray(texRef, cuArray);
```

テクスチャ参照にテクスチャを結束するのに指定されたフォーマットは、テクスチャ参照を宣言するとき指定されたパラメタと合わなければなりません;さもなければ、テクスチャ・フェッチの結果は未定義となります;

cudaUnbindTexture()はテクスチャ参照の非拘束に使われます。

4.5.2.7 OpenGL 相互運用性

SectionD.8 の関数は OpenGL の相互運用性の管理に使われます。

バッファ・オブジェクトはマップできる前に CUDA に登録されなければなりません。これは **cudaGLRegisterBufferObject** で行われます。

```
GLuint bufferObj;
cudaGLRegisterBufferObject(bufferObj);
```

それがいったん登録されていると、バッファ・オブジェクトにが **cudaGLMapBufferObject()**で返したデバイス・メモリ・アドレスを使用して、読出しか、またはカーネルで書込むことができます；

```
GLuint bufferObj;
float* devPtr;
cudaGLMapBufferObject((void**)&devPtr, bufferObj);
```

非マッピングは **cudaGLUnmapBufferObject()**で、非登録は **cudaGLUnregisterBufferObject()**.で行われます。

4.5.2.8 Direct3D 相互運用性

SectionD.9 の関数は Direct3D で相互運用性の管理に使います。

Direct3D での相互運用性は **cudaD3D9Begin()**を使って初期化し、**cudaD3D9End()**.を使って終了しなければなりません。

これらのコールの間では、マップすることができる前に、CUDA に頂点オブジェクトを登録しなければなりません。これは **cudaD3D9RegisterVertexBuffer()**によって為されます。

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
cudaD3D9RegisterVertexBuffer(vertexBuffer);
```

それがいったん登録されていると、頂点バッファは **cudaD3D9MapVertexBuffer()**が返したデバイス・メモリ・アドレスが使用して、カーネルで読出すか書込むことができます；

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
float* devPtr;
cudaD3D9MapVertexBuffer((void**)&devPtr, vertexBuffer);
```

非マッピングは **cudaD3D9UnmapVertexBuffer()**で、非登録は **cudaD3D9UnregisterVertexBuffer()**で為されます。

4.5.2.9 デバイス・エミューション・モードを使ったデバッグ

プログラミング環境はデバイスで動作するあらゆるコードのネーティブ・デバッグ・サポートを含みませんが、デバッグ目的用のデバイス・エミュレーションでできるようになります。コンパイルするとき、アプリケーションはこのモード（**-deviceemu** を使って）です。デバイス・コードは、ホストでコンパイルして動作します、プログアラマはそれがホスト・アプリケーションであるかのように、アプリケーションをデバッグするのに、ホストのネイティブ・デバッグ・サポートを使用するのを許容します。プロセッサ・マクロの**_DEVICE_EMULATION_**はこのモードで定義されます。使用されるどんなライブラリも含む、アプリケーションのためのすべてのコードは、デバイス・エミュレーション用かデバイス実行用かのいずれかがコンパイルされていなければなりません。リンク・コードは初期化の際に、以下のランタイム・エラーを返す原因となるデバイス実行用コード・コンパイルで、デバイス・エミュレーション用にコンパイルされます。

cudaErrorMixedDeviceExecution.

デバイス・エミュレーション・モードでのアプリケーションが動作している時に、プログラミング・モデルはランタイムによりエミュレートされます。スレッド・ブロックの各スレッドについては、ランタイムはホストでスレッドを生成します。

プログラマは、以下のことを確実にする必要があります：

- ホストは動作するブロックあたりのスレッドの最多の数と、マスター・スレッド用に1つを加えて使えます。
- 各スレッドが 256KB のスタックを確保することを知った上で、十分なメモリは全てのスレッドを動作させることができます。

デバイス・エミュレーション・モードで提供された多くの機能が非常に効果的なデバッグ用ツールにします：

- ホストのネイティブ・デバッグを使用することによって、サポートプログラマはデバッガがサポートするすべての機能を使用することができます、区切り点を設定して、データを点検するように。
- デバイス・コードがホストで走るためにコンパイルされるので、デバイスで動くことができないコードで、コードは増大することができます。ファイルやスクリーンへの入出力操作(printf()など)のように。
- すべてのデータがホストの上にあるので、デバイスかホスト・コードのどちらからでも、あらゆるデバイスかホスト特有のデータを読出すことができます； 同様に、あらゆるデバイスやホスト関数はデバイスかホストのどちらかからコールすることができます。
- 本質的な同期でない不正確な用法の場合には、ランタイムはデッド・ロックしている状況を検出します。

プログラマは、デバイス・エミュレーション・モードがそれをシミュレートするのではなく、デバイスをエミュレートすることであることを覚えておかなければなりません。ということで、デバイス・エミュレーション・モードは、アルゴリズムの誤りを見つける際に非常に役に立ちますが、いくつかの誤りは見つけ難いものです：

- メモリ・ロケーションがグリッド内部のマルチ・スレッドにアクセスするのと同時に、デバイス・エミュレーション・モードが動作する結果と、エミュレーション・モード・スレッドが順番に実行してから、デバイスで動作するときの結果は潜在的に異なります。
- デバイスのホスト・メモリへのポインタかホストのグローバル・メモリへのポインタを修飾するとき、デバイス実行は、おおよそ確実に同じ未定義の方法で失敗します。デバイス・エミュレーションは作成できますが、結果を修正して下さい。
- デバイス・エミュレーション・モード内のホストで実行されたものとして、デバイスで実行されたときは、ほとんど同じ浮動小数点演算は同じ結果にならないでしょう。
- これは一般的に予測されろことですが、あなたが同じ浮動小数点の計算のために異なった結果を得たのは、わずかに異なったコンパイラ・オプション、異なったコンパイラ、異なった命令セット、または異なったアーキテクチャのためです。

特に、幾つかのホスト・プラットフォームが、拡張精度レジスタで単精度浮動小数点の計算をした中間結果を保存します。デバイス・エミュレーション・モードで動作したら、潜在的に精度の重要な違いをもたらします。これが現れると、プログラマは動作することを保証はしないけれども、以下のあらゆる方法を試すことができます：

- 幾つかの浮動小数点変数は、強制単精度ストレージで揮発性であると宣言して下さい；
- **gcc** の **-ffloat-store** コンパイラ・オプションを使って下さい；
- Visual C++コンパイラの **/Op** か **/fp** オプションを使ってください；
- Linux の **_FPU_GETCW()** と **_FPU_SETCW()**、または Windows の **_controlfp()**を、下記で囲んだ、コードの部分のための強制単精度浮動小数点計算を使って下さい。

```
unsigned int originalCW;
_FPU_GETCW(originalCW);
unsigned int cw = (originalCW & ~0x300) | 0x000;
_FPU_SETCW(cw);
```

または、

```
unsigned int originalCW = _controlfp(0, 0);
_controlfp(_PC_24, _MCW_PC);
```

制御ワードの現在の値を保存する最初の時と、それを変更して、下記に使用している24 ビットに保存される仮数を強制してください。

```
_FPU_SETCW(originalCW);
```

または

```
_controlfp(originalCW, 0xfffff);
```

最後に、オリジナルの制御ワードをリストアして下さい。

演算デバイス（**エラー! 参照元が見つかりません。**を参照下さい）ではないホスト・プラットフォームはまた、通常では非正規化の数をサポートします。これは、何らかの計算が、ある場合は有限の結果で、もう片方で無限の結果を生むかも知れないことで、デバイス・エミュレーションとデバイス実行モード間での劇的に異なる結果を誘導できます。

4.5.3 ドライバ API

ドライバ API はハンドル・ベースで必須の API です; ほとんどのオブジェクトは不透明なハンドルで参照されます。おそらくオブジェクトを操るのに関数を指定されるでしょう。CUDA で可能なオブジェクトは Table 4-1 の通りです。

Table 4-1. CUDA ドライバ API で可能なオブジェクト

Object	Handle	Description
Device	CUdevice	CUDA-capable device
Context	CUcontext	Roughly equivalent to a CPU process
Module	CUmodule	Roughly equivalent to a dynamic library
Function	CUfunction	カーネル
Heap メモリ	CUdeviceptr	Pointer to device メモリ
CUDA array	CUarray	Opaque container for one-次数 al or two-次数 al data on the device, readable via texture references
Texture reference	CUtexref	Object that describes how to interpret texture メモリ data

4.5.3.1 初期化

Initialization with **cuInit()** is required before any function from Appendix E is called (see Section **エラー! 参照元が見つかりません。**).

4.5.3.2 デバイス管理

SectionE.2 の関数はシステムの現状のデバイスを管理するのに使われます。

cuDeviceGetCount()及び **cuDeviceGet()**はそれらの特性を検出するために SectionE.2 から、これらのデバイスと他の機能を数え上げる方法を提供します；

```
int deviceCount;
cuDeviceGetCount(&deviceCount);
int device;
for (int device = 0; device < deviceCount; ++device) {
    CUdevice cuDevice;
```

```
    cuDeviceGet(&cuDevice, device);
    int major, minor;
    cuDeviceComputeCapability(&major, &minor, cuDevice);
}
```

4.5.3.3 コンテクスト管理

SectionE.3 の関数は CUDA コンテクストで生成、アタッチ、デタッチするのに使います。

CUDA コンテクストは、CPU プロセスに類似しています。全てのリソースと演算 API 内で実行されるアクションは CUDA コンテクストにカプセル化されています。 そして、コンテクストが無効にされるとき、システムは自動的にこれらのリソースをきれいにします。さらに、モジュールやテクスチャ参照、それぞれのコンテクストなどのオブジェクトには、それ自身で独自の 32 ビットのアドレス空間があります。その結果 **CUdeviceptr** は参照の異なる CUDA コンテクストと異なるメモリ・ロケーションを評価します。コンテクストには、ホスト・スレッドとの 1 対1の通信があります。1つのホスト・スレッドで、同時には 1 つのデバイス・コンテキストだけを保有できるかもしれません。**cuCtxCreate()**でコンテクストを作成するとき、コールしているホスト・スレッドを現行にします。

コンテクスト(デバイス列挙かコンテクスト管理にかかわらないほとんどの関数)で作動する CUDA 関数は有効なコンテクストが現行スレッドでないなら **CUDA_ERROR_INVALID_CONTEXT** に戻るでしょう。

同じコンテクストで作動しながら第三者が書いたコードの間で相互運用性を容易にするために、ドライバーAPI は与えられたコンテクストの、それぞれの独自のクライアントによって増加したカウントを保持します。 例えば、3 つのライブラリが同じ CUDA コンテクストを使用するためにロードされるなら、ライブラリがコンテクストを使用し終わっていると、各ライブラリは使ったカウントを減少させるために、使ったカウントと **cuCtxDetach()**を増加するように **cuCtxAttach()**をコールしなければなりません。使ったカウントが 0 まで行くとき、コンテクストは無効にされます。ほとんどのライブラリに関しては、ライブラリをロードするか、または初期化する前にアプリケーションが CUDA コンテクストを作成すると予想されます;そのように、アプリケーションはそれ自身の試行錯誤法を使用することでコンテクストを生成することができます、そして、ライブラリは単にそれに手渡されたコンテクストを動かします。

4.5.3.4 モジュール管理

SectionE.4 の関数はモジュールのロード及び非ロードとハンドルの取得か変数のポインターやモジュールで定義した関数に使用します。モジュールは、デバイス・コードとデータのダイナミックにローダブルなパッケージと **nvcc**(Section4.2.5 を参照下さい)により出力した同種の Windows の DLLs です。関数、グローバル変数およびテクスチャ参照するものを含むすべてのシンボルのための名前は、第三者によって書かれたモジュールが同じ CUDA コンテクストで共同利用することができるように、モジュール・スコープで維持されます。

このコード・サンプルは、モジュールをロードして、何らかのカーネルにハンドルを取得します:

```
CUmodule cuModule;
cuModuleLoad(&cuModule, "myModule.cubin");
CUfunction cuFunction;
cuModuleGetFunction(&cuFunction, cuModule, "myKernel");
```

4.5.3.5 実行制御

SectionE.7 で記述された関数はデバイスにおけるカーネルの実行を管理します。**cuFuncSetBlockShape()**は与えられた関数用のブロックあたりのスレッド数とそれらの threadIDs がどう割り当てられるかををセットします。**cuFuncSetSharedSize()**は関数用のシェアード・メモリのサイズをセットします。**cuParam*()**関数ファミリーは **cuLaunchGrid()**か **cuLaunch()**がカーネルを始めるのにコールされる次の時に、カーネルに提供されるパラメタを指定するのに使います:

```
cuFuncSetBlockShape(cuFunction, blockWidth, blockHeight, 1);
```

```
int offset = 0;
int i;
cuParamSeti(cuFunction, offset, i);
offset += sizeof(i);
float f;
cuParamSetf(cuFunction, offset, f);
offset += sizeof(f);
char data[32];
cuParamSetv(cuFunction, offset, (void*)data, sizeof(data));
offset += sizeof(data);
cuParamSetSize(cuFunction, offset);
cuFuncSetSharedSize(cuFunction, numElements * sizeof(float));
cuLaunchGrid(cuFunction, gridWidth, gridHeight);
```

4.5.3.6 メモリ管理

SectionE.8 の関数はデバイス・メモリの割り当てと開放、そしてホストとデバイス・メモリ間のデータ転送に使われます。

リニア・メモリは **cuMemAlloc()**か **cuMemAllocPitch()**を使って割り当て、**cuMemFree()**を使って開放します：

次のコード・サンプルはリニア・メモリ内の 256 浮動小数点要素の行列に割り当てます：

```
CUdeviceptr devPtr;
cuMemAlloc(&devPtr, 256 * sizeof(float));
```

割り当てが Section5.1.2.1 で記述した整列要求を満たすために、適切に水増しするのを確実にするので、2D 行列の割り当て用に **cuMemAllocPitch()**を推奨します。

返されたピッチ(スキャン幅)は行列の要素にアクセスするのに使用しなければなりません。次のコード・サンプルは、浮動小数点の値の **width**×**height** の 2D 行列を割り当てて、デバイス・コードの行列の要素の上でどのようにループするかを示します：

```
// host code
CUdeviceptr devPtr;
int pitch;
cuMemAllocPitch(&devPtr, &pitch,
                width * sizeof(float), height, 4);
cuModuleGetFunction(&cuFunction, cuModule, "myKernel");
cuFuncSetBlockShape(cuFunction, 512, 1, 1);
cuParamSeti(cuFunction, 0, devPtr);
cuParamSetSize(cuFunction, sizeof(devPtr));
cuLaunchGrid(cuFunction, 100, 1);

// device code
_global_ void myKernel(float* devPtr)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
```

```
    }
}
```

CUDA 行列は **cuArrayCreate()**を使って生成され、**cuArrayDestroy()**を使って破棄します；

以下のコード・サンプルは1つの 32 ビット浮動小数点コンポーネントの **width×height** の CUDA 行列の割り当てを許可します：

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = width;
desc.Height = height;
CUarray cuArray;
cuArrayCreate(&cuArray, &desc);
```

SectionE.8 は **cuMemAlloc()**で割り当て、メモリとリニア・メモリ間のコピーに使った全ての前の関数を列挙しています；リニア・メモリは CUDA 行列と **cuMemAllocPitch()**で割り当てます。次のコード・サンプルは 2D 行列から前のコード・サンプルで割り当てた CUDA 行列へコピーします：

```
CUDA_MEMCPY2D copyParam;
memset(&copyParam, 0, sizeof(copyParam));
copyParam.dstMemoryType = CU_MEMORYTYPE_ARRAY;
copyParam.dstArray = cuArray;
copyParam.srcMemoryType = CU_MEMORYTYPE_DEVICE;
copyParam.srcDevice = devPtr;
copyParam.srcPitch = pitch;
copyParam.WidthInBytes = width * sizeof(float);
copyParam.Height = height;
cuMemcpy2D(&copyParam);
```

次のコード・サンプルは幾つかのホスト・メモリ行列からデバイス・メモリへコピーします：

```
float data[256];
int size = sizeof(data);
CUdeviceptr devPtr;
cuMemAlloc(&devPtr, size);
cuMemcpyHtoD(devPtr, data, size);
```

4.5.3.7 ストリーム管理

SectionE.5 の関数はストリームの生成及び破棄と、ストリームの全ての操作が完結したか否かに関係なく、決定するのに使われます。次のコード・サンプルは2つのストリームを生成します：

```
CUstream stream[2];
for (int i = 0; i < 2; ++i)
    cuStreamCreate(&stream[i], 0);
```

それらの各ストリームは、ホストからデバイスへの1つのメモリ・コピーや、1つのカーネルが起動してデバイスからホストへの1つのメモリ・コピーの手順として、次のコード・サンプルのように定義されます：

```
for (int i = 0; i < 2; ++i)
    cuMemcpyHtoDAsync(inputDevPtr + i * size, hostPtr + i * size,
                      size, stream[i]);
for (int i = 0; i < 2; ++i) {
    cuFuncSetBlockShape(cuFunction, 512, 1, 1);
```

```
    int offset = 0;
    cuParamSeti(cuFunction, offset, outputDevPtr);
    offset += sizeof(int);
    cuParamSeti(cuFunction, offset, inputDevPtr);
    offset += sizeof(int);
    cuParamSeti(cuFunction, offset, size);
    offset += sizeof(int);
    cuParamSetSize(cuFunction, offset);
    cuLaunchGridAsync(cuFunction, 100, 1, stream[i]);
}
for (int i = 0; i < 2; ++i)
    cuMemcpyDtoHAsync(hostPtr + i * size, outputDevPtr + i * size,
                      size, stream[i]);
cudaCtxSynchronize();
```

各ストリームは、その入力行列 **hostPtr** の部分をデバイス・メモリ内の行列 **inputDevPtr** へコピーして、**cuFunction** でコールしたデバイス上の **inputDevPtr** を処理します；そして結果 **outputDevPtr** をコピーして **hostPtr** の同じ部分へ戻します。2 つのストリームを使用する処理 hostPtr は、1 つのストリームのメモリ・コピーが潜在的にもう片方のストリームのカーネル実行に重なるのを許容します。**hostPtr** はどんなオーバラップも起こるようにページ固定されたホスト・メモリをポイントしなければなりません；

```
float* hostPtr;
cuMemAllocHost((void**)&hostPtr, 2 * size);
```

cuCtxSynchronize()は、全てのストリームがさらに進む前に、終わるのを確実にするため最後でコールされます。

4.5.3.8 イベント管理

SectionE.6 のこの関数はイベントの生成、記録及び破棄と2つのイベント間の経過時間を問い合わせるのに使います。次のコード・サンプルは2つのイベントを生成します：

```
CUevent start, stop;
cuEventCreate(&start);
cuEventCreate(&stop);
```

以下の方法で前の Section のコードのサンプルの時間にこれらのイベントを使用できます：

```
cuEventRecord(start, 0);
for (int i = 0; i < 2; ++i)
    cuMemcpyHtoDAsync(inputDevPtr + i * size, hostPtr + i * size,
                      size, stream[i]);
for (int i = 0; i < 2; ++i) {
    cuFuncSetBlockShape(cuFunction, 512, 1, 1);
    int offset = 0;
    cuParamSeti(cuFunction, offset, outputDevPtr);
    offset += sizeof(int);
    cuParamSeti(cuFunction, offset, inputDevPtr);
    offset += sizeof(int);
    cuParamSeti(cuFunction, offset, size);
    offset += sizeof(int);
    cuParamSetSize(cuFunction, offset);
    cuLaunchGridAsync(cuFunction, 100, 1, stream[i]);
}
for (int i = 0; i < 2; ++i)
    cuMemcpyDtoHAsync(hostPtr + i * size, outputDevPtr + i * size,
                      size, stream[i]);
cuEventRecord(stop, 0);
```

```
cuEventSynchronize(stop);
float elapsedTime;
cuEventElapsedTime(&elapsedTime, start, stop);
```

4.5.3.9 テクスチャ参照管理

SectionE.9 の関数はテクスチャ参照の管理に使います：

カーネルがテクスチャ・メモリから読出すのにテクスチャ参照を使える前に、テクスチャ参照は **cuTexRefSetAddress()**か **cuTexRefSetArray()**を使って拘束しなければなりません。

もし、モジュール **cuModule** がテクスチャ<float, 2, cudaReadModeElementType> texRef として定義された、幾つかのテクスチャ参照 **texRef** 含んでいるなら、次のコード・サンプルは **texRef** のハンドルを取得します。

```
CUtexref cuTexRef;
cuModuleGetTexRef(&cuTexRef, cuModule, "texRef");
```

次のコード・サンプルは **texRef** から **devPtr** によってポイントされた、幾つかのリニア・メモリを結合します。

```
cuTexRefSetAddress(NULL, cuTexRef, devPtr, size);
```

次のコード・サンプルは **texRef** から CUDA 行列 **cuArray** を結合します。

```
cuTexRefSetArray(cuTexRef, cuArray, CU_TRSA_OVERRIDE_FORMAT);
```

SectionE.9 には幾つかのテクスチャ参照のためにアドレス・モード、フィルター・モード、フォーマット及び他のフラッグを設定するのに使った、前の関数を列挙しています。これらのフォーマットはテクスチャからテクスチャ参照定義されます。テクスチャ参照にテクスチャを結合るとき指定されたフォーマットは、テクスチャ参照を宣言するとき指定されたパラメターと一致しなければなりません；さもなければ、テクスチャ・フェッチの結果は未定義です。

4.5.3.10 OpenGL 相互運用性

SectionE.10 からの関数は、OpenGL と共に相互運用性を制御するのに使用されます。**cuGLInit()**を使用して OpenGL の相互運用性を初期化しなければなりません。それをマッピングできる前に CUDA にバッファ・オブジェクトを登録しなければなりません。**cuGLRegisterBufferObject()**と共に行います：

```
GLuint bufferObj;
cuGLRegisterBufferObject(bufferObj);
```

それがいったん登録されていると、**cuGLMapBufferObject()**で返した、デバイス・メモリ・アドレスを使って、カーネルはバッファ・オブジェクトを読出しか書込むことができます：

```
GLuint bufferObj;
CUdeviceptr devPtr;
int size;
cuGLMapBufferObject(&devPtr, &size, bufferObj);
```

非マッピングは **cuGLUnmapBufferObject()**で行い、非登録は **cuGLUnregisterBufferObject()**で行います：

4.5.3.11 Direct3D の相互運用性

SectionD.9 の関数は Direct3D の相互運用性の制御に使われます。Direct3D の相互運用性 **cuD3D9Begin()**を使って初期化され、**cuD3D9End()**を使ってターミネートされなければなりません。

これらのコールの間では、頂点オブジェクトをマッピングできる前に、CUDA オブジェクトに登録しなければなりません。これは **cuD3D9RegisterVertexBuffer()**で行います：

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
cuD3D9RegisterVertexBuffer(vertexBuffer);
```

いったんそれが登録されると、頂点バッファは **cuD3D9MapVertexBuffer()**で返されたデバイス・メモリ・アドレスを使って、カーネルによって読出しか書込みができます：

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
CUdeviceptr devPtr;
int size;
cuD3D9MapVertexBuffer(&devPtr, &size, vertexBuffer);
```

非マッピングは **cuD3D9UnmapVertexBuffer()**で行われ、非登録は **cuD3D9UnregisterVertexBuffer()**で行われます。

Chapter 5.
性能ガイドライン

5.1 性能命令

スレッドのワープの命令処理のためにマルチプロセッサがしなければならないことは：

- ワープの各スレッド用の命令オペランドの読出し；
- 命令の実行；
- ワープの各スレッド用の結果の書込み

したがって、有効な命令スループットはメモリ待ち時間や帯域幅と同様に公称の命令スループットに依存します。それは次により最大化されます：

- 低いスループット(Section5.1.1 を参照下さい)で命令の使用を最小にします；
- メモリの各カテゴリのために利用可能なメモリ帯域幅の使用を最大にします。(Section5.1.2 を参照下さい)
- できるだけ可能な数学的計算のオーバーラップ・メモリ・トランザクションへのスレッド・スケジューラを許可するのは、次のことが必要です：
  - スレッドによって実行されるプログラムは、メモリ処理毎の算術処理の高い算術能力があること；
  - Section5.2 に詳細に記述しているように、マルチプロセッサ毎のアクティブなスレッドが多くあること。

5.1.1 命令スループット

5.1.1.1 算術命令

ワープ用の1つの命令を発行するのにマルチプロセッサが行うことは

- 浮動小数点加算、浮動小数点乗算、浮動小数点乗算-加算、整数加算、ビット単位演算、比較、最小化、最大化、型変換命令のための4クロック・サイクル；
- 逆数、逆平方根、_log(x)のための 16 クロック・サイクル（Table B-2 を参照下さい）

32 ビットの整数乗法は 16 クロック周期を取りますが、**_mul24** と**_umul24** (Appendix B を参照ください)は符合化し、非符合化 24 ビットの整数乗法を 4 クロック周期で提供します。しかしながら将来のアーキテクチャは、**_[u]mul24** は 32 ビット整数乗算より遅くなります。それで、私達はアプリケーションにより適切にコールされる**_[u]mul24** と、もう1つは一般的 32 ビット整数乗算の2つのカーネルを提供することを推奨します。

整数分割とモジュロ演算を特にコストがかかり、できれば避けるか、またはビット単位処理に取り替えるべきです。可能であるときはいつも:

もし n が 2 であるなら, (i/n) は (i>>log2(n)) と同等で、(i%n) は (i&(n-1))と同等です; n がリテラルなら、コンパイラはこれらの変換を実行するでしょう。

それらが幾つかの命令の組み合わせとして実行されるとき、他の関数はより多くのクロック周期を取ります。

浮動小数点平方根は、乗算により許可された逆平方根に代わって、逆数により許可された逆平方根として実装されます。そして、それは無限数とゼロのために正しい結果を与えます。 ということで、それはワープのために 32 ビット・クロック・サイクルを取ります。浮動小数点除算は 36 クロック・サイクルを取りますが、_fdividef(x, y)は 20 クロック・サイクルで高速除算を提供します(Appendix B を参照下さい)。

_sin(x), _cos(x), _exp(x) は 32 クロック・サイクルを取ります。

時には、コンパイラは追加実行サイクルを紹介して、変換命令を挿入しなければなりません。

- オペランドが一般的に int に変換される必要がある char か short を作動させる関数、
- 単精度の浮動小数点の計算に入力されるように使用される倍精度浮動小数点定数(あらゆる型の接尾語なしで定義されます)、
- 入力パラメタとして Table B-1 で定義された数学関数の倍精度バージョンに使用される、単精度の浮動小数点の変数。

以下を使用することによって、最後の 2 つのケースを避けることができます:

- 3.141592653589793f, 1.0f, 0.5f,のような接尾語が f で定義された単精度浮動小数点定数
- sinf(), logf(), expf().のような接尾語 f で定義された数学的関数の単精度バージョン

単精度コードのために、浮動型と数学的単精度関数の使用を強く推奨します。デバイスのために 1.x 演算能力のデバイスのように、ネーティブ倍精度サポートなしでコンパイルするとき、倍型はデフォルトとそれらの単精度方程式にマップされた倍精度数学的関数によって浮動へ格下げされます。ところで、倍精度をサポートするそれらの将来のデバイスでは、これらの関数は実装を倍精度にマップするでしょう。

5.1.1.2 フロー命令の制御

同じワープのスレッドが分岐を引き起こすことで、どんなフロー制御命令(if, switch, do, for, while)も有効命令スループットにかなり影響を与えることができます。 それは次の異なる実行パスです。もし、これが起こるなら、異なった実行パスは順序立てなければなりません、このワープによって実行された命令の総数を増加させて。すべての異なった実行パスが完結した時に、スレッドは同じ実行パスの一点に集中して戻ります。コントロール・フローがスレッドの ID で最も良い性能を得るために、制御状態は、分岐しているワープの数を最小にするために書かれているべきです。これは可能です。なぜなら、ブロックへのワープの配信は Section3.2 にあるように次ののステップが常に1つに決まるからです。些細な例は、制御状態は WSIZE がワープ・サイズのとき、(threadIdx / WSIZE)に依存するだけです。この場合、制御状態がワープに完全に整理されるので、ワープは全く分岐しません。

時には、コンパイラがループを広げるか、以下で詳しく述べられるように、代わりに分岐述語を使用することによって、出力 if か switch ステートメントを最適化するかもしれません。これらの場合、ワープは分岐することができません。プログラマはまた、**#pragma unroll** の指示を使うことでループ展開を制御できます（Section4.2.5.2 を参照下さい）。

分岐述部を使用するとき、実行が制御状態による指示のいずれもスキップされません。代わりに、それらの各々はスレッド単位状態コードか*述部*に関係しています。それは制御状態で正か否かに設定されるといえども、それらの命令の各々は実行のためのスケジュールを得ます。正述部の命令のみが実際に実行されます。否述部の命令は結果を書込まず、アドレスを評価せず、またオペランドを読出しません。

コンパイラは、一部のしきい値以下の分岐状態により制御された命令の数の時だけ、述部命令を分岐命令に入れ替えます。もし、コンパイラが、多くの分岐ワープを発生させそうであるのを決定するなら、このしきい値は 7 か、さもなければ 4 です。

5.1.1.3 メモリ命令

メモリ命令はシェアードあるいはグローバル・メモリへの読み書きをする、あらゆる命令が含まれます。1つのマルチプロセッサはワープ用の1つのメモリ命令発行で4クロック・サイクルを取ります。グローバル・メモリへアクセスしているとき、それらは、メモリ待ち時間が 400 から 600 クロック・サイクルにあります。

例として、次のサンプル・コードでの割り当て操作は：

```
_shared_ float shared[32];
_device_ float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

グローバル・メモリからの読出しを発行するために4クロック・サイクルを取り、シェアード・メモリへの書込み発行のために4クロック・サイクルを取りますが、上記の全ての 400 から 600 クロック・サイクルでグローバル・メモリから浮動を読出します。

このグローバル・メモリの待ち時間の多くは、十分に独立した算術命令があれば、スレッドのスケジューラに隠れることができます。グローバル・メモリのアクセスが完了するのを待っている間、それを発行することができます。

5.1.1.4 同期命令

_syncthreads は、もし他のあらゆるほかのスレッド用に待つスレッドが無い場合、ワープを発行するのに4ブロック・サイクルを取ります。

5.1.2 メモリ帯域幅

各メモリスペースの有効な帯域幅は以下のサブ−Section で詳しく述べられるようにメモリ・アクセス・パターンにかなり依存します。

デバイス・メモリは待ち時間が長く、オン・チップ・メモリより低い帯域幅ですので、デバイス・メモリ・アクセスは最小にされるべきです。

デバイス・メモリからシェアード・メモリに行くステージ・データには典型的なプログラミング・パターンがあります; 1 ブロックの各スレッドを持つために、言い換えれば：

- デバイス・メモリからシェアード・メモリへデータをロードして下さい、
- 各スレッドが安全に異なったスレッドによって書かれたシェアード・メモリ位置を読むことができるように、ブロックの他のすべてのスレッドに同期して下さい、
- シェアード・メモリ内でデータを処理して下さい、

- 再度同期して、必要なら結果をアップデートしたシェアード・メモリを確実にして下さい、
- デバイス・メモリへ結果の返しを書込んで下さい。

5.1.2.1 グローバル・メモリ

グローバルなメモリスペースはキャッシュされませんので、最大のメモリ帯域幅を得るために、正しいアクセス・パターンに従うのはとても重要です。とくにデバイス・メモリへの高いコストのアクセスがどうあるかは：

まず、デバイスは 32 ビット、64 ビット、または 128 ビットのワードをグローバル・メモリから、ただ一つの命令でレジスタに読み込むことができます。割り当てるために：

```
_device_ type device[32];
type data = device[tid];
```

単ロード命令にコンパイルするのに、**type** がそのようなものであるに違いないので、**sizeof(type)** は 4、8、または 16 に等しく、型 **type** の変数を sizeof(**type**)バイトに並べなければなりません。（すなわち、それらのアドレスを **sizeof(type)**の倍数で持ってください）。

整列要求は Section4.3.1.1 のビルト−イン型のために、**float2** や **float4** のように自動的に条件は満足します。

構造のために、サイズと整列要求は整列指定子**_align_(8)** や **_align_(16)**を使ったコンパイラで強制できます。次のように、

```
struct _align_(8) {
    float a;
    float b;
};
```

又は

```
struct _align_(16) {
    float a;
    float b;
    float c;
};
```

16 バイトを超える構造のために、コンパイラは幾つかのロード命令を生成します。 確実にするために、それは**_align_(16)**で定義された構造にように、命令の最小数を生成します。次のように、

```
struct _align_(16) {
    float a;
    float b;
    float c;
    float d;
    float e;
};
```

5 つの 32 ビットのロード命令の代わりに 2 つの 128 ビットのロード命令にコンパイルされます。

次にグローバル・メモリは、単一の読み書き命令が単一の隣同士が合体できるようにメモリ・アクセスが実行している間は、半ワープの各スレッドのアクセスが一斉にアドレスします。

より正確に、それぞれの半ワープ内では、半ワープ内部のスレッド数 **N** はアドレスにアクセスしなければなりません。

```
HalfWarpBaseAddress + N
```

ここで、**HalfWarpBaseAddress** は **type***と **type** 型で、上記で述べた整

要求とサイズに合致するものです。おまけに、**HalfWarpBaseAddress** は **16*sizeof(type)**バイトに合わせなければなりません（例：**16*sizeof(type)**の倍数に）。

SectionD.5 か E.8 からメモリ・アロケーション・ルーチンの1つによって返されたかグローバル・メモリ内に存在する変数のあらゆるアドレス **BaseAddress** はいつも、最小でも 256 バイトに整列されます。そしてメモリ整列の規則を満たすように、**HalfWarpBaseAddress-BaseAddress** は **16*sizeof(type)**の倍数でなくてはなりません。

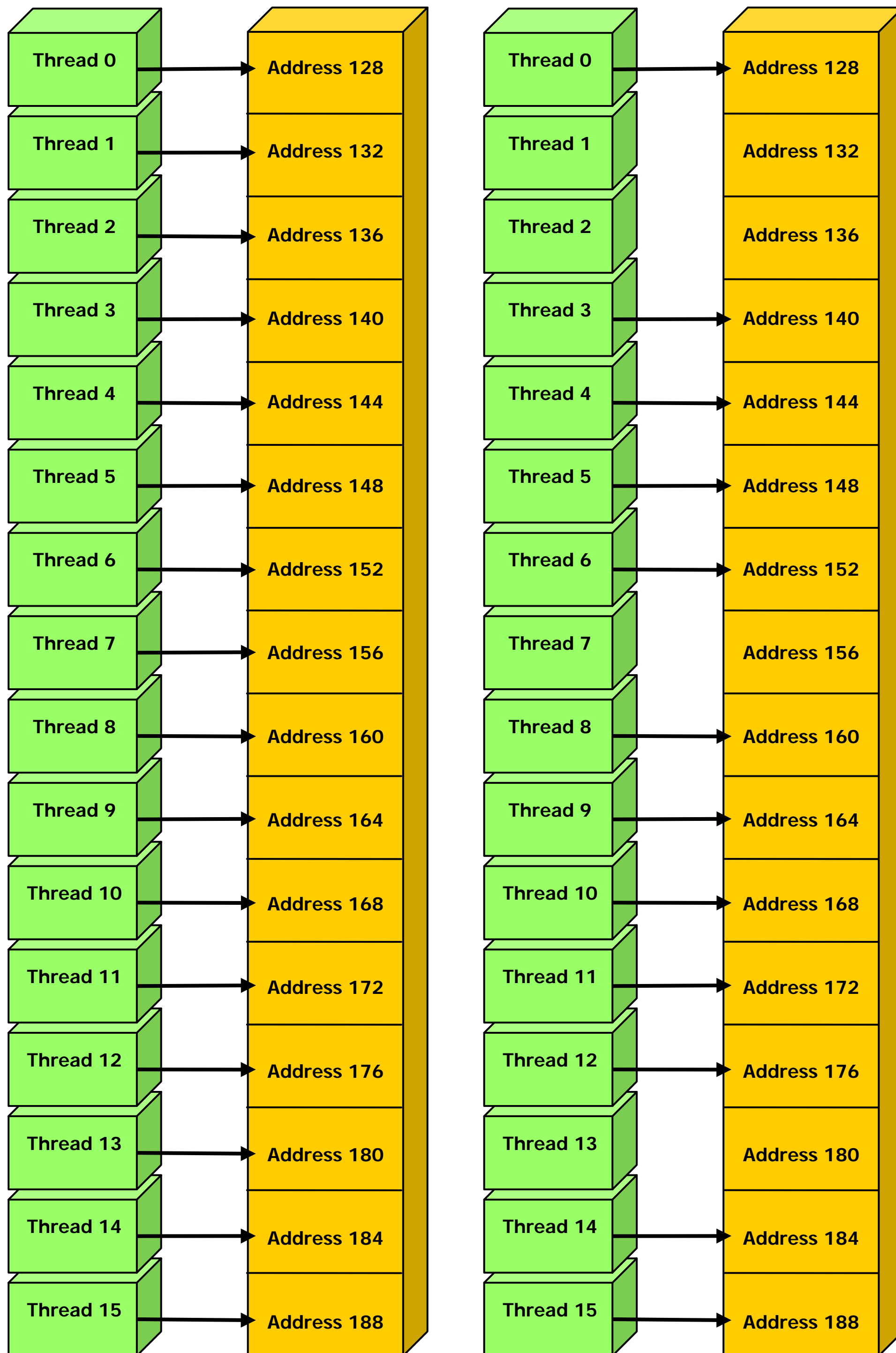
半ワープが上記の全ての要件を実現させ、半ワープの幾つかのスレッドが実際にメモリにアクセスしなくても、1 スレッドあたりのメモリ・アクセスが結合することに注意してください。

私達は将来のデバイスが適切な結合のためにそれを必要とするので、各々別々に半分に分割することに反対で、これらの全てのワープのために結合要求を満たすことを推奨します。

Figure 5-2 と Figure 5-3 は結合していないメモリ・アクセスに関する幾つかの例を示しますが、Figure 5-1 は結合しているメモリ・アクセスに関する幾つかの例を示します。

結合している 64 ビットのアクセスは、結合している 32 ビットのアクセスと、結合している 128 ビットのアクセスが、結合している 32 ビットのアクセスより顕著に低い帯域幅を配信するより、少し低い帯域幅を配信します。

しかし、これらのアクセスが 32 ビットであるときに、結合していないアクセスのための帯域幅は、結合しているアクセスよりおよそ 1 桁低いのです。それは 128 ビットの時より 2 倍、64 ビットの時よりたった 4 倍遅いのです。



左: 結合した **float** メモリ・アクセス

右: 結合した **float** メモリ・アクセス (分岐したワープ)

Figure 5-1.　結合したグローバル・メモリ・アクセス・パターンの例

左:非順列 float メモリ・アクセス

右: 調整不良スターティング・アドレス

Figure 5-2. 非結合グローバル・メモリ・パターンの例

左: 非連続 **float** メモリ・アクセス
右: 非結合 **float3** メモリ・アクセス

Figure 5-3. 非結合グローバル・メモリ・アクセス・パターンの例

共通グローバル・メモリ・アクセス・パターンは、スレッド ID **tid** の各スレッドが型 **type***のアドレス **BaseAddress** に以下のアドレスを使用することで位置する、行列の 1 つの要素にアクセスする時です：

```
BaseAddress + tid
```

メモリを結合させるために、**type** は上で述べたサイズと整列要求を満たさなければなりません。特に、この **type** が 16 バイトより大きい構造であることを意味し、それは、これらの要求を満たすいくつかの構造に分けられるべきで、そしてデータは型 **type***.の1つの行列の代わりに、これらの構造の幾つかの行列のリストとしてメモリで配置されるべきです。他の共通グローバル・メモリ・アクセス・パターンは型 **type***と、次のアドレスを使っている幅 **width** にあるアドレス **BaseAddress** インデックス(**tx,ty**)の各スレッドが、2D 行列の1つの要素にアクセスする時です：

```
BaseAddress + width * ty + tx
```

このような場合には、次の1つの時だけ、スレッド・ブロックの全ての半ワープメモリが結合します：

- スレッド・ブロックの幅は半ワープ・サイズの倍数です；
- **Width** は 16 の倍数です。

特に、この意味は、もしそれが 16 の倍数に最も近く端数が切り上げられた幅に実際に割り当てられたなら、幅が 16 の倍数でない行列は、はるかに効率的にアクセスされるということです。**cudaMallocPitch()** 及び **cuMemAllocPitch()**関数と SectionD.5 及び E.8 で述べた関連メモリ・コピー関数はプログラマーが、これらの規則に準拠して割り当てた行列に、非ハードウェア依存コードを書込むことを可能にします。

5.1.2.2 定数メモリ

定数メモリ空間がキャッシュされるので、定数メモリからの読出しはキャッシュ・ミス上でのみデバイス・メモリから、1つのメモリ読出しの負担をかけます。さもなければ、それは定数キャッシュから、だた1つの読出しの負担をかけます。

半ワープの全スレッドのために、定数キャッシュの読出しはレジスタからの読出しと同等の速度で、同じアドレスを読出す全スレッドと同等の長さです。異なったアドレスの数が全てのスレッドによって読出されている状態のコスト・スケールは比例します。私たちは、将来のデバイスがフル・スピードの読込みためにそれを要求するので、その半分だけの各々が含まれている全スレッドに対抗した同じアドレスを読込む、全体のワープの全てのスレッドを持っていることを推奨します。

5.1.2.3 テクスチャ・メモリ

テクスチャ・メモリ空間はキャッシュされるので、テクスチャ・フェッチはキャッシュ・ミス上でのみのメモリ・デバイスから、1つのメモリ読出しの負担をかけます。さもなければ、それはテクスチャ・キャッシュから、ただ1つの読出しの負担をかけます。テクスチャ・キャッシュは2D 空間場所のために最適化されますので、近くに一緒にあるテクスチャ・アドレスを読出す同じワープのスレッドが最も良い性能を達成するでしょう。 また、それはストリーミングのフェッチのために一定の待ち時間で設計されます。例えば、キャッシュ・ヒットはフェッチの待ち時間ではなく、DRAM 帯域幅デマンドを削減します。

テクスチャ・フェッチング経由のデバイス・メモリの読出しは、Section5.4 に詳しく述べたようにグローバルか定数メモリのデバイス・メモリを読出す有利な代替手段かも知れません。

5.1.2.4 シェアード・メモリ

シェアード・メモリ空間はローカル及びグローバルメモリ空間よりオン・チップであるから高速です。事実上、ワープのすべてのスレッドのためにシェアード・メモリのアクセスは次に詳細に述べるように、レジスタと同等の速度で、スレッドの間のバンク競合が無いのと同等の長さです。

高いメモリ帯域幅を達成するために、シェアード・メモリは同時にアクセスすることができるバンクと呼ばれる記憶モジュールと同じサイズに分割されます。

それで、異なる *n* メモリ・バンクで同調したあらゆるメモリは、*n* アドレスで作られた読み書き要求を同時に提供できます。柔軟な有効帯域幅は単一モジュールより *n* 倍高速の帯域幅をもたらします。

ところで、もし2つのメモリ・アドレスが同じメモリ・バンクに同調する要求をすると、それらはバンク競合し、アクセスはシリアル化されなければなりません。ハードウェアは必要ならば、競合しない要求に多く分けるように、バンク競合のメモリ要求を分割します。分割したメモリ要求の数と同等の要素により有効が減少します。もし、分割メモリ要求の数が *n* の場合、初期メモリ要求は *n* 種類のバンク競合を引き起こすといわれます。

最高の性能を得るために、バンク競合を最小にするために、どのようにメモリ・アドレス・マップをメモリ・バンクで理解しているかは、メモリ要求の計画をするように重要です。シェアード・メモリ空間の場合は、バンクが組織化されているので、連続した 32 ビットのワードは連続したバンクに割り当てられ、各バンクには、2 クロック周期あたり 32 ビットの帯域幅があります。

演算能力 1.x のデバイスのためのワープ・サイズは 32 でバンクの数は 16 です（Section5.1 を参照下さい）。シェアード・メモリがワープを求めるのはワープ前半の 1 つの要求と、ワープ後半の 1 つの要求に分けられます。

結果として、ワープの前半に属するスレッドと同じワープの後半に属するスレッドとのバンク競合があるはずがありません。

よくある例は、各スレッドがスレッド ID **tid** と幾つかのストライド **s** でインデックスをつけられた行列から 32 ビットのワードにアクセスすることです：

```
__shared__ float shared[32];
float data = shared[BaseIndex + s * tid];
```

この場合、スレッド **tid** と **tid+n** は **s*n** が同等かバンク **m** の数の倍数であるときや **n** が **m/d** の倍数（ここでで **d** は **m** と **s** の最大公約数です） であるときはいつも、同じバンクにアクセスします。帰結として、もしそれらがワープ・サイズの半分が **m/d** 以下の場合のみ、それらのバンク競合がなくなるでしょう。演算能力 1.x のデバイスのために、**d** が1か、他のワードにあるか、**m** は2の累乗なので **s** は奇数のときだけ、変換にバンク競合がありません。**エラー! 参照元が見つかりません。**と Figure 5-5 は**エラー! 参照元が見つかりません。**に表したバンク競合によるメモリ・アクセスの幾つかの例の間での、メモリ競合アクセスのない幾つかの例を表します。言及する価値がある他のケースは、各スレッドは 32 ビットの超るか、未満のサイズの要素にアクセスことです。例として、もし **char** が次の方法でアクセスした行列である場合、バンク競合をするでしょう：

```
__shared__ char shared[32];
char data = shared[BaseIndex + tid];
```

例えば、**shared[0]**, **shared[1]**, **shared[2]** 及び **shared[3]**は同じバンクに所属しています。もし、同じ行列が次の方法でアクセスしたときは、これらはあらゆるバンク競合がありません:

```
char data = shared[BaseIndex + 4 * tid];
```

構造割り当ては、これらが構造のメンバーとして要求した幾つかのメモリ・アクセスにコンパイルされます。次のコードのように:

```
_shared_ struct type shared[32];
struct type data = shared[BaseIndex + tid];
```

結果として:

- 3つの分割したメモリは、もし **type** が次のように定義されたなら、バンク競合なしで読み出します。

```
struct type {
        float x, y, z;
};
```

各メンバーは 3 つの 32 ビットのワープのストライドによってアクセスされます;

- 2つの分割したメモリは、もしが次のように定義されたならバンク競合して読み出されます。

```
struct type {
        float x, y;
};
```

各メンバーは2つの 32 ビットのワープのストライドによってアクセスされます;

- 2つの分割したメモリは、もしが次のように定義されたならバンク競合して読み出されます。

```
struct type {
        float f;
        char  c;
};
```

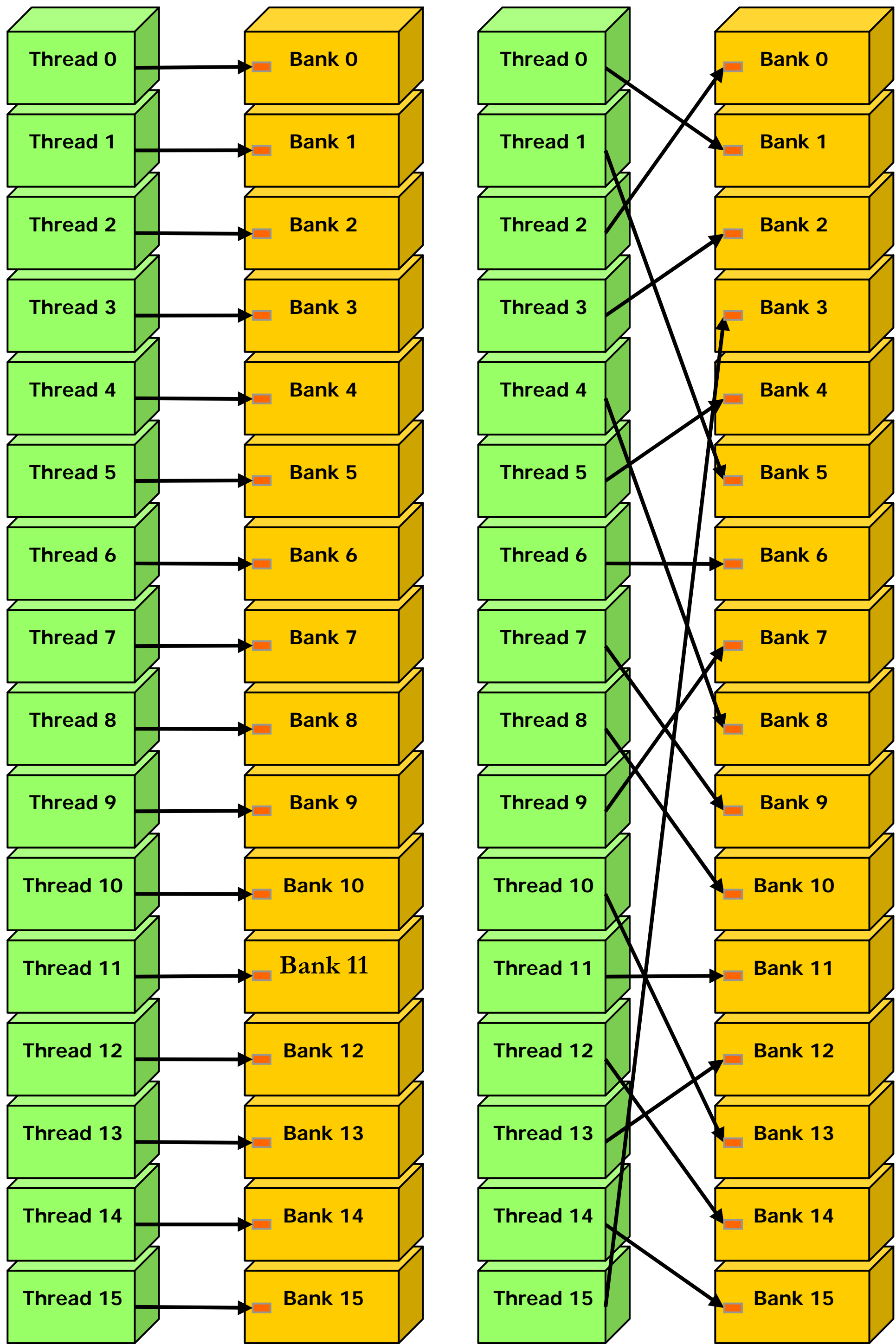
各メンバーは 5 バイトのストライドによってアクセスされます。

最後に、シェアード・メモリはまた、1つのメモリ読み出しをしているときに、32 ビット・ワードが幾つかのスレッドを同時に読み出せてブロードキャストできることによって、ブロードキャスト機構を特徴付けます

半ワープの幾つかのスレッドが同じ 32 ビットワードを含むアドレスから読み出したとき、このバンク競合の数は減少します。より正確に、幾つかのアドレスのメモリは読出し要求は、全てのアドレスが提供されるまで、ステップ毎のこれらのアドレスの1つの競合内サブセットを提供している− 2 クロック周期毎の1ステップの−時間をかけて、幾つかのステップ内で提供されます;各ステップでサブセットは以下の手順を用いてまだ提供されていない、残ったアドレスから組み立てられます:

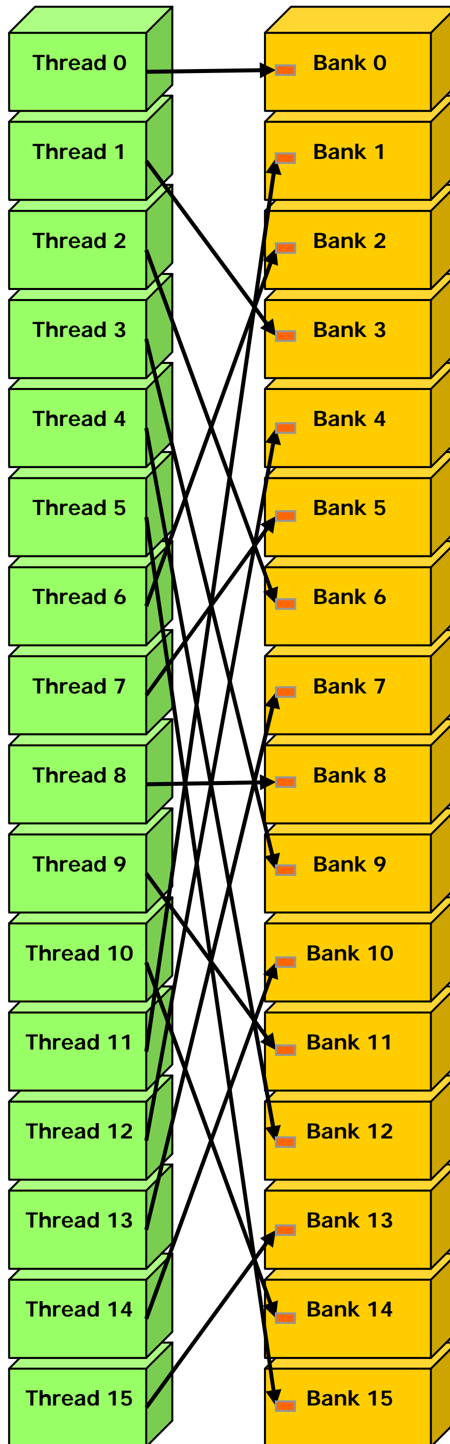
- ブロードキャスト・ワードとして残っているアドレスによって示されたワードの 1 つを選択して下さい;
- サブセットでは次を含めて下さい;
- 全てのアドレスはブロードキャスト・ワードを含んでいます、
- 各バンクの1つのアドレスは残されたアドレスにより指示します。.

どのワードがブロードキャスト・ワードとして選定されるか、そして、どのアドレスがそれぞれのサイクルに各バンクのために取り出されるかは、特定されません。共通競合なしケースは同じ 32 ビットを含むアドレスからの半ワープの全スレッドを読み出した時です。Figure 5-7 にブロードキャスト機構を取り込んだメモリ読み出しアクセスと同じ例を表します。



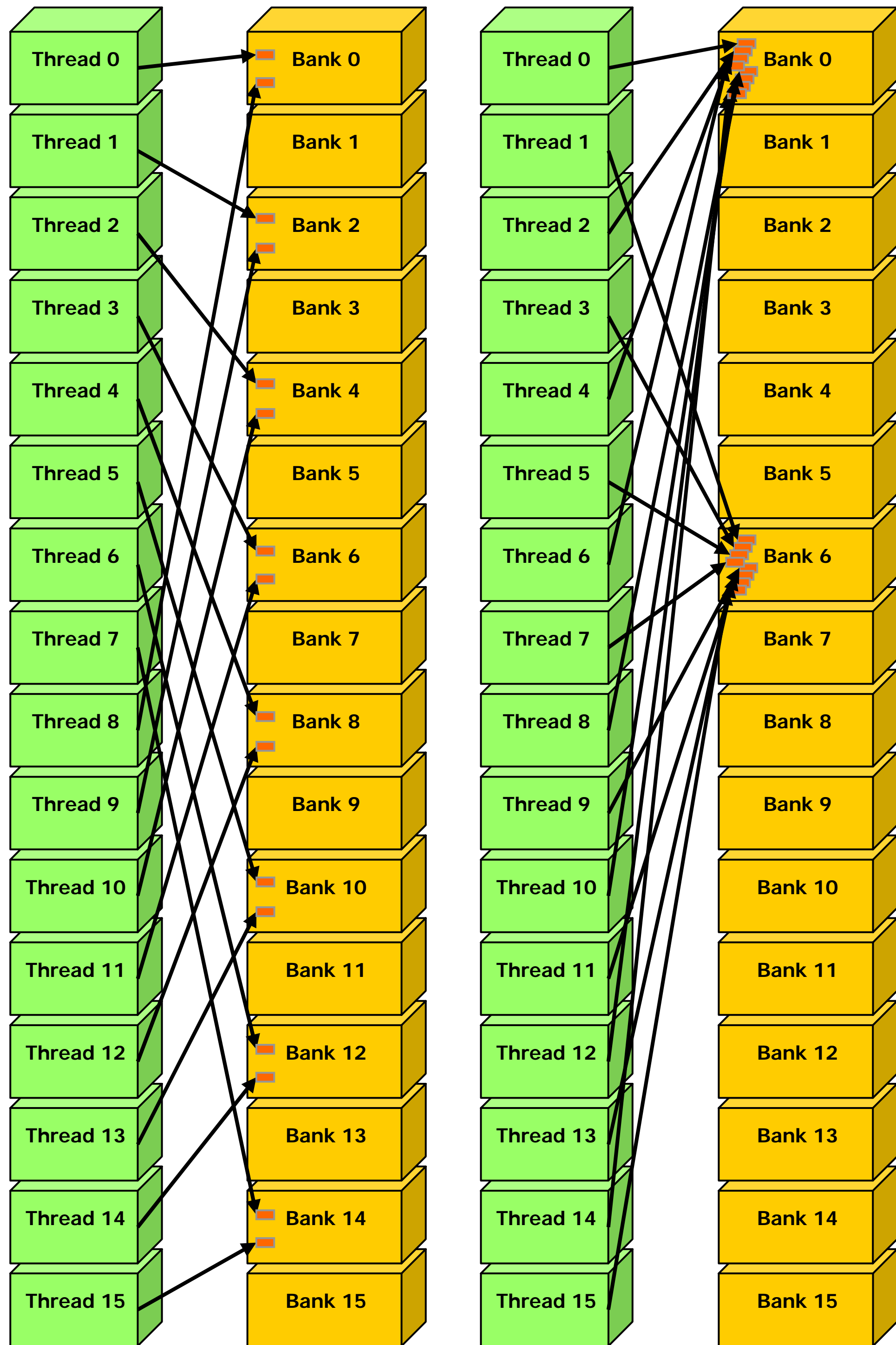
左: 1つの 32 ビットのストライドのリニア・アドレッシング
右: ランダム順列

Figure 5-4. バンク競合のシェアード・メモリ・アクセスパターンの例



3つの 32 ビットワードのストライドのリニア・アドレッシング

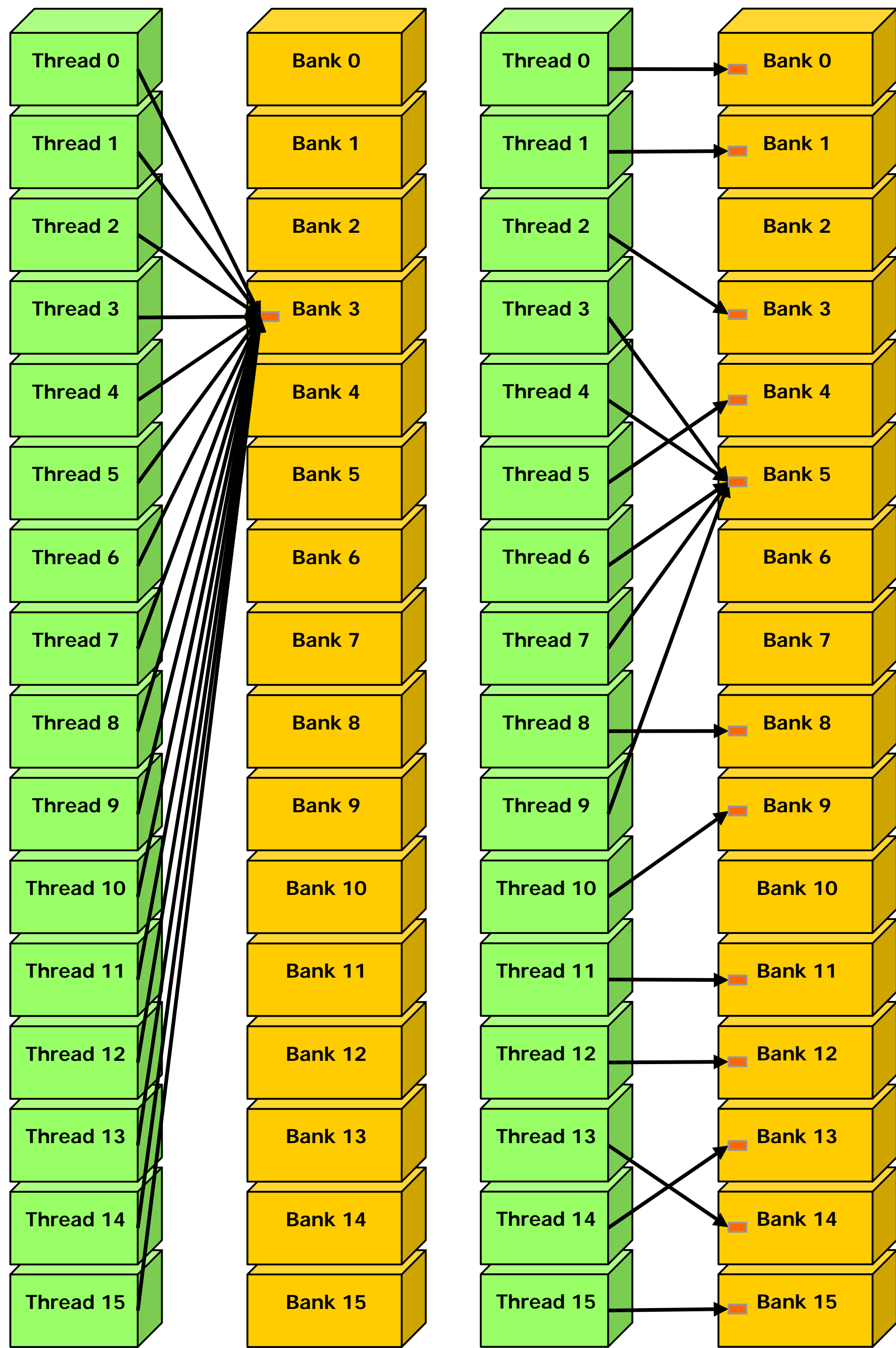
Figure 5-5. バンク競合のないシェアード・メモリ・アクセスパターン例



左: 2ウェイバンク競合による 2 つの 32 ビットのストライドのリニア・アドレッシング

右: 8ウェイバンク競合による 8 つの 32 ビット・ワードのストライドのリニア・アドレッシング

Figure 5-6. バンク競合のシェアード・メモリ・アクセス・パターンの例



左:同じ 32 ビットワードのアドレスからの全スレッドの読出しによる競合なしのこのアクセス・パターンです。

右: もしバンク 5 からワードが最初のステップか2ウェイバンク競合の間でブロードキャスト・ワードとして選ばれたなら、バンク競合なしによるアクセス・パターンです。

Figure 5-7. ブロードキャストのシェアード・メモリ読出しアクセス・パターンの例

5.1.2.5 レジスタ

一般的に、レジスタのアクセスは命令毎のゼロ拡張クロック周期です。しかし、遅延はレジスタが書込み後の読出し依存とメモリ・バンク競合によりおそらく発生します。

それらを隠すために少なくとも 1 マルチプロセッサあたりで 192 アクティブ・スレッドがあるとすぐに、遅延は書込み後の読出し依存により誘導されます。

コンパイラとスレッド・スケジューラはレジスタ・メモリ・バンク競合を避けることが最適な命令をスケジュールします。それらはブロックあたりのスレッドの数が 64 の倍数である時にベストの結果を達成します。次のルール以外にありません。アプリケーションはこれらのバンク競合を超えた非直接制御を持ちます。特に、これらはデータが **float4** か **int4** 型に返すことは不要です。

5.2 ブロックあたりのスレッドの数

グリッドあたりのスレッドの総数、ブロックあたりのスレッドの数またはブロックと同等の数を与え、利用可能な演算資源を最大化するために選ばれるべきです。このことは、少なくともマルチプロセッサがデバイスにあるのと同じくらい多くのブロックがあることを意味しています。その上に、1 マルチプロセッサあたり 1 ブロックだけを走らせると、ロード待ち時間を覆うために十分な 1 ブロックあたりのスレッドがなければ。マルチプロセッサはスレッド同期とデバイス・メモリ読出しの間も、待機することを強制されるでしょう。したがって、ブロックが走ることができるのと、ブロッの待ちとの間のオーバラップを許容するために、2 以上のブロックが各マルチプロセッサでアクティブであることを許容しているほうが良いです。これが起こるように、マルチプロセッサがデバイスにあるのと同じくらい、少なくとも2倍多くのブロックがなくてはなりません。しかしまた、1 ブロックあたりの割り当てられたシェアード・メモリの量は、1 マルチプロセッサあたり利用可能なシェアード・メモリの総量の半分であるべきです(Section3.2 を参照下さい)。デバイス経由のパイプライン方式のより多くのスレッド・ブロック・ストリームはオーバーヘッドをさらに償却します。

十分に大きい数のブロックで、1 ブロックあたりのスレッドの数はコンピュータ資源を浪費するのを避けるためにワープ・サイズの 64 の倍数として選ばれるべきです（理由は Section5.1.2.5 で記述しました）。ブロックあたりのより多くのスレッドの割り当ては効率的なタイム・スライスのために良いことです。しかし、ブロックあたりのより多くのスレッドに、少しのレジスタはスレッド単位で利用可能です。もしカーネルが実行構成によって許容されている、より多くのレジスタにコンパイルするなら、これはカーネルの呼び出しが成功するのを防ぐかもしれません。**--ptxas-options=-v** オプションでコンパイルするとき、カーネルがコンパイルするレジスタの数（ローカル、シェアード及び定数メモリの使用と同様）はコンパイラによって報告されます。

演算能力 1.x のデバイスのために、スレッドあたりのレジスタの可能数は：

$$\frac{R}{B \times ceil(T, 32)}$$

ここで *R* は Appendix A で与えられたマルチプロセッサあたりのレジスタの総数で、*B* はマルチプロセッサあたりのアクティブ・ブロックの数で、*T* はブロックあたりのスレッドの数で、*ceil(T, 32)*は 32 の倍数に近いとこまで切り上げた *T* です。

ブロックあたり 64 スレッドは最少で、もしそれらがマルチプロセッサあたり複数のアクティブ・ブロックであるなら筋が通っています。ブロックあたり 192 か 256 スレッドはより良く、通常はコンパイル用の十分なレジスタのために許可されています。

グリッドあたりのブロックの数は少なくとも 100 で、もしそれが将来のデバイスで使いたいなら、1000 ブロックなら数世代にわたり有効です。マルチプロセッサあたりのアクティブ・ワープの数に対し、アクティブ・ワープ(**エラー! 参照元が見つかりません。**で与えられた)の最大数の比率はマルチプロセッサ占有（*occupancy*)と呼ばれます。占有を最大にするために、コンパイラは、レジスタの使用を最小にするのを試みます、そして、プログラマは慎重に実行構成を選ぶ必要があります。CUDA ソフトウェア・デベロップメント・キットはプログラマがシェアード・メモリ上のスレッド・ブロック・サイズとレジスタ要求を選択するのを支援するためにスプレッドシートを提供します。

5.3 ホストとデバイス間のデータ転送

デバイスとデバイス・メモリ間の帯域幅はデバイス・メモリとホスト・メモリ間の帯域幅より十分に高速です。ということで、それはホストとデバイスの間のデータ転送を最小にするように努力するべきです。例として、ホストからデバイスに多くのコードが動くことで、それが低い並列演算のカーネルで動作することと同じ意味です。中間データ構造はおそらくデバイス・メモリで生成され、デバイスにより操作され、ホストでマップされずに放棄されたか、ホスト・メモリへコピーされたものです。

また、オーバーヘッドは各転送で関連付けられ、大きなものへの多くの小さな一括転送は各転送を切り替えるより、常により良い性能になります。

最後に、ホストとデバイス間のより高速の帯域幅は、Section4.5.1.2 に記述したよに、ページ・ロック・メモリを使って達成します。

5.4 テクスチャ・フェッチ対グローバルまたは定数メモリ読出し

デバイス・メモリは、グローバルか定数メモリからの読出しの上で、幾つかの利益があるテクスチャ・フェッチを経由して読出します：

- もし、それらがテクスチャ・フェッチに位置するのであれば、潜在的により高い帯域幅を示して、キャッシュされます；
- それらはグローバルまたは定数メモリのメモリ・アクセス・パターンでの制約のサブジェクトではありません（Section5.1.2.1 と 5.1.2.2 を参照下さい）；
- データへのランダム・アクセスを実行するアプリケーションの性能向上の可能性のために、アドレッシング算出の待ち時間は隠すほうが良いです；
- パックしたデータはおそらく1つの操作で変数を分割してブロードキャストします；
- 8ビットと16ビット整数入力データはおそらく、[0.0, 1.0] か [−1.0, 1.0] (Section 4.3.4.1 を参照下さい)の範囲で 32 ビット浮動小数点値にオプション変換されます。

もし、テクスチャが CUDA 行列（Section4.3.4.2 を参照下さい）なら、ハードウェアはおそらく異なるアプリケーション、特に画像処理に使える他の能力を供給します。

機能	用途	変換

フィルタリング	高速、低精度, テクセル間の補間	有効な時のみ、テクスチャ参照は浮動小数点データを返します
テクスチャ座標の正規化	解像度非依存のコーディング	
アドレッシング・モード	バウンドケースの自動ハンドリング	正規化テクスチャ座標に対してのみ使えます

ところで、同じカーネル・コールで、テクスチャ・キャッシュはグローバル・メモリ書込みを順守する一貫性を維持しません。それで、アドレスへのあらゆるテクスチャ・フェッチは同じカーネルが未定義データを呼び返す中へのグローバルの書込みを経由して書かれています。 別のいいかたをすれば、スレッドは、もしこのメモリ・ロケーションが前のカーネル・コールかメモリ・コピーにより更新しているときで、しかし、もしそれが、同じカーネル・コールからの同じまたは他のスレッドにより、前に更新されていない、その時のみ幾つかのメモリ・ロケーションのテクスチャを経由して安全に読出すことができます。ところで、カーネルが CUDA 行列に書込みができないようにリニア・メモリからフェッチしたときだけ、これは関連します。

5.5 総合的な性能の最適化戦略

性能の最適化は3つの基本的戦略の周囲を巡回します:

- 並列実行の最大化;
- メモリ帯域幅の最大化を達成するのためのメモリ使用の最適化;
- 命令スループットの最大化を達成するための命令使用の最適化。.

並行実行の最大化は、できるだけ多くのデータ並行化を顕在化する、処理の1つの方法のアルゴリズムの構造化で開始します。並行化のアルゴリズムのポイントは、幾つかのスレッドが各々他の間のデータの共有のために同期を必要としますので崩壊します。これらには2つのケースがあります:いずれのそれらのスレッドは__syncthreads()を使い、同じカーネル・コールにあるシェアード・メモリを経由してデータを共有しなければならないか同じブロックに所属しているか、またはそれらは、グローバル・メモリから、片方が書込み用に、他方は読出し用として、2つの異なるカーネルの起動を使ったグローバル・メモリを経由して、データを共有しなければならない異なるブロックに所属しています。

アルゴリズムの一回の並列化はそれは、できるだけ効率的にハードウェアへのマッピングしたように顕在化されています。Section5.2 で詳しく述べられるように、慎重に各々のカーネルの起動を実行する構成を選ぶことによってなされます。また、Section4.5.1.5 に記述しているように、アプリケーションはストリームを経由して、デバイスでコンカレントの実行を明白に顕在化することで並行処理を最大化して、ホストとデバイスの間のコンカレントの実行を最大にすることも同様にするべきです。

メモリ使用量を最適化するのは低帯域幅でデータ転送を最小にするのから始まります。これらにはデバイスとグローバルなメモリの間のデータ転送よりはるかに低い帯域幅があるので、Section5.3 で詳しく述べられるように、ホストとデバイスの間のデータ転送を最小にするということを意味します。また、それはデバイスのシェアード・メモリの使用を最大にすることによって、デバイスとグローバル・メモリの間のデータ転送を最少にすることを意味します、Section5.1.2 で言及されるように。

時々は最良の最適化は、それが必要なときにいつでも代わって、データの簡単な再算出により、最初の場所のあらゆるデータ転送を避けるのと同じかも知れません。

Section **エラー! 参照元が見つかりません。**、5.1.2.2、5.1.2.3 及び **エラー! 参照元が見つかりません。**に詳細に記述してますように、有効な帯域幅はメモリの各々の型のためのアクセス・パターンでの重要性の順序で変更することができます。メモリ使用量の最適化することの次のステップは、最適なメモリ・アクセス・パターンに基づいてメモリ・アクセスをできるだけ最適に組織化することです。この最適化はグローバル・メモリの帯域幅が低く、その待ち時間が数百のクロック周期としてのグローバル・メモリ・アクセスに特に重要です（Section5.1.1.3 を参照下さい）。他の処理でのシェアード・メモリ・アクセスは通常は高次のバンク競合がある場合だけ最適化する価値があります。命令用法を最適化することに関して、低いスループット（Section5.1.1.1 を参照下さい）を伴う数学命令の使用は最小にされるべきです。これは通常の関数（組み込み関数は Table B-2 に列挙しています）の代わりの組み込みや、倍精度の代わりの単精度のように、それは結末に影響しない時に速度の精度と取引することを含んでいます。特に Section5.1.1.2 に詳細に記述したデバイスの SIMD の性質から起因する制御フローに、注意を払わなければなりません。

Chapter 6.
行列乗算の例

6.1 概要

それぞれ 2 つの行列次数*(wA, hA)*と*(wB, wA)*の *A* と *B* の積 *C* を計算するタスクは以下の方法で数個スレッドに分けられます：

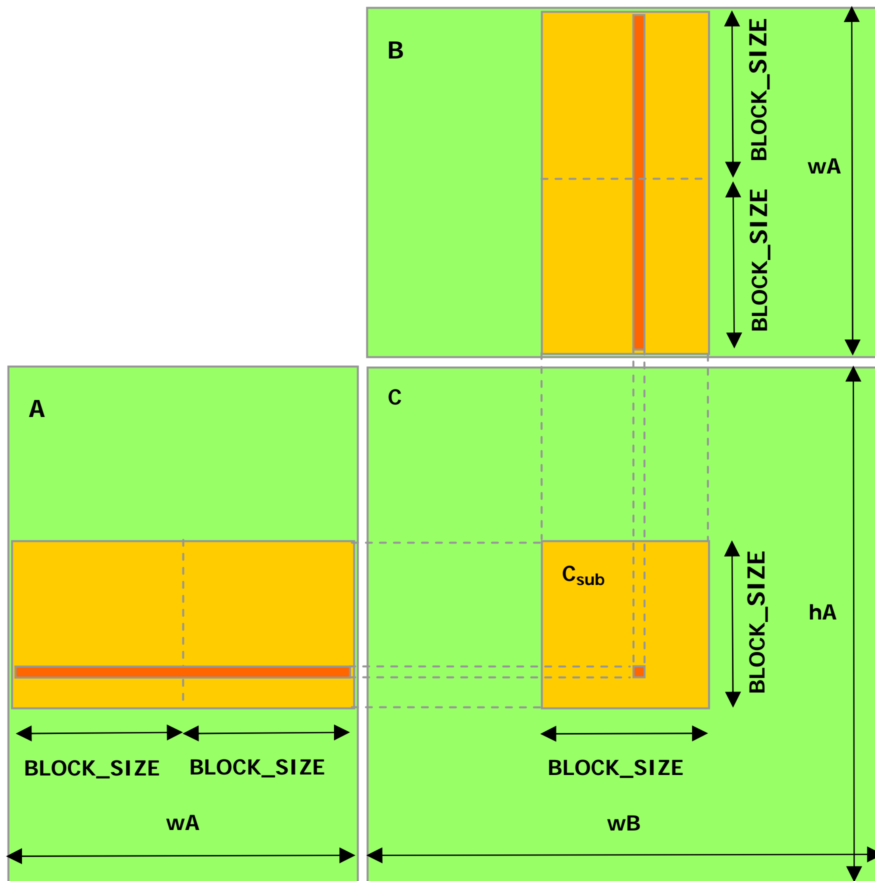
- 各スレッド・クロックは1つの平方副行列 *C* の C_{sub} を計算する責任があります
- ブロックを含む各スレッドは1つの C_{sub} 要素を計算する責任があります

C_{sub}の *block_size* 次元は 16 と同等に選ばれ、ブロック毎のスレッド数はワープ・サイズ（Section5.2）の倍数と以下のブロックあたりのスレッドの最大行列数の残りです(**エラー! 参照元が見つかりません。**)。

Figure 6-1 に示したように C_{sub} は2つの矩形行列の積と同等です：次数*(wA, block_size)*の *A* の副行列は C_{sub},としての同じ行インデックスと*(block_size, wA)* 次数の *B* の副行列を持ち、C_{sub}.としての同じ列インデックスを持っています。デバイスのリソースにフィットするように、これら2つの矩形行列は必要な *block_size* 次数の幾つかの矩形行列の分割で、C_{sub} はこれらの矩形行列の積の和として計算されます。それらの積の各々は、各行列の1つのスレッドがロードする1つの要素の最初の2つのグローバル・メモリからシェアード・メモリへ対応する矩形行列のロードで実行されます。それから、各スレッドを持っていることで積の 1 つの要素を計算します。各スレッドはそれらの各積の結果をレジスタに累算し、グローバル・メモリへの書込みを一度行います。

このように計算をブロックすることによって、私達は高速シェアード・メモリの優位性を持ち、*A* と *B* からの*(wA ／ block_size)* 回数だけのグローバル・メモリからの読出しにより、多量のグローバル・メモリの帯域幅をセーブします。

にも関わらず、この例は前の CUDA プログラミング原則に明快な説明で書かれていて、高性能カーネルを一般的なマトリクス乗算に供給するという目標でなく、そのように解釈すべきではありません。



各スレッド・ブロックは1つの C の C_{sub} を計算します。ブロックを含む各スレッドは C_{sub} の1つの要素を計算します。

Figure 6-1. 行列乗法

6.2 ソース・コードのリスト

```
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication function
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
//   hA is the height of A
//   wA is the width of A
//   wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB,
         float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
```

```
// Device multiplication function called by Mul()
// Compute C = A * B
//   wA is the width of A
//   wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();
```

```
        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

6.3 ソース・コード・ウォークスルー

ソース・コードには2つの関数が含まれています：

- **Mul()**；**Muld()**をラップして提供するホスト関数
- **Muld()**；デバイスの行列乗算を実行するカーネル

6.3.1 Mul()

Mul() は入力として取得します：

- *A* 及び *B* の要素へ指示するホスト・メモリへの2つのポインタ
- *A* の幅と高さと *B* の幅
- *C* は書込まれていなければならない指示をするホスト・メモリへのポインタ

Mul() は次の関数を実行します：

- **cudaMalloc()**を使っているストア *A, B* 及び *C* へ充分なグローバル・メモリを配分します；
- **cudaMemcpy()**を使っているグローバル・メモリへのホスト・メモリから *A* と *B* をコピーします；
- デバイスの *C* を演算するために **Muld()**をコールします；
- **cudaMemcpy()**を使っているホスト・メモリへのグローバル・メモリから *C* をコピーします；
- **cudaFree()**を使っている *A, B* 及び *C* のために割り当てられたグローバル・メモリを開放します。

6.3.2 Muld()

Muld() はホスト・メモリに代わってデバイス・メモリへのポインタが指示するのを除き **Mul()**としての同じ入力を持ちます。

各ブロックのために、C_{sub}を計算するために要求した *A* と *B* の全ての副行列を経由して **Muld()** は繰り返します。各々の繰り返しに於いて：

- *A* の1つの副行列とシェアード・メモリへのグローバル・メモリからの *B* の副行列をロードします;
- 両方の副行列はブロックを含む全てのスレッドによって完全にロードするのを確実にするために同期します;
- 積が前の繰り返しの間で求めるために2つの副行列の積と和を計算します;
- 次の繰り返しを開始する前に行った2つの副行列の積を確実にするために再度同期します。

一旦全ての副行列が扱い続けた C_{sub} は完全に計算され、グローバル・メモリへ **Muld()**が書込まれます。

Muld()は Section5.1.2.1 及び 5.1.2.4 による最高のメモリ性能へ書込まれます。

実際には、**wA** 及び **wB** は Section5.1.2.1 で推奨した 16 の倍数と仮定すれば、グローバル・メモリが結合は a, b 及び c が全て 16 と同等の **BLOCK_SIZE** の倍数なので確実になります。

これらはまた、各々の半ワープによるシェアード・メモリのバンク競合はなく、**ty** 及び **k** は全てのスレッドのために同じで、**tx** は **0** から **15** に変化しますので、各スレッドはメモリ・アクセス **As[ty][tx]**、**Bs[ty][tx]** 及び **Bs[k][tx]**のための異なるバンクとメモリ・アクセス **As[ty][k]**用の同じバンクにアクセスします。

Appendix A.
技術仕様

1.x の演算能力を持つ全てのデバイスはこの appendix の詳細技術仕様に準拠しています。

原子関数は演算能力 1.1（Section4.4.6 を参照下さい）のデバイスのためにだけ可能です。

マルチプロセッサの数と全てのデバイスがサポートしている CUDA の演算能力は次の table を参照下さい：

	マルチプロセッサの数	演算能力
GeForce 8800 Ultra, 8800 GTX	16	1.0
GeForce 8800 GT	14	1.1
GeForce 8800M GTX	12	1.1
GeForce 8800 GTS	12	1.0
GeForce 8800M GTS	8	1.1
GeForce 8600 GTS, 8600 GT, 8700M GT, 8600M GT, 8600M GS	4	1.1
GeForce 8500 GT, 8400 GS, 8400M GT, 8400M GS	2	1.1
GeForce 8400M G	1	1.1
Tesla S870	4x16	1.0
Tesla D870	2x16	1.0
Tesla C870	16	1.0
Quadro Plex 1000 Model S4	4x16	1.0
Quadro Plex 1000 Model IV	2x16	1.0
Quadro FX 5600	16	1.0
Quadro FX 4600	12	1.0
Quadro FX 1700, FX 570, NVS 320M, FX 1600M, FX 570M	4	1.1
Quadro FX 370, NVS 290, NVS 140M, NVS 135M, FX 360M	2	1.1
Quadro NVS 130M	1	1.1

クロック周波数とデバイス・メモリの総数はランタイムを使いながら呼び出されます（Section4.5.2.2 及び 4.5.3.2）。

A.1 一般仕様

- ブロック毎のスレッドの最大数は 512 です；
- スレッド・ブロックの x-, y- 及び z- 次数の最大サイズは各々512, 512 及び 64 です；
- スレッド・ブロックのグリッドの各次数の最大サイズは 65535 です；
- ワープは 32 スレッドです；
- マルチプロセッサあたりのレジスタ数は 8192 です；
- マルチプロセッサあたりのシェアード・メモリ可能の数は 16 バンク内で組織化された 16KB です（Section5.1.2.4 を参照下さい）；
- 定数メモリの総計は 64KB です；
- 定数メモリのキャッシュ・ワーキング・セットはマルチプロセッサあたり 8KB です；
- テクスチャ・メモリのキャッシュ・ワーキング・セットはマルチプロセッサあたり 8KB です；
- マルチプロセッサあたりのアクティブ・ブロックの最大数は 8 です；
- マルチプロセッサあたりのアクティブ・ワープの最大数は 24 です；
- マルチプロセッサあたりのアクティブ・スレッドの最大数は 768 です；
- 1 次元 CUDA 行列へのテクスチャ参照バウンドのための最大幅は 2^{13} です；
- 2 次元 CUDA 行列へのテクスチャ参照バウンドのための最大幅は 2^{16} で、最大高さは 2^{15} です；
- リニア・メモリへのテクスチャ参照バウンドのための最大幅は 2^{27} です；
- カーネル・サイズの制限は 2 百万ネーティブ命令です；
- 各マルチプロセッサは 8 つのプロセッサで構成されていて、マルチプロセッサは 4 クロック周期内のワープの 32 スレッドを処理可能です。

A.2 標準浮動小数点

演算デバイスは次の偏差を伴う単精度バイナリ浮動小数点計算の IEEE-754 スタンダードに準拠しています： 直接直前 +/- 無限丸めはサポートされません

- 加算器と乗算器は時々単一の乗算-加算命令（FMAD）へ併用されます；
- 除算は非標準規格方法の逆数を経由して実装されます；
- 平方根は非標準規格方法の逆平方根経由で実装されます；
- 加算器と乗算器のために最近同一丸めと直前ゼロ丸めのみは静的丸めモードを経由してサポートされます； 直接の直前 +/- 無限丸めはサポートされません；
- これらは動的可変丸めモードではありません；

- 非正規化数はサポートされません; 浮動小数点演算器と比較命令は非正規化オペランドからゼロ・プライアから浮動小数点操作へ変換します;
- アンダーフローの結果はゼロにフラッシュされます;
- これらは浮動小数点例外はオカレンスを持ち、浮動小数点例外は常にマスクされた検知メカニズムではありませんが、例外がマスク反応を出現したときは標準規格です;
- 信号伝達 NaNs はサポートされません;
- 操作の結果は関与している1つ以上の入力 NaNs は 0x7fffffff ビットパターンの静止 NaN です。IEEE-754R 標準への適合に留意下さい。もし、fminf(), fmin(), fmaxf() または fmax()への入力パラメータの1つは NaN ですが、結果は非 NaN パラメータであり、他ではありません。

浮動小数点値が整数フォーマット範囲外に落ちたケースの整数値への浮動小数点値の変換は IEEE-754 により定義されない残りです。演算デバイスのために、振る舞いはサポート範囲の終端へのクランプのためです。これは x86 アーキテクチャの振る舞いに似ていません。

Appendix B. 数学関数

SectionB.1 での関数はホストとデバイスの両方の関数で使用でき、一方で SectionB.2 での関数はデバイス関数でのみ使用できます。

B.1 共通ランタイム・コンポーネント

Table B-1 では CUDA ランタイム・ライブラリによりサポートされている全ての数学的標準ライブラリ関数を列挙します。それはまた、デバイスで実行した時の各関数のエラー・バウンドを定義します。それらのエラー・バウンドはまたホストが関数を提供しないケースで、関数がホストで実行した時に適用します。これらは広範だけど包括的ではないテストを生成し、それらはバウンドを保証しません。

加算器と乗算器は IEEE 準拠で、0.5 ulp の最大エラーを持ちます。それらは時々、乗算の中間結果を切り捨てる単一の乗算-加算命令(FMAD)を組み合わします。

浮動小数点の数である結果を整数への浮動小数点オペランドの丸めの推奨方法は rintf()であり roundf()ではありません。その理由は、8 命令平方への roundf()マップで、一方 rintf()は単一命令へマップします。truncf(), ceilf() 及び floorf()の各々はまた単一命令をマップします。

CUDA ランタイム・ライブラリもまた単一命令へマップする整数 min() 及び max()をサポートします。

Table B-1. 最大 ULP エラーの数学的標準ライブラリ関数

関数	最大 ulp エラー
x/y	2 (full range)
1/x	1 (full range)
1/sqrtf(x) rsqrtf(x)	2 (full range)
sqrtf(x)	3 (full range)
cbrtf(x)	1 (full range)
hypotf(x)	3 (full range)
expf(x)	2 (full range)
exp2f(x)	2 (full range)
exp10f(x)	2 (full range)
expm1f(x)	1 (full range)

関数	最大 ulp エラー
logf(x)	1 (full range)
log2f(x)	3 (full range)
log10f(x)	3 (full range)
log1pf(x)	2 (full range)
sinf(x)	2 (full range)
cosf(x)	2 (full range)
tanf(x)	4 (full range)
sincosf(x,sptr,cptr)	2 (full range)
asinf(x)	4 (full range)
acosf(x)	3 (full range)
atanf(x)	2 (full range)
atan2f(y,x)	3 (full range)
sinhf(x)	3 (full range)
coshf(x)	2 (full range)
tanhf(x)	2 (full range)
asinhf(x)	3 (full range)
acoshf(x)	4 (full range)
atanhf(x)	3 (full range)
powf(x,y)	16 (full range)
erff(x)	4 (full range)
erfcf(x)	8 (full range)
lgammaf(x)	6 (outside interval −10.001 ... −2.264; larger inside)
tgammaf(x)	11 (full range)
fmaf(x,y,z)	0 (full range)
frexpf(x,exp)	0 (full range)
ldexpf(x,exp)	0 (full range)
scalbnf(x,n)	0 (full range)
scalblnf(x,l)	0 (full range)
logbf(x)	0 (full range)
ilogbf(x)	0 (full range)
fmodf(x,y)	0 (full range)
remainderf(x,y)	0 (full range)
remquof(x,y,iptr)	0 (full range)
modff(x,iptr)	0 (full range)
fdimf(x,y)	0 (full range)
truncf(x)	0 (full range)
roundf(x)	0 (full range)
rintf(x)	0 (full range)

関数	最大 ulp エラー
nearbyintf(x)	0 (full range)
ceilf(x)	0 (full range)
floorf(x)	0 (full range)
lrintf(x)	0 (full range)
lroundf(x)	0 (full range)
llrintf(x)	0 (full range)
llroundf(x)	0 (full range)
signbit(x)	N/A
isinf(x)	N/A
isnan(x)	N/A
isfinite(x)	N/A
copysignf(x,y)	N/A
fminf(x,y)	N/A
fmaxf(x,y)	N/A
fabsf(x)	N/A
nanf(cptr)	N/A
nextafterf(x,y)	N/A

B.2 デバイス・ランタイム・コンポーネント

Table B-2 はデバイス・コードでのみサポートした組み込み関数を列挙します。これらのエラー・バウンドは GPU 特定のものです。これらの関数は正確ではありませんが、Table B-1 の幾つかの関数のバージョンは高速です；それらは（_sinf(x))のような接頭辞_を持ちます。

_fadd_rz(x,y)は丸め-前-ゼロの丸めモードの浮動小数点パラメータ x 及び y の和を計算します。

_fmul_rz(x,y)は丸め-前-ゼロの丸めモードの浮動小数点パラメータ x 及び y の積を計算します。

普通の浮動小数点除算と_fdividef(x,y)の両方は同じ精度を持ちますが、$2^{126} < y < 2^{128}$, _fdividef(x,y)のためにゼロの結果を供給します。一方で、普通の除算は Table B-1 の精度状況を含んで得た結果を供給します。また、もし x が無限なら、$2^{126} < y < 2^{128}$ のために_fdividef(x,y)は普通の除算が無限を返している間、（ゼロのよる無限の乗算の結果としての）NaN を供給します。

_[u]mul24(x,y)は整数パラメータ x 及び y の 24 の最下位ビットの積を計算し、結果の 32 の最下位ビットを供給します。x または y の 8 の最上位ビットは無視されます。

_[u]mulhi(x,y)は整数パラメータ x 及び y 積を計算し、64 ビット結果の 32 の最上位ビットを供給します。

_[u]mul64hi(x,y) は 64 ビット整数パラメータ x 及び y 積を計算し、128 ビット結果の 64 の最上位ビットを供給します。

_saturate(x)はもし x が 0 未満なら 0 を、もし x が 1.を超えるか x でなければ 1 を返します。

_[u]sad(x,y,z) （絶対差の和）は整数パラメータ z と整数パラメータ x 及び y の間の違いの絶対値の和を返します。

_clz(x)は連続したゼロ・ビットが整数パラメータ x の最上位ビット（例：ビット 31）で開始している 0 と 32 包括の間の数を返します。

_clzll(x) は連続したゼロ・ビットが 64 ビット整数パラメータ x の最上位ビット（例：ビット 61）で開始している 0 と 64 包括の間の数を返します。

_ffs(x)は整数パラメータ x 内の最初（最下位）のビット・セットの位置を返します。最下位ビットは位置 1 です。もし x が 0 なら_ffs()は 0 を返します。これは Linux 関数 ffs と等しいことに留意して下さい。

_ffsll(x)は 64 ビット整数パラメータ x 内の最初（最下位）のビット・セットの位置を返します。最下位ビットは位置 1 です。もし x が 0 なら_ffsll()は 0 を返します。これは Linux 関数 ffsll と等しいことに留意して下さい。

Table B-2. 演算能力 1.x のデバイスのための個別エラー・バウンド CUDA ランタイム・ライブラリによりサポートされる同一関数

Function	Error bounds
_fadd_rz(x,y)	IEEE-compliant.
_fmul_rz(x,y)	IEEE-compliant.
_fdividef(x,y)	For **y** in $[2^{-126}, 2^{126}]$, the maximum ulp error is 2.
_expf(x)	The maximum ulp error is **2 + floor(abs(1.16 * x))**.
_exp10f(x)	The maximum ulp error is **2 + floor(abs(2.95 * x))**.
_logf(x)	For **x** in [0.5, 2], the maximum absolute error is $2^{-21.41}$, otherwise, the maximum ulp error is 3.
_log2f(x)	For **x** in [0.5, 2], the maximum absolute error is 2^{-22}, otherwise, the maximum ulp error is 2.
_log10f(x)	For **x** in [0.5, 2], the maximum absolute error is 2^{-24}, otherwise, the maximum ulp error is 3.
_sinf(x)	For **x** in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.41}$, and larger otherwise.
_cosf(x)	For **x** in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.19}$, and larger otherwise.
_sincosf(x,sptr,cptr)	Same as **sinf(x)** and **cosf(x)**.
_tanf(x)	Derived from its implementation as **_sinf(x) * (1 / _cosf(x))**.
_powf(x, y)	Derived from its implementation as **exp2f(y * _log2f(x))**.
_mul24(x,y) _umul24(x,y)	N/A
_mulhi(x,y) _umulhi(x,y)	N/A
_int_as_float(x)	N/A
_float_as_int(x)	N/A
_saturate(x)	N/A
_sad(x,y,z) _usad(x,y,z)	N/A
_clz(x)	N/A
_ffs(x)	N/A

Appendix C.
原子関数

原子関数はデバイス関数内でのみ使えます。

C.1 算術関数

C.1.1 atomicAdd()

```
int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address,
                       unsigned int val);
```

32 ビットのワード **old** がグローバル・メモリのアドレス **address** で位置づけた読出し、(**old + val**)を計算し、同じアドレスのグローバル・メモリへ結果を返しストアします。これらの3つのオペランドは1つの原子取引を実行します。関数は **old** を返します。

C.1.2 atomicSub()

```
int atomicSub(int* address, int val);
unsigned int atomicSub(unsigned int* address,
                       unsigned int val);
```

32 ビットのワード **old** がグローバル・メモリのアドレス **address** で位置づけた読出し、(**old + val**)を計算し、同じアドレスのグローバル・メモリへ結果を返しストアします。これらの3つのオペランドは1つの原子取引を実行します。関数は **old** を返します。

C.1.3 atomicExch()

```
int atomicExch(int* address, int val);
unsigned int atomicExch(unsigned int* address,
                        unsigned int val);
float atomicExch(float* address, float val);
```

32 ビットのワード **old** がグローバル・メモリのアドレス **address** で位置づけた読出し、**val** を同じアドレスのグローバル・メモリへ返しストアします。これらの 2 つのオペランドは1つの原子取引を実行します。関数は **old** を返します。

C.1.4 atomicMin()

```
int atomicMin(int* address, int val);
unsigned int atomicMin(unsigned int* address,
                        unsigned int val);
```

32 ビットのワード **old** がグローバル・メモリのアドレス **address** で位置づけた読出し、**old** 及び **val** の最少を計算し、同じアドレスのグローバル・メモリへ結果を返しストアします。これらの3つのオペランドは1つの原子取引を実行します。関数は **old** を返します。

C.1.5 atomicMax()

```
int atomicMax(int* address, int val);
unsigned int atomicMax(unsigned int* address,
                        unsigned int val);
```

32 ビットのワード **old** がグローバル・メモリのアドレス **address** で位置づけた読出し、**old** 及び **val** の最大を計算し、同じアドレスのグローバル・メモリへ結果を返しストアします。これらの3つのオペランドは1つの原子取引を実行します。関数は **old** を返します。

C.1.6 atomicInc()

```
unsigned int atomicInc(unsigned int* address,
                        unsigned int val);
```

32 ビットのワード **old** がグローバル・メモリのアドレス **address** で位置づけた読出し、**((old >= val) ? 0 : (old+1))** を計算し、同じアドレスのグローバル・メモリへ結果を返しストアします。これらの3つのオペランドは1つの原子取引を実行します。関数は **old** を返します。

reads the 32-bit word **old** located at the address **address** in global メモリ, computes **((old >= val) ? 0 : (old+1))**, and stores the result back to global メモリ at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

C.1.7 atomicDec()

```
unsigned int atomicDec(unsigned int* address,
                        unsigned int val);
```

32 ビットのワード **old** がグローバル・メモリのアドレス **address** で位置づけた読出し、**(((old == 0) | (old > val)) ? val : (old-1))**を計算し、同じアドレスのグローバル・メモリへ結果を返しストアします。これらの3つのオペランドは1つの原子取引を実行します。関数は **old** を返します。

C.1.8 atomicCAS()

```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address,
                        unsigned int compare,
                        unsigned int val);
```

32 ビットのワード **old** がグローバル・メモリのアドレス **address** で位置づけた読出し、**(old == compare ? val : old)**を計算し、同じアドレスのグローバル・メモリへ結果を返しストアします。これらの3つのオペランドは1つの原子取引を実行します。関数は **old** を返します（比較と交換）。

C.2 ビット単位関数

C.2.1 atomicAnd()

```
int atomicAnd(int* address, int val);
unsigned int atomicAnd(unsigned int* address,
                       unsigned int val);
```

32 ビットのワード **old** がグローバル・メモリのアドレス **address** で位置づけた読出し、(**old & val**)を計算し、同じアドレスのグローバル・メモリへ結果を返しストアします。これらの3つのオペランドは1つの原子取引を実行します。関数は **old** を返します。

C.2.2 atomicOr()

```
int atomicOr(int* address, int val);
unsigned int atomicOr(unsigned int* address,
                      unsigned int val);
```

32 ビットのワード **old** がグローバル・メモリのアドレス **address** で位置づけた読出し、(**old | val**)を計算し、同じアドレスのグローバル・メモリへ結果を返しストアします。これらの3つのオペランドは1つの原子取引を実行します。関数は **old** を返します。

C.2.3 atomicXor()

```
int atomicXor(int* address, int val);
unsigned int atomicXor(unsigned int* address,
                       unsigned int val);
```

32 ビットのワード **old** がグローバル・メモリのアドレス **address** で位置づけた読出し、(**old ^ val**)を計算し、同じアドレスのグローバル・メモリへ結果を返しストアします。これらの3つのオペランドは1つの原子取引を実行します。関数は **old** を返します。

Appendix D.
ランタイム API 参照

ランタイム API のために2つのレベルがあります。

低-レベルの API(cuda_runtime_api.h)は nvcc のコンパイルを要求しない C-型インターフェイスです。

高-レベルの API (cuda_runtime.h)は低-レベル API の上に構築される C++-型インターフェイスです。それは引数をオーバーロードし、参照し、履行せずに幾つかの低いレベルの API ルーチンをラップします。それらの包括は C++コードから使用でき、あらゆる C++コンパイラでコンパイルできます。高-レベル API はまた、シンボル、テクスチャ及びデバイス関数を取引し、低-レベルのルーチンをラップした幾つかの CUDA 特定の包括を持ちます。これらの包括は nvcc の使用を要求します。なぜなら、それらはコンパイラ(Section4.2.5 を参照下さい)により生成されたコード次第だからです。例えば、Section4.2.3 で記述された実行構成構文から呼び出されたカーネルは nvcc でコンパイルされたソース・コードでのみ可能です。

D.1 デバイス管理

D.1.1 cudaGetDeviceCount()

```
cudaError_t cudaGetDeviceCount(int* count);
```

*count 内の返しは実行可能な 1.0 以上の演算能力のデバイスの数です。もし、そのようなデバイスがなければ、cudaGetDeviceCount()は 1 を返し、デバイス 0 はデバイス・エミュレーション・モードのみサポートし、演算能力は 1.0 未満にして下さい。

D.1.2 cudaSetDevice()

```
cudaError_t cudaSetDevice(int dev);
```

アクティブ・ホスト・スレッドがデバイス・コードを実行するデバイスとして dev を記録します。

D.1.3 cudaGetDevice()

```
cudaError_t cudaGetDevice(int* dev);
```

アクティブ・ホスト・スレッドがデバイス・コードを実行したデバイス*dev を返します。

D.1.4 cudaGetDeviceProperties()

```
cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp* prop,
                                    int dev);
```

デバイス **dev** の特性***prop** を返します。**cudaDeviceProp** の構成は次に定義されます：

```
struct cudaDeviceProp {
  char   name[256];
  size_t totalGlobalMem;
  size_t sharedMemPerBlock;
  int    regsPerBlock;
  int    warpSize;
  size_t memPitch;
  int    maxThreadsPerBlock;
  int    maxThreadsDim[3];
  int    maxGridSize[3];
  size_t totalConstMem;
  int    major;
  int    minor;
  int    clockRate;
  size_t textureAlignment;
};
```

ここで：

- **name** はデバイスで識別している ASCII 文字列；
- **totalGlobalMem** はバイトのデバイスでグローバル・メモリが可能な総数です；
- **sharedMemPerBlock** はバイトのブロックあたりでシェアード・メモリが可能な総数です；
- **regsPerBlock** はブロックあたりでレジスタが可能な総数です；
- **warpSize** はワープのサイズです；
- **memPitch** は **cudaMallocPitch()**（Section5.2）経由で割り当てられたメモリ・レジスタを取り込んだ、SectionD.5 のメモリ・コピー関数で許されている最大間隔です；
- **maxThreadsPerBlock** はブロックあたりのスレッドの最大数です；
- **maxThreadsDim[3]**はブロックの各次数の最大サイズです；
- **maxGridSize[3]**はグリッドの各次数の最大サイズです；
- **totalConstMem** はバイトのデバイスで可能な定数メモリの総数です；
- **major** 及び **minor** はデバイスの演算能力で定義しているメジャーとマイナーのレビジョン番号です。
- **clockRate** はキロヘルツのクロック周波数です；
- **textureAlignment** は Section4.3.4.3 で言及した整理要求です；テクスチャ・ベース・アドレスはテクスチャ・フェッチへオフセット要求が必要のない整理された **textureAlignment** バイトです。

D.1.5 cudaChooseDevice()

```
cudaError_t cudaChooseDevice(int* dev,
                            const struct cudaDeviceProp* prop);
```

最適な特性*prop の *dev デバイスへ返します。

D.2 スレッド管理

D.2.1 cudaThreadSynchronize()

```
cudaError_t cudaThreadSynchronize(void);
```

デバイスが持つ全ての要求を処理しているタスクを完結するまでブロックします。**cudaThreadSynchronize()**はもし処理している1つのタスクが失敗したらエラーを返します。

D.2.2 cudaThreadExit()

```
cudaError_t cudaThreadExit(void);
```

全てのホスト・スレッドをコールしながら関連付けられたランタイム関連リソースをクリーンアップします。あらゆる後続 API は最初期化するランタイムをコールします。**cudaThreadExit()**はホスト・スレッド終了の暗黙的コールです。

D.3 ストリーム管理

D.3.1 cudaStreamCreate()

```
cudaError_t cudaStreamCreate(cudaStream_t* stream);
```

ストリームを生成します。

D.3.2 cudaStreamQuery()

```
cudaError_t cudaStreamQuery(cudaStream_t stream);
```

もし全てのストリームのオペレーションが完了したか、そうでない **cudaErrorNotReady** なら **cudaSuccess** を返します。

D.3.3 cudaStreamSynchronize()

```
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
```

ストリームの全てのオペレーションが完了するまでデバイスをブロックします。

D.3.4 cudaStreamDestroy()

```
cudaError_t cudaStreamDestroy(cudaStream_t stream);
```

ストリームを破棄します。

D.4 イベント管理

D.4.1 cudaEventCreate()

```
cudaError_t cudaEventCreate(cudaEvent_t* event);
```

イベントを生成します。

D.4.2 cudaEventRecord()

```
cudaError_t cudaEventRecord(cudaEvent_t event, CUstream stream);
```

イベントを記録します。もし **stream** がゼロでない場合、イベントでストリームが完了した全ての処理オペレーションの後に記録されます；他方それは CUDA コンテクストが完了した全ての処理オペレーションの後に記録されます。このオペレーションは非同期ですので、**cudaEventQuery()**及び／または **cudaEventSynchronize()**はイベントが実際に記録された時に決定するのに使わなければなりません。

もし **cudaEventRecord()**前にコールされていて、イベントが記録されていないなら、この関数は **cudaErrorInvalidValue** を返します。

D.4.3 cudaEventQuery()

```
cudaError_t cudaEventQuery(cudaEvent_t event);
```

もし、イベントの記録が完了しているか **cudaErrorNotReady** でないなら **cudaSuccess** を返します。もし **cudaEventRecord()**がイベントでコールされていないのなら、関数は **cudaErrorInvalidValue** を返します。

D.4.4 cudaEventSynchronize()

```
cudaError_t cudaEventSynchronize(cudaEvent_t event);
```

イベントが実際に記録されるまでブロックします。もし **cudaEventRecord()**がこのイベントでコールされていないなら、関数は **cudaErrorInvalidValue** を返します。

D.4.5 cudaEventDestroy()

```
cudaError_t cudaEventDestroy(cudaEvent_t event);
```

イベントを破棄します。

D.4.6 cudaEventElapsedTime()

```
cudaError_t cudaEventElapsedTime(float* time,
                                 cudaEvent_t start,
                                 cudaEvent_t end);
```

2つのイベント（約 0.5 秒のリゾリューションのミリ秒）の間の経過時間を演算します。もし、どちらかのイベントが記録されていないなら、関数は **cudaErrorInvalidValue**.を返します。もしどちらかのイベントが非ゼロ・ストリームを記録したなら、結果は非定義です。

D.5 メモリ管理

D.5.1 cudaMalloc()

```
cudaError_t cudaMalloc(void** devPtr, size_t count);
```

デバイスにあるリニア・メモリの **count** バイトを割り当て、割り当てたメモリへのポインタ***devPtr** へ返します。割り当てられたメモリはどんな種類の変数のためにも適切に並べられます。メモリはクリアされません。**cudaMalloc()**は失敗した時は **cudaErrorMemoryAllocation** を返します。

D.5.2 cudaMallocPitch()

```
cudaError_t cudaMallocPitch(void** devPtr,
                            size_t* pitch,
                            size_t widthInBytes,
                            size_t height);
```

デバイス上に少なくとも **widthInBytes*height** バイトを割り当て、メモリに割り当てられたメモリへ ***devPtr** ポインタを返します。関数は対応する、与えられたあらゆる行は、行から行（Section5.1.2.1 を参照下さい）に更新されたアドレスとして、結合している整列要求に合致するのを続けるポインタの割り当てを確実にするために埋め込みます。ピッチは割り当てのバイトの幅 **cudaMallocPitch()** により***pitch** へ返されます。ピッチの意図された用法は、2D 行列内部の計算アドレスに使われた、割り当ての別々のパラメタとしてあります。型 T の行列の要素を行と列に与えます。アドレスは次のように計算されます

```
T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;
```

2D 行列の割り当てのために、それはプログラマが **cudaMallocPitch()**.を使っているプログラミング間隔を考慮する推奨されます。間隔整列によってハードウェアに再構築します。これはもしアプリケーションがデバイス・メモリ（リニア・メモリや CUDA 行列に関係なく）の異なる区域に2D メモリ・コピーを実行しているときは特に真です。

D.5.3 cudaFree()

```
cudaError_t cudaFree(void* devPtr);
```

cudaMalloc() または **cudaMallocPitch()**への前のコールにより返されなければならない **devPtr** により指定されたメモリ空間を開放します。さまなければ、または **cudaFree(devPtr)**が以前に既にコールされたことがあるなら、エラーは返されます。 **devPtr** が 0 であるなら、オペレーションは全く実行されません。 **cudaFree()**は失敗の場合に **cudaErrorInvalidDevicePointer** を返します。

D.5.4 cudaMallocArray()

```
cudaError_t cudaMallocArray(struct cudaArray** array,
                    const struct cudaChannelFormatDesc* desc,
                      size_t width, size_t height);
```

cudaChannelFormatDesc 構造 **desc** に応じて CUDA 行列を割り当てて、ハンドルを***array** の新しい CUDA 行列に返します。**cudaChannelFormatDesc** は Section4.3.4 で記述されています。

D.5.5 cudaFreeArray()

```
cudaError_t cudaFreeArray(struct cudaArray* array);
```

CUDA 行列 **array** を開放します。もし **array** が 0 なら実行されません。

D.5.6 cudaMallocHost()

```
cudaError_t cudaMallocHost(void** hostPtr, size_t size);
```

デバイスにアクセス可能でページ-ロックのホスト・メモリの **size** を割り当てます。ドライバーは、この関数で割り当てられた仮想記憶範囲を追跡して、**cudaMemcpy*()**.などの関数のコールを自動的に加速します。メモリがデバイスにより直接アクセスできることで、それは、**malloc()**などの機能で得られたメモリを「ページ-可能」するよりはるかに高い帯域幅でそれを読むか、または書くことができます。**cudaMallocHost()**があるメモリの過剰な量を割り当てると、ページングのシステムに利用可能なメモリの量を減少させるので、システム性能は下げられるかもしれません。結果として、ホストとデバイスの間のデータ交換のための中間準備地域を割り当てるのに、慎重にこの関数を使用するのは最良です。

D.5.7 cudaFreeHost()

```
cudaError_t cudaFreeHost(void* hostPtr);
```

前のコールによって **cudaMallocHost()**に返されたに違いない **hostPtr** によって示されたメモリ空間を解放します。

D.5.8 cudaMemset()

```
cudaError_t cudaMemset(void* devPtr, int value, size_t count);
```

定数バイト値 **value** で **devPtr** によって示されたメモリ領域の最初の **count** バイトに書き込みます。

D.5.9 cudaMemset2D()

```
cudaError_t cudaMemset2D(void* dstPtr, size_t pitch,
                    int value, size_t width, size_t height);
```

行列（**width** バイトの **hight** 行各々）が **dstPtr** で示した指定値 **value** へセットします。**pitch** は **dstPtr** によって示された 2D 行列のバイトのメモリの幅です。それぞれの行の終端に加えるあらゆる埋込みを含みます(SectionD.5.2 を参照下さい)。

この関数は間隔が **cudaMallocPitch()**により後ろへ返されて1の時に最速で実行します。

D.5.10 cudaMemcpy()

```
cudaError_t cudaMemcpy(void* dst, const void* src,
                       size_t count,
                       enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyAsync(void* dst, const void* src,
                            size_t count,
                            enum cudaMemcpyKind kind,
                            cudaStream_t stream);
```

src によって示されたメモリ領域から **dst** によって示されたメモリ領域まで **count** バイトをコピーします。ここで **kind** は **cudaMemcpyHostToHost**、**cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost** か **cudaMemcpyDeviceToDevice** の1つで、コピーの方向を指定します。メモリ領域はおそらくオーバーラップしません。**dst** と **src** がある **cudaMemcpy()**をコピーの指示に合わないポインタをコールしていると、未定義の動作がもたらされます。

cudaMemcpyAsync()は、非同期であり、非ゼロ **stream** 引数を渡すことによって、ストリームに任意に関連づけることができます。

ページ可能メモリへのポインタが入力されるように渡されるなら、ページーロック・ホスト・メモリに働くだけであって、誤りを返します。

D.5.11 cudaMemcpy2D()

```
cudaError_t cudaMemcpy2D(void* dst, size_t dpitch,
                         const void* src, size_t spitch,
                         size_t width, size_t height,
                         enum cudaMemcpyKind kind);
cudaError_t cudaMemcpy2DAsync(void* dst, size_t dpitch,
                              const void* src, size_t spitch,
                              size_t width, size_t height,
                              enum cudaMemcpyKind kind,
                              cudaStream_t stream);
```

src により指示されたメモリ領域から行列（**width** バイトの **height** 行各々)を **dst** により示されたメモリ領域へコピーします。ここで種類は **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost** または **cudaMemcpyDeviceToDevice** の1つで、コピーの方向を指定します。**dpitch** 及び **spitch** は **dst** 及び **src** により指示された2D 行列のバイト内メモリの幅で、各列の終端へのあらゆる埋込みを含みます（SectionD.5.2 を参照下さい）。メモリ領域はおそらくオーバーラップしません。

dst 及び **src** がある **cudaMemcpy2D()**をコピーの指示に合わないポインタをコールしていると、未定義の動作がもたらされます。

cudaMemcpy2D()はもし **dpitch** か **spitch** が最大許容（SectionD.1.4 の **memPitch** を参照下さい）を超えるとエラーを返します。

cudaMemcpy2DAsync()は非同期で、非ゼロ **stream** 引数をパスすることでストリームへオプションで関連付けられます。それはページ・ロック ホスト・メモリでのみで動作し、もしページ可能メモリへのポインタが入力されるように渡されるとエラーを返します。

D.5.12 cudaMemcpyToArray()

```
cudaError_t cudaMemcpyToArray(struct cudaArray* dstArray,
                              size_t dstX, size_t dstY,
                              const void* src, size_t count,
                              enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyToArrayAsync(struct cudaArray* dstArray,
                                   size_t dstX, size_t dstY,
                                   const void* src, size_t count,
                                   enum cudaMemcpyKind kind,
                                   cudaStream_t stream);
```

左上隅(**dstX**, **dstY**)で始まって、**src** によって示されたメモリ領域から CUDA 行列 **dstArray** まで **count** バイトをコピーします。ここで種類は **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**,か **cudaMemcpyDeviceToDevice** の1つで、コピーの方向を指定します。

cudaMemcpyToArrayAsync()は非同期で、非ゼロ **stream** 引数を渡すことでストリームはオプションで関連付けられます。それはページ・ロック ホスト・メモリでのみで動作し、もしページ可能メモリへのポインタが入力されるように渡されるとエラーを返します。

D.5.13 cudaMemcpy2DToArray()

```
cudaError_t cudaMemcpy2DToArray(struct cudaArray* dstArray,
                                size_t dstX, size_t dstY,
                                const void* src, size_t spitch,
                                size_t width, size_t height,
                                enum cudaMemcpyKind kind);
cudaError_t cudaMemcpy2DToArrayAsync(struct cudaArray* dstArray,
                                     size_t dstX, size_t dstY,
                                   const void* src, size_t spitch,
                                     size_t width, size_t height,
                                     enum cudaMemcpyKind kind,
                                     cudaStream_t stream);
```

左上隅(**dstX**, **dstY**)で始まって、**src** によって示されたメモリ領域から CUDA 行列 **dstArray** まで行列(**width** バイトの **height** 行各々)をコピーします。ここで種類は **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost** か **cudaMemcpyDeviceToDevice** の1つで、コピーの方向を指定します。**Spitch** は **src** により指定された2D 行列のバイトのメモリの幅で、各列の終端へのあらゆる埋込みを含みます(SectionD.5.2 を参照下さい。**cudaMemcpy2D()** はもし **spitch** か **spitch** が最大許容(SectionD.1.4 の **memPitch** を参照下さい)を超えるとエラーを返します。

cudaMemcpy2DToArrayAsync()は非同期で、非ゼロ **stream** 引数をパスすることでストリームへオプションで関連付けられます。それはページ・ロック ホスト・メモリでのみで動作し、もしページ可能メモリへのポインタが入力されるように渡されるとエラーを返します。

D.5.14 cudaMemcpyFromArray()

```
cudaError_t cudaMemcpyFromArray(void* dst,
                                const struct cudaArray* srcArray,
                                size_t srcX, size_t srcY,
                                size_t count,
                                enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyFromArrayAsync(void* dst,
                                const struct cudaArray* srcArray,
                                     size_t srcX, size_t srcY,
                                     size_t count,
                                     enum cudaMemcpyKind kind,
                                     cudaStream_t stream);
```

左上隅(**srcX**, **srcY**)で始まって CUDA 行列 **srcArray** から **dst** によって示されたメモリ領域まで **count** バイトをコピーします。ここで種類は **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost** か **cudaMemcpyDeviceToDevice** の１つで、コピーの方向を指定します。

cudaMemcpyFromArrayAsync()は非同期で、非ゼロ **stream** 引数を渡すことでストリームはオプションで関連付けられます。それはページ・ロック ホスト・メモリでのみで動作し、もしページ可能メモリへのポインタが入力されるように渡されるとエラーを返します。

D.5.15 cudaMemcpy2DFromArray()

```
cudaError_t cudaMemcpy2DFromArray(void* dst, size_t dpitch,
                                const struct cudaArray* srcArray,
                                 size_t srcX, size_t srcY,
                                 size_t width, size_t height,
                                 enum cudaMemcpyKind kind);
cudaError_t cudaMemcpy2DFromArrayAsync(void* dst, size_t dpitch,
                                const struct cudaArray* srcArray,
                                      size_t srcX, size_t srcY,
                                     size_t width, size_t height,
                                     enum cudaMemcpyKind kind,
                                      cudaStream_t stream);
```

それは左上隅(**srcX**, **srcY**)で始まる CUDA 行列 **srcArray** から **dst** によって示されたメモリ空間まで行列（**width** バイトの **height** 行各々)をコピーします。ここで種類は **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost** か **cudaMemcpyDeviceToDevice** の１つで、コピーの方向を指定します。**dpitch** は **dst** により指定された2D 行列のバイトのメモリの幅で、各列の終端へのあらゆる埋込みを含みます（SectionD.5.2 を参照下さい。**cudaMemcpy2D()** はもし **dpitch** が最大許容（SectionD.1.4 の **memPitch** を参照下さい）を超えるとエラーを返します。

cudaMemcpy2DToArrayAsync()は非同期で、非ゼロ **stream** 引数をパスすることでストリームへオプションで関連付けられます。それはページ・ロック ホスト・メモリでのみで動作し、もしページ可能メモリへのポインタが入力されるように渡されるとエラーを返します。

D.5.16 cudaMemcpyArrayToArray()

```
cudaError_t cudaMemcpyArrayToArray(struct cudaArray* dstArray,
                                   size_t dstX, size_t dstY,
                                   const struct cudaArray* srcArray,
                                   size_t srcX, size_t srcY,
                                   size_t count,
                                   enum cudaMemcpyKind kind);
```

左上隅(**dstX**, **dstY**),で始まって、左上隅(**srcX**, **srcY**)で始まる CUDA 行列 **srcArray** から CUDA 行列 **dstArray** まで **count** バイトをコピーします。ここで種類は **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost** か **cudaMemcpyDeviceToDevice** の1つで、コピーの方向を指定します。

D.5.17 cudaMemcpy2DArrayToArray()

```
cudaError_t cudaMemcpy2DArrayToArray(struct cudaArray* dstArray,
                                     size_t dstX, size_t dstY,
                                     const struct cudaArray* srcArray,
                                     size_t srcX, size_t srcY,
                                     size_t width, size_t height,
                                     enum cudaMemcpyKind kind);
```

左上隅(**dstX**, **dstY**),で始まって、それは左上隅(**srcX**, **srcY**)で始まる CUDA 行列 **srcArray** から CUDA 行列 **dstArray** まで行列(**width** バイトの **height** 行各々)をコピーします。ここで種類は **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost** か **cudaMemcpyDeviceToDevice** の1つで、コピーの方向を指定します。

D.5.18 cudaMemcpyToSymbol()

```
template<class T>
cudaError_t cudaMemcpyToSymbol(const T& symbol, const void* src,
                               size_t count, size_t offset = 0,
               enum cudaMemcpyKind kind = cudaMemcpyHostToDevice);
```

src によって示されたメモリ領域から **offset** バイトによってシンボル **symbol** の始まりから示されたメモリ領域まで **count** バイトをコピーします。メモリ領域はオーバーラップしないかもしれません。**symbol** は、グローバルにある変数か定数メモリ空間のどちらかがキャラクタ文字列であるかもしれません、グローバルか定数メモリ空間にある変数を命名して。

種類は、**cudaMemcpyHostToDevice** か **cudaMemcpyDeviceToDevice** のどちらかであるかもしれません。

D.5.19 cudaMemcpyFromSymbol()

```
template<class T>
cudaError_t cudaMemcpyFromSymbol(void *dst, const T& symbol,
                                 size_t count, size_t offset = 0,
               enum cudaMemcpyKind kind = cudaMemcpyDeviceToHost);
```

offset バイトによってシンボル **symbol** の始まりから示されたメモリ領域から **dst** によって示されたメモリ領域まで **count** バイトをコピーします。メモリ領域はおそらくオーバーラップしません。**symbol** は、グローバルにある変数か定数メモリ空間のどちらかがキャラクタ文字列であるかもしれません、グローバルか定数メモリ空間にある変数を命名して。種類は、**cudaMemcpyDeviceToHost** か **cudaMemcpyDeviceToDevice** のどちらかであるかもしれません。

D.5.20 cudaGetSymbolAddress()

```
template<class T>
cudaError_t cudaGetSymbolAddress(void** devPtr, const T& symbol);
```

デバイスにて ***devPtr** でシンボル **symbol** のアドレスを返します。**symbol** はグローバル・メモリ空間にある変数であるかそれがキャラクタ文字列であるかもしれません、グローバル・メモリ空間にある変数を命名して。**symbol** を見つけることができないか、または **symbol** がグローバルメモリ空間で宣言されないなら、***devPtr** は変わりなく、誤りは返されます。**cudaGetSymbolAddress()**は失敗の場合に **cudaErrorInvalidSymbol** を返します。

D.5.21 cudaGetSymbolSize()

```
template<class T>
cudaError_t cudaGetSymbolSize(size_t* size, const T& symbol);
```

size** でシンボル **symbol** のサイズを返します。**symbol** はグローバルにある変数か定数メモリ空間のどちらかで、それはキャラクタ文字列であるかもしれません、グローバルか定数メモリ空間にある変数を命名して。**symbol** を見つけることができないか、または **symbol** がグローバルメモリ空間で宣言されないなら、size** は変わりなく、誤りは返されます。**cudaGetSymbolSize()**は失敗の場合に **cudaErrorInvalidSymbol** を返します。

D.6 テクスチャ参照管理

D.6.1 低レベル API

D.6.1.1 cudaCreateChannelDesc()

```
struct cudaChannelFormatDesc
cudaCreateChannelDesc(int x, int y, int z, int w,
                      enum cudaChannelFormatKind f);
```

それぞれのコンポーネント x, y, z 及び w のビットのフォーマット f と数と共にチャンネル記述子を返します、y、z、w.。**cudaChannelFormatDesc** は Section4.3.4 で記述されます。

D.6.1.2 cudaGetChannelDesc()

```
cudaError_t cudaGetChannelDesc(struct cudaChannelFormatDesc* desc,
                               const struct cudaArray* array);
```

***desc** で CUDA 行列 **array** に関するチャンネル記述子を返します。

D.6.1.3 cudaGetTextureReference()

```
cudaError_t cudaGetTextureReference(
```

```
                          struct TextureReference** texRef,
                          const char* symbol);
```

***texRef** でシンボル **symbol** によって定義されたテクスチャ参照に関連づけた構造を返します。

D.6.1.4 cudaBindTexture()

```
cudaError_t cudaBindTexture(size_t* offset,
                      const struct TextureReference* texRef,
                      const void* devPtr,
                   const struct cudaChannelFormatDesc* desc,
                      size_t size = UINT_MAX);
```

テクスチャ参照 **texRef** に **devPtr** によって示されたメモリ領域の **size** バイトを拘束します。**desc** はテクスチャから値をとって来るとき、メモリがどう解釈されるかを記述します。全てのメモリは texRef へのバウンド前は非バウンドです。ハードウェアが整列要求をテクスチャ・ベース・アドレスに押しつけるので、**cudaBindTexture()**は***offset** 1 バイトで必要なメモリから読出すためにテクスチャ・フェッチに適用しなければならないオフセットを返します。それらを **tex1Dfetch()**関数に適用できるように、このオフセットはテクセル・サイズが割られて、テクスチャから読んだカーネルに渡さなければなりません。もし、**cudaMalloc()**からデバイス・メモリ・ポインタを返したなら、オフセットは 0 になるように保証されます、そして、NULL はオフセットパラメタとして渡されるかもしれません。

D.6.1.5 cudaBindTextureToArray()

```
cudaError_t cudaBindTextureToArray(
                      const struct TextureReference* texRef,
                      const struct cudaArray* array,
                   const struct cudaChannelFormatDesc* desc);
```

テクスチャ参照 **texRef** に CUDA 行列 **array** を拘束します。**desc** はテクスチャから値をとって来るとき、メモリがどう解釈されるかを記述します。あらゆる CUDA 行列は **texRef** に拘束する前は非拘束です。

D.6.1.6 cudaUnbindTexture()

```
cudaError_t cudaUnbindTexture(
                     const struct textureReference* texRef);
```

テクスチャ・バウンドをテクスチャ参照 texRef に非拘束します。

D.6.1.7 cudaGetTextureAlignmentOffset()

```
cudaError_t cudaGetTextureAlignmentOffset(size_t* offset,
                     const struct textureReference* texRef);
```

***offse** でテクスチャ参照 **texRef** が拘束であったときに返されたオフセットを返します。

D.6.2 高レベル API

D.6.2.1 cudaCreateChannelDesc()

```
template<class T>
struct cudaChannelFormatDesc cudaCreateChannelDesc<T>();
```

フォーマットが型 T に合っていて、それはチャンネル記述子を返します。T は Section4.3.1.1 のどんな型でもあるかもしれません。3 コンポーネントの型は 4 コンポーネント・フォーマットをデフォルトとします。

D.6.2.2 cudaBindTexture()

```
template<class T, int dim, enum cudaTuxtureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaBindTeture(size_t* offset,
               const structTexture<T, dim, readMode>& texRef,
               const void* devPtr,
               const struct cudaChannelFormatDesc& desc,
               size_t size = UINT_MAX);
```

テクスチャ参照 texRef に devPtr によって示されたメモリ領域の **size** バイトを拘束します。**desc** はテクスチャから値をとって来るとき、メモリがどう解釈されるかを記述します。オフセット・パラメタは SectionD.6.1.4 に記述された低レベル **cudaBindTexture()**関数としてオプションのバイト・オフセットです。あらゆるメモリは **texRef** に拘束される前は非拘束です。.

```
template<class T, int dim, enum cudaTeture ReadMode readMode>
static __inline__ __host__ cudaError_t
cudaBindTexture(size_t* offset,
               const structTexture<T, dim, readMode>& texRef,
               const void* devPtr,
               size_t size = UINT_MAX);
```

参照 **texRef** に **devPtr** によって示されたメモリ領域の **size** バイトを拘束します。チャンネル記述子はテクスチャ参照型から引き継がれます。オフセット・パラメタは SectionD.6.1.4 に記述された低レベル **cudaBindTexture()**関数としてオプションのバイト・オフセットです。あらゆるメモリは **texRef** に拘束される前は非拘束です。

D.6.2.3 cudaBindTextureToArray()

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaBindTextureToArray(
                 const structTexture<T, dim, readMode>& texRef,
                 const struct cudaArray* cuArray,
                 const struct cudaChannelFormatDesc& desc);
```

テクスチャ参照 **texRef** に CUDA 行列 **array** を拘束します。**desc** はテクスチャから値をとって来るとき、メモリがどう解釈されるかを記述します。あらゆるメモリは **texRef** に拘束される前は非拘束です。

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaBindTextureToArray(
                 const structTexture<T, dim, readMode>& texRef,
                 const struct cudaArray* cuArray);
```

テクスチャ参照 **texRef** に CUDA 行列 **array** を拘束します。チャンネル記述子は CUDA 行列から引き継がれます。あらゆるメモリは **texRef** に拘束される前は非拘束です。

D.6.2.4 cudaUnbindTexture()

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaUnbindTexture(const structTexture<T, dim, readMode>& texRef);
```

テクスチャ拘束をテクスチャ参照 texRef に非拘束します。

D.7 実行制御

D.7.1 cudaConfigureCall()

```
cudaError_t cudaConfigureCall(dim3 gridDim, dim3 blockDim,
                              size_t sharedMem = 0,
                              int tokens = 0);
```

デバイス・コールが Section4.2.3 で記述した実行構成構文と同様の状態で実行されるためにグリッドとブロック次数を指定します。

cudaConfigureCall() はスタック・ベースです。各コールは実行スタックの上でデータをプッシュします。このデータはグリッドのための次数と、コールのためのあらゆる引数のスレッドブロックを含んでいます。

D.7.2 cudaLaunch()

```
template<class T> cudaError_t cudaLaunch(T entry);
```

デバイスの関数 **entry** で起動します。**entry** はデバイスで実行する関数か、それがデバイスで実行する関数を命名するか、キャラクタ文字列であるかもしれません。**Entry** は**_global_** 関数として宣言されなければなりません。

cudaConfigureCall()によって実行スタックからプッシュしたデータをポップさせるので、コールは**cudaConfigureCall()**に **cudaLaunch()**に先行しなければなりません。

D.7.3 cudaSetupArgument()

```
cudaError_t cudaSetupArgument(void* arg,
                              size_t count, size_t offset);
template<class T> cudaError_t cudaSetupArgument(T arg,
                                                size_t offset);
```

パラメタ通過領域の始まりから **offset** バイトの **arg** により指定された引数の **count** をプッシュします。それはオフセット 0 で始まります。引数は実行スタックの先頭に格納されます。**cudaSetupArgument()**は **cudaConfigureCall()**へのコールにより準備されなければなりません。

D.8 OpenGL 相互運用性

D.8.1 cudaGLRegisterBufferObject()

```
cudaError_t cudaGLRegisterBufferObject(GLuint bufferObj);
```

CUDA によるアクセスのために ID bufferObj のバッファ・オブジェクトを登録します。CUDA がバッファ・オブジェクトをマップすることができる前に、この関数を呼ばなければなりません。それが登録されている間は、コマンドを引き出す OpenGL のためのデータ送信端末以外の、どんな OpenGL コマンドでもバッファオブジェクトを使用することはできません。

D.8.2 cudaGLMapBufferObject()

```
cudaError_t cudaGLMapBufferObject(void** devPtr,
                                  GLuint bufferObj);
```

ID **bufferObj** のバッファ・オブジェクトを CUDA のアドレス空間にマップして、***devPtr** の結果として起こるマッピングのベース・ポインタを返します。

D.8.3 cudaGLUnmapBufferObject()

```
cudaError_t cudaGLUnmapBufferObject(GLuint bufferObj);
```

CUDA によるアクセスのために ID **bufferObj** のバッファ・オブジェクトを非マップします。

D.8.4 cudaGLUnregisterBufferObject()

```
cudaError_t cudaGLUnregisterBufferObject(GLuint bufferObj);
```

CUDA によるアクセスのために ID bufferObj のバッファ・オブジェクトを非登録します。

D.9 Direct3D 相互運用性

D.9.1 cudaD3D9Begin()

```
cudaError_t cudaD3D9Begin(IDirect3DDevice9* device);
```

Direct3D デバイス **device** と共に相互運用性を初期化します。CUDA が **device** からあらゆるオブジェクトをマップすることができる前に、この関数をコールしなければなりません。そのアプリケーションは **cudaD3D9End()**がコールされるまで Direct3D デバイスによって所有されていた頂点バッファをマップできます。

D.9.2 cudaD3D9End()

```
cudaError_t cudaD3D9End(void);
```

前に **cudaD3D9Begin()**.を指定した Direct3D デバイスの相互運用性を完結します。

D.9.3 cudaD3D9RegisterVertexBuffer()

```
cudaError_t
cudaD3D9RegisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

CUDA によりアクセスするための頂点バッファ **VB** を登録します。

D.9.4 cudaD3D9MapVertexBuffer()

```
cudaError_t cudaD3D9MapVertexBuffer(void** devPtr,
                                    IDirect3DVertexBuffer9* VB);
```

頂点バッファ **VB** を現在の CUDA コンテクストのアドレス空間にマップし、***devPtr** の結果としてのマッピングのベース・ポインタを返します。

D.9.5 cudaD3D9UnmapVertexBuffer()

```
cudaError_t cudaD3D9UnmapVertexBuffer(IDirect3DVertexBuffer9* VB);
```

CUDA によるアクセスのために頂点バッファ VB を非マップします。

D.9.6 cudaD3D9UnregisterVertexBuffer()

```
cudaError_t
cudaD3D9UnregisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

CUDA によるアクセスのために頂点バッファ VB を非登録します。

D.9.7 cudaD3D9GetDevice()

```
cudaError_t
cudaD3D9GetDevice(int* dev, const char* adapterName);
```

EnumDisplayDevices か **IDirect3D9::GetAdapterIdentifier** から得られたアダプター名 **adapterName** に対応するデバイスを***dev** に返します。

D.10 エラーの取り扱い

D.10.1 cudaGetLastError()

```
cudaError_t cudaGetLastError(void);
```

cudaSuccess への結果と同じホスト・スレッド内のランタイム・コールのあらゆるものから返された最後のエラーを返します。

D.10.2 cudaGetErrorString()

```
const char* cudaGetErrorString(cudaError_t error);
```

エラーコードからのメッセージ文字列を返します。

Appendix E.
ドライバ API 参照

E.1 初期化

E.1.1 cuInit()

```
CUresult cuInit(unsigned int Flags);
```

ドライバ API を初期化し、ドライバ API からあらゆる他の関数の前にコールされなければなりません。現在、**Flags** パラメータは 0 でなければなりません。もし **cuInit()**コールされていないなら、ドライバ API からのあらゆる関数は **CUDA_ERROR_NOT_INITIALIZED** を返します。

E.2 デバイス管理

E.2.1 cuDeviceGetCount()

```
CUresult cuDeviceGetCount(int* count);
```

***count** で実行可能な演算能力 1.0 以上のデバイスの数を返します。もし、デバイスがないようなら、**cuDeviceGetCount()**は 1 を返し、デバイス 0 はデバイス・エミュレーション・モードをサポートするだけであって、演算能力 1.0 未満のものです。

E.2.2 cuDeviceGet()

```
CUresult cuDeviceGet(CUdevice* dev, int ordinal);
```

範囲[**0, cuDeviceGetCount()-1**]内の序数を与えられたデバイス・ハンドル***dev** に返します。

E.2.3 cuDeviceGetName()

```
CUresult cuDeviceGetName(char* name, int len, CUdevice dev);
```

name によって示された NULL-終了文字列内のデバイス **dev** を特定する ASCII 文字列を返します。**len** は返されるかもしれない文字列の最長の長さを指定します。

E.2.4 cuDeviceTotalMem()

```
CUresult cuDeviceTotalMem(unsigned int* bytes, CUdevice dev);
```

***bytes** にてバイト内のデバイス **dev** で利用可能なメモリの総量を返します。

E.2.5 cuDeviceComputeCapability()

```
CUresult cuDeviceComputeCapability(int* major, int* minor,
                                   CUdevice dev);
```

major** とminor** でデバイス **dev** の演算能力を定義するメジャーとマイナーのレビジョン番号を返します。

E.2.6 cuDeviceGetAttribute()

```
CUresult cuDeviceGetAttribute(int* value,
                              CUdevice_attribute attrib,
                              CUdevice dev);
```

***value** でデバイス **dev** の属性 **attrib** の整数値を返します。サポートされる属性は：

- **CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK**: ブロックあたりのスレッドの最大数；
- **CU_DEVICE_ ATTRIBUTE_MAX_BLOCK_DIM_X**: ブロックの最大 x-次数；
- **CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y**: ブロックの最大 y-次数；
- **CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z**: ブロックの最大 z-次数；
- **CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X**: グリッドの最大 x-次数；
- **CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y**: グリッドの最大 y-次数；
- **CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z**: グリッドの最大 z-次数；
- **CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK**: バイト内のブロックあたりのシェアード・メモリの可能総数；
- **CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY**: バイト内のブロックあたりのデバイスの可能総数；
- **CU_DEVICE_ATTRIBUTE_WARP_SIZE**: ワープ・サイズ；
- **CU_DEVICE_ATTRIBUTE_MAX_PITCH**: SectionE.8 のメモリ・コピー関数により許容された最大間隔で、それは cuMemAllocPitch()を経由して割り当てられたメモリ領域にかかわります（SectionE8.3）；
- **CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK**: ブロックあたりの登録可能総数；

- **CU_DEVICE_ATTRIBUTE_CLOCK_RATE**: クロック周波数（キロ Hz）;
- **CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT**: Section4.3.4.3 で言及している整列要求；テクスチャ・ベース・アドレスで、それはテクスチャ・フェッチに適用したオフセットの必要のない **TextureAlign** に整列されます；
- **CU_DEVICE_ATTRIBUTE_GPU_OVERLAP**:もし、デバイスがカーネルを実行している間に同時にホストとデバイスの間にメモリをコピーすることができるなら 1 で、もし否なら 0 です。

E.2.7 cuDeviceGetProperties()

```
CUresult cuGetDeviceProperties(CUdevprop* prop, CUdevice dev);
```

デバイス **dev** の ***prop** 特性へ返します。**Cudevprop** 構造は次のものとして定義されます：

```
typedef struct CUdevprop_st {
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    int sharedMemPerBlock;
    int totalConstantMemory;
    int SIMDWidth;
    int memPitch;
    int regsPerBlock;
    int clockRate;
    intTextureAlign;
} CUdevprop;
```

ここで：

- **maxThreadsPerBlock** はブロックあたりのスレッドの総数です：
- **maxThreadsDim[3]** はブロックの各次数の最大サイズです；
- **maxGridSize[3]** はグローバル・メモリの各次数の最大サイズです；
- **sharedMemPerBlock** はバイト内のブロックあたりのシェアード・メモリの総数です；
- **totalConstantMemory** はバイト内のデバイスの定数メモリの可能総数です；
- **SIMDWidth** はワープ・サイズです；
- **memPitch** は SectionE.8 のメモリ・コピー関数により許容される最大間隔で、それはメモリ **cuMemAllocPitch** を経由して割り当てられた領域にかかわります（SectionE.8.3）；
- **regsPerBlock** はブロックあたりの登録の可能総数です；
- **clockRate** はクロック周波数です（キロ Hz）；
- **textureAlign** は Section4.3.4.3 で言及している整列要求です；テクスチャ・ベース・アドレスはテクスチャ・フェッチへ適応されるオフセットの必要のない **textureAlign** バイトへ整列されます。

E.3 コンテクスト管理

E.3.1 cuCtxCreate()

```
CUresult cuCtxCreate(CUcontext* pCtx, unsigned int Flags, CUdevice dev);
```

デバイスのために新しいコンテクストを作成して、それをコールしているスレッドに関連づけます。現在では **Flags** パラメータは 0 でなければなりません。コンテクストは 1 の用法カウントで作成されます、そして、コンテクストを使用し終わっていると、**cuCtxCreate()**の呼び出し元は **cuCtxDetach()** をコールしなければなりません。この関数はコンテクストが既にスレッドの流れであるなら失敗します。

E.3.2 cuCtxAttach()

```
CUresult cuCtxAttach(CUcontext* pCtx, unsigned int Flags);
```

アプリケーションがコンテクストで終了すると、コンテクストの用法カウントを増加し、***pCtx** のコンテクスト・ハンドルはパス・バックし、**cuCtxDetach()** へパスしなければなりません。スレッドへのコンテクストの流れが全くなければ、**cuCtxAttach()**は失敗します。現在では **Flags** パラメータは 0 でなければなりません。

E.3.3 cuCtxDetach()

```
CUresult cuCtxDetach(CUcontext ctx);
```

コンテクストの用法カウントが減少して、用法カウントが 0 まで行ったら、コンテクストを破壊します。コンテクストは **cuCtxCreate()**か **cuCtxAttach()**によって戻されたハンドルでなくてはならず、コールしているスレッドへ流れなければなりません。

E.3.4 cuCtxGetDevice()

```
CUresult cuCtxGetDevice(CUdevice* device);
```

***device** で現在のコンテクストのデバイスの序数を返します。

E.3.5 cuCtxSynchronize()

```
CUresult cuCtxSynchronize(void);
```

デバイスがすべての前に要求されたタスクを完成するまで、ブロックします。前のタスクの 1 つが失敗したなら、cuCtxSynchronize()はエラーを返します。

E.4 モジュール管理

E.4.1 cuModuleLoad()

```
CUresult cuModuleLoad(CUmodule* mod, const char* filename);
```

ファイル名 **filename** を取って、対応するモジュール Y を現在のコンテクストにロードします。CUDA ドライバ API は、モジュールによって必要とされたリソースを割り当てるのを怠惰に試みません；モジュールによって必要とされた関数とデータ(定数とグローバル)のためのメモリを割り当てることができないなら、**cuModuleLoad()**は失敗します。ファイルは **nvcc** による出力として、*cubin* ファイルにしなければなりません（Section4.2.5 を参照下さい）。

E.4.2 cuModuleLoadData()

```
CUresult cuModuleLoadData(CUmodule* mod, const void* image);
```

ポインタ **image** を取って、対応するモジュール **mod** を現在のコンテクストにロードします。テキスト文字列として *cubin* ファイルを渡して、*cubin* ファイルをマップすることによって、ポインターを得るか、または *cubin* オブジェクトを実行可能なリソースに組み入れ、ポインターを入手するのに Windows の **FindResource()**などのオペレーション・システムコールを使用します。

E.4.3 cuModuleLoadFatBinary()

```
CUresult cuModuleLoadFatBinary(CUmodule* mod, const void* fatBin);
```

ポインター**fatBin** を取って、対応するモジュール **mod** を現在のコンテクストにロードします。ポインターは異なった *cubin* ファイルのコレクションである *fat binary* オブジェクトを表し、全ては同じデバイス・コードを表していますが、異なるアーキテクチャのためにコンパイルと最適化されます。現在は構築のための API の書類はなく、プログラマによるファット・バイナリ・オブジェクトを使っていますので、この関数は CUDA のこのバージョンの内部関数です。さらなる情報は **nvcc** 書類で見いだすことができます。

E.4.4 cuModuleUnload()

```
CUresult cuModuleUnload(CUmodule mod);
```

現在のコンテクストからモジュール **mod** を非ロードします。

E.4.5 cuModuleGetFunction()

```
CUresult cuModuleGetFunction(CUfunction* func,
                             CUmodule mod, const char* funcname);
```

***func** でモジュール **mod** で見つけられた名前 **funcname** の関数のハンドルを返します。その名前の関数が全く存在していないなら、**cuModuleGetFunction()**は **CUDA_ERROR_NOT_FOUND**.を返します。

E.4.6 cuModuleGetGlobal()

```
CUresult cuModuleGetGlobal(CUdeviceptr* devPtr,
                           unsigned int* bytes,
                           CUmodule mod, const char* globalname);
```

devPtr** とbytes** では、モジュール **mod** で見つけられた名前 **globalname** のグローバルのベース・ポインターとサイズを返します。その名前の変数が全く存在していないなら、**cuModuleGetGlobal()**は **CUDA_ERROR_NOT_FOUND** を返します。**devPtr** 及び **bytes** の両方のパラメータはオプションです。もし、それらの１つがヌルなら、それは無視されます。

E.4.7 cuModuleGetTexRef()

```
CUresult cuModuleGetTexRef(CUtexref* texRef,
                           CUmodule hmod, const char* texrefname);
```

***texref** でモジュール **mod** の名前 **texrefname** のテクスチャ参照のハンドルを返します。その名前のテクスチャ参照が全く存在していないなら、cuModuleGetTexRef()は **CUDA_ERROR_NOT_FOUND** を返します。モジュールが非ロードされるとき、それが破棄されるので、このテクスチャ参照ハンドルを破棄するべきではありません。

E.5 ストリーム管理

E.5.1 cuStreamCreate()

```
CUresult cuStreamCreate(CUstream* stream, unsigned int flags);
```

ストリームを生成します。現在のところ **flags** は 0 になるために必要です。

E.5.2 cuStreamQuery()

```
CUresult cuStreamQuery(CUstream stream);
```

もし、全てのストリームの操作が完了するか **CUDA_ERROR_NOT_READY** でないなら、**CUDA_SUCCESS** を返します。

E.5.3 cuStreamSynchronize()

```
CUresult cuStreamSynchronize(CUstream stream);
```

ストリームの全ての操作がデバイスで完了するまでブロックします。

E.5.4 cuStreamDestroy()

```
CUresult cuStreamDestroy(CUstream stream);
```

ストリームを破棄します。

E.6 イベント管理

E.6.1 cuEventCreate()

```
CUresult cuEventCreate(CUevent* event, unsigned int flags);
```

イベントを生成します。現在のところ **flags** は 0 になるために必要です。

E.6.2 cuEventRecord()

```
CUresult cuEventRecord(CUevent event, CUstream stream);
```

イベントを記録します。もし **stream** が非ゼロであるなら、ストリームのすべての処理されている操作が完了した後にイベントは記録されています；さもなければ、CUDA コンテクストのすべての処理している操作が完了した後にそれは記録されます。その後はこの操作は非同期で、イベントがいつ実際に記録されたかを決定するのに **cuEventQuery()** 及び／または **cuEventSynchronize()** を使用しなければなりません。もし、**cuEventRecord()**が以前にコールされて、イベントがまだ記録されていないなら、この関数は **CUDA_ERROR_INVALID_VALUE** を返します。

E.6.3 cuEventQuery()

```
CUresult cuEventQuery(CUevent event);
```

イベントが実際に記録されたか、**CUDA_ERROR_NOT_READY** でなければ、**CUDA_SUCCESS** を返します。もし、**cuEventRecord()**がこのイベントでコールせれていないなら、関数は **CUDA_ERROR_INVALID_VALUE** を返します。

E.6.4 cuEventSynchronize()

```
CUresult cuEventSynchronize(CUevent event);
```

イベントが実際に記録されるまでブロックします。もし **cuEventRecord()**がイベントでコールされていないなら、関数は **CUDA_ERROR_INVALID_VALUE** を返します。

E.6.5 cuEventDestroy()

```
CUresult cuEventDestroy(CUevent event);
```

イベントを破棄します。

E.6.6 cuEventElapsedTime()

```
CUresult cuEventDestroy(float* time,
                        CUevent start, CUevent end);
```

2つのイベント間（約 0.5 ミクロ秒のリゾリューションでのミリ秒）の経過時間を計算します。もし、どちらかのイベントが未だ記録されていないなら、この関数は **CUDA_ERROR_INVALID_VALUE** を返します。もし、どちらかのイベントが未だ非セロ・ストリームで記録されていないなら、結果は未定義です。

E.7 実行制御

E.7.1 cuFuncSetBlockShape()

```
CUresult cuFuncSetBlockShape(CUfunction func,
                             int x, int y, int z);
```

func の起動によりカーネルを与えられたときに、スレッド・ブロックの X, Y 及び Z の次数を指示します。

E.7.2 cuFuncSetSharedSize()

```
CUresult cuFuncSetSharedSize(CUfunction func, unsigned int bytes);
```

func の起動によりカーネルを与えられたときに、**bytes** を経由して各スレッド・ブロックに利用可能になるシェアード・メモリの量を設定します。

E.7.3 cuParamSetSize()

```
CUresult cuParamSetSize(CUfunction func, unsigned int numbytes);
```

関数 **func** の関数パラメータにより必要とされるバイトの合計サイズ **numbytes** を経由して設定します。

E.7.4 cuParamSeti()

```
CUresult cuParamSeti(CUfunction func,
                     int offset, unsigned int value);
```

func に対応するカーネルが呼び出される次の時に指定される整数パラメータを設定します。**offset** はバイト・オフセットです。

E.7.5 cuParamSetf()

```
CUresult cuParamSetf(CUfunction func,
                     int offset, float value);
```

func に対応するカーネルが呼び出される次の時に指定される浮動小数点パラメータを設定します。**offset** はバイト・オフセットです。

E.7.6 cuParamSetv()

```
CUresult cuParamSetv(CUfunction func,
                     int offset, void* ptr,
                     unsigned int numbytes);
```

func に対応するカーネルのパラメータ空間に任意の量のデータをコピーします。**offset** はバイト・オフセットです。

E.7.7 cuParamSetTexRef()

```
CUresult cuParamSetTexRef(CUfunction func,
                          int texunit, CUtexref texRef);
```

CUDA をテクスチャとしてデバイス・プログラムに利用可能なテクスチャ参照 **texRef** への行列かリニア・メモリ拘束を作成します。CUDA のこのバージョンでは、**cuModuleGetTexRef()**を通してテクスチャ参照を入手しなければなりません、そして、**texunit** パラメータは **CU_PARAM_TR_DEFAULT** へ設定しなければなりません。

E.7.8 cuLaunch()

```
CUresult cuLaunch(CUfunction func);
```

ブロックの 1x1 グラフィックスのカーネル **func** を呼び出します。このブロックには **cuFuncSetBlockShape()**を前のコールにより指定されたスレッドの数を含んでいます。

E.7.9 cuLaunchGrid()

```
CUresult cuLaunchGrid(CUfunction func,
                      int grid_width, int grid_height);
CUresult cuLaunchGridAsync(CUfunction func,
                           int grid_width, int grid_height,
                           CUstream stream);
```

ブロックの **grid_width** × **grid_height** グリッドのカーネルを呼び出します。各ブロックには **cuFuncSetBlockShape()**を前のコールにより指定されたスレッドの数を含んでいます。

cuLaunchGridAsync()は非ゼロ **stream** 引数をパスすることでストリームへオプションで関連付けつことができます。それはページ・ロック ホスト・メモリで稼動し、もしページ可能メモリへのポインタが入力としてパスされたエラーを返します。

E.8 メモリ管理

E.8.1 cuMemGetInfo()

```
CUresult cuMemGetInfo(unsigned int* free, unsigned int* total);
```

***free** 及び ***total** に各々返します。バイトで表現される CUDA コンテクストによる配分に利用可能なメモリの自由及び合計の量。

E.8.2 cuMemAlloc()

```
CUresult cuMemAlloc(CUdeviceptr* devPtr, unsigned int count);
```

デバイスでのリニア・メモリの **count** バイトを割り当て、割り当てたメモリへの***devPtr** ポインタに返します。割り当てられたメモリはどんな種類の変数のためにも適切に並べられます。そのメモリはクリアにされません。もし、**count** が 0 なら、**cuMemAlloc()**は **CUDA_ERROR_INVALID_VALUE** を返します。

E.8.3 cuMemAllocPitch()

```
CUresult cuMemAllocPitch(CUdeviceptr* devPtr,
                         unsigned int* pitch,
                         unsigned int widthInBytes,
                         unsigned int height,
                         unsigned int elementSizeBytes);
```

デバイスでのリニア・メモリの最低 **widthInBytes*height** バイトを割り当て、割り当てたメモリへの***devPtr** ポインタに返します。関数は対応するポインターを確実にするために割り当てを埋め込むかもしれませんし、列から列までアドレスをアップデートされたものとして、与えられた列は、結合するための整理要求に応じし続けます（Section5.1.2.1 を参照下さい）。

elementSizeBytes は、最も大きい読出しサイズを指定して、メモリ範囲に実行されると書込みます。**elementSizeBytes** は 4、8 または 16 でしょう（結合しているメモリ・トランザクションは他のデータサイズで可能ではありません）。もし、**elementSizeBytes** が実際に読み／書きしたカーネルのサイズより小さいなら、カーネルは正しく稼動するでしょうが、ことによると速度が低下するかも知れません。***pitch** で **cuMemAllocPitch()**によって返されたピッチは割り当てられたバイトの幅です。ピッチの意図された用法は割り当ての別々のパラメタは 2D 行列の中でアドレスを計算するのに使用されます。型 T の行列要素の行と列を与えます。アドレスは次の計算によります：

```
T* pElement = (T*)((char*)BaseAddress + Row * Pitch) + Column;
```

cuMemAllocPitch()によって返されたピッチは、あらゆる環境のもとで **cuMemcpy2D()**と動作するのを保証されます。2D 行列の割り当ては、プログラマが cuMemAllocPitch()を使用することでピッチの割り当ての実行に検討することを推奨します。ハードウェアの整列制限が原因で、アプリケーションが異なるデバイス・メモリ領域の間の 2D メモリコピーを実行するなら、これは特に真です（リニア・メモリか CUDA 行列でありましても）。

E.8.4cuMemFree()

```
CUresult cuMemFree(CUdeviceptr devPtr);
```

devPtr により指定され、前のコールで **cuMemMalloc()** または **cuMemMallocPitch()**により返されたメモリ空間を開放します。

E.8.5cuMemAllocHost()

```
CUresult cuMemAllocHost(void** hostPtr, unsigned int count);
```

ページ・ロックでデバイスにアクセス可能なホスト・メモリの **count** バイトを割り当てます。ドライバはこの関数で割り当てられた仮想メモリ範囲をトラックし、**cuMemcpy()**のような関数のコールを自動的に加速します。メモリはデバイスによるアクセスを直接可能にすることで、**malloc()**.のような関数で獲得された、ページ可能メモリより高速な帯域幅で読み／書きすることができます。。ページングのシステムに利用可能なメモリの量を減少させることにより、**cuMemAllocHost()**でメモリの過剰な量を割り当てると、システム性能は低下するでしょう。結果として、ホストとデバイスの間のデータ交換のために、中間準備区域を割り当てるのにこの関数を控えめに使用するのは最良です。

E.8.6cuMemFreeHost()

```
CUresult cuMemFreeHost(void* hostPtr);
```

hostPtr により指定され、**cuMemAllocHost()**への前のコールによる返されるに違いないメモリ空間を開放します。

E.8.7cuMemGetAddressRange()

```
CUresult cuMemGetAddressRange(CUdeviceptr* basePtr,
                              unsigned int* size,
                              CUdeviceptr devPtr);
```

basePtr** とサイズと入力ポインター**devPtr** を含む **cuMemAlloc()** か **cuMemAllocPitch()**により割り当てたsize** にベース・アドレスを返します。

パラメータ **basePtr** 及び **size** の両方はオプションです。もし、それらの1つがヌルなら、それは無視されます。

E.8.8 cuArrayCreate()

```
CUresult cuArrayCreate(CUarray* array,
                       const CUDA_ARRAY_DESCRIPTOR* desc);
```

CUDA_ARRAY_DESCRIPTOR 構造 **desc** に従って CUDA 行列を作成し、ハンドルを***array** の新しい CUDA 行列に返します。**CUDA_ARRAY_DESCRIPTOR** 構造は次により定義されます：

```
typedef struct {
    unsigned int Width;
    unsigned int Height;
    CUarray_format Format;
    unsigned int NumChannels;
} CUDA_ARRAY_DESCRIPTOR;
```

ここで:

- **Width** 及び **Height** は（要素内の）CUDA 行列の幅と高さです；CUDA 行列は、もし **height** が 0 のときの次数は 1 で、それ以外のときの次数は 2 です；
- **NumChannels** は CUDA 行列要素あたりのパックしたコンポーネントの数を指定します；それは 1, 2 または 4 になります；
- **Format** は要素のフォーマットを指定します；**Format** は次のように定義します：

```
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8  = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8    = 0x08,
    CU_AD_FORMAT_SIGNED_INT16   = 0x09,
    CU_AD_FORMAT_SIGNED_INT32   = 0x0a,
    CU_AD_FORMAT_HALF           = 0x10,
    CU_AD_FORMAT_FLOAT          = 0x20
} CUarray_format;
```

ここで、CUDA 行列記述の例は：

- 2048 浮動の CUDA 行列のための記述です：

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 2048;
desc.Height = 1;
```

- 浮動の 64×64 CUDA 行列の記述です：

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 64;
desc.Height = 64;
```

- 浮動 16 の 64 ビット、4x16 ビットの **width**×**height** CUDA 行列用の記述です：

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.FormatFlags = CU_AD_FORMAT_HALF;
```

```
desc.NumChannels = 4;
desc.Width = width;
desc.Height = height;
```

- 16 ビット要素の **width×height**CUDA 行列用の記述です; 各々は 2 つの 8 ビット非割り当て char です:

```
CUDA_ARRAY_DESCRIPTOR arrayDesc;
desc.FormatFlags = CU_AD_FORMAT_UNSIGNED_INT8;
desc.NumChannels = 2;
desc.Width = width;
desc.Height = height;
```

E.8.9 cuArrayGetDescriptor()

```
CUresult cuArrayGetDescriptor(CUDA_ARRAY_DESCRIPTOR* arrayDesc,
                              CUarray array);
```

***arrayDesc** では、CUDA 行列 **array** を作成するのに使用された記述子を返します。それは CUDA 行列を渡されたサブルーチンに役立ちますが、検証か他の目的のための CUDA 行列パラメタを知る必要があります。

E.8.10 cuArrayDestroy()

```
CUresult cuArrayDestroy(CUarray array);
```

CUDA 行列 **array** を開放します。

E.8.11 cuMemset()

```
CUresult cuMemsetD8(CUdeviceptr dstDevPtr,
                    unsigned char value, unsigned int count);
CUresult cuMemsetD16(CUdeviceptr dstDevPtr,
                     unsigned short value, unsigned int count);
CUresult cuMemsetD32(CUdeviceptr dstDevPtr,
                     unsigned int value, unsigned int count);
```

値 **value** を指定した **count** 8, 16 または 32 ビット値のメモリ範囲を設定します。

E.8.12 cuMemset2D()

```
CUresult cuMemsetD2D8(CUdeviceptr dstDevPtr,
                      unsigned int dstPitch,
                      unsigned char value,
                      unsigned int width, unsigned int height);
CUresult cuMemsetD2D16(CUdeviceptr dstDevPtr,
                       unsigned int dstPitch,
                       unsigned short value,
                       unsigned int width, unsigned int height);
CUresult cuMemsetD2D32(CUdeviceptr dstDevPtr,
                       unsigned int dstPitch,
                       unsigned int value,
                       unsigned int width, unsigned int height);
```

値 **value** を指定した **width** 8, 16 または 32 ビット値の 2D メモリ範囲を設定します。**Height** は設定の行の数を指定し、**dstPitch** は各行の間のバイトの数を指定します（SectionE.8.3 を参照下さい）。cuMemAllocPitch()によって戻されたピッチが 1 のときに、これらの関数は最も速く稼動します。

E.8.13 cuMemcpyHtoD()

```
CUresult cuMemcpyHtoD(CUdeviceptr dstDevPtr,
                      const void *srcHostPtr,
                      unsigned int count);
CUresult cuMemcpyHtoDAsync(CUdeviceptr dstDevPtr,
                           const void *srcHostPtr,
                           unsigned int count,
                           CUstream stream);
```

ホスト・メモリからデバイス・メモリへコピーします。**dstDevPtr** 及び **srcHostPtr** は各々送信元と宛先のベース・アドレスを指定します。**Count** はコピーするバイトの数を指定します。

cuMemcpyHtoDAsync()は非同期で、非ゼロ **stream** 数を渡すことによって、ストリームにオプションで関連づけることができます。それはページ・ロック ホスト・メモリでのみ稼動し、もし、ポインタからページ可能メモリへ入力として渡されたらエラーを返します。

E.8.14 cuMemcpyDtoH()

```
CUresult cuMemcpyDtoH(void* dstHostPtr,
                      CUdeviceptr srcDevPtr,
                      unsigned int count);
CUresult cuMemcpyDtoHAsync(void* dstHostPtr,
                           CUdeviceptr srcDevPtr,
                           unsigned int count,
                           CUstream stream);
```

デバイスからホスト・メモリへコピーします。**dstHostPtr** 及び **srcDevPtr** は各々の送信元と宛先のベース・アドレスを指定します。**Count** はコピーするバイトの数を指定します。

cuMemcpyDtoHAsync()は非同期で、非ゼロ **stream** 数を渡すことによって、ストリームにオプションで関連づけることができます。それはページ・ロック ホスト・メモリでのみ稼動し、もし、ポインタからページ可能メモリへ入力として渡されたらエラーを返します。

E.8.15 cuMemcpyDtoD()

```
CUresult cuMemcpyDtoD(CUdeviceptr dstDevPtr,
                      CUdeviceptr srcDevPtr,
                      unsigned int count);
```

デバイス・メモリからデバイス メモリへコピーします。デバイスからホスト・メモリへコピーします。**dstDevice** 及び **srcDevPtr** は各々の送信元と宛先のベース・アドレスを指定します。**Count** はコピーするバイトの数を指定します。

E.8.16 cuMemcpyDtoA()

```
CUresult cuMemcpyDtoA(CUarray dstArray,
                      unsigned int dstIndex,
                      CUdeviceptr srcDevPtr,
                      unsigned int count);
```

1 次元 CUDA 行列からデバイス・メモリへコピーします。**dstArray** 及び **dstIndex** は CUDA 行列ハンドルと宛先データのインデックスの始めを指定します。**srcDevPtr** はソースのベース・ポインタを指定します。**Count** はコピーのバイトの数を指定します。

E.8.17 cuMemcpyAtoD()

```
CUresult cuMemcpyAtoD(CUdeviceptr dstDevPtr,
                      CUarray srcArray,
                      unsigned int srcIndex,
                      unsigned int count);
```

1 次元 CUDA 行列からデバイス・メモリへコピーします。**dstDevPtr** は宛先のベース・ポインタを指定し、CUDA 行列要素の自然な整列しなければなりません。**srcArray** 及び **srcIndex** は CUDA 行列ハンドルとコピーが始まる行列要素のインデックス(行列要素の)を指定します。**Count** はコピーするバイトの数を指定し、行列要素サイズにより均等に分割可能にしなければなりません。

E.8.18 cuMemcpyAtoH()

```
CUresult cuMemcpyAtoH(void* dstHostPtr,
                      CUarray srcArray,
                      unsigned int srcIndex,
                      unsigned int count);
CUresult cuMemcpyAtoHAsync(void* dstHostPtr,
                           CUarray srcArray,
                           unsigned int srcIndex,
                           unsigned int count,
                           CUstream stream);
```

1 次元 CUDA 行列からデバイス・メモリへコピーします。**dstHostPtr** は宛先のベース・ポインタを指定します。**srcArray** 及び **srcIndex** は CUDA 行列ハンドルと宛先データのインデックスの始めを指定します。**count** はコピーのバイトの数を指定します。

cuMemcpyAtoHAsync()は非同期で、非ゼロ **stream** 引数を渡すことでストリームとオプションで関連付けできます。それはページ・ロック・ホスト・メモリでのみ稼動し、もしポインタがページ可能メモリへ入力として渡されたらエラーを返します。

E.8.19 cuMemcpyHtoA()

```
CUresult cuMemcpyHtoA(CUarray dstArray,
                      unsigned int dstIndex,
                      const void *srcHostPtr,
                      unsigned int count);
CUresult cuMemcpyHtoAAsync(CUarray dstArray,
```

```
                         unsigned int dstIndex,
                         const void *srcHostPtr,
                         unsigned int count,
                         CUstream stream);
```

1 次元 CUDA 行列からデバイス・メモリへコピーします。**srcHostPtr** は宛先のベース・ポインタを指定します。**dstArray** 及び **sdstIndex** は CUDA 行列ハンドルと宛先データのインデックスの始めを指定します。**count** はコピーのバイトの数を指定します。

cuMemcpyHtoAAsync()は非同期で、非ゼロ **stream** 引数を渡すことでストリームとオプションで関連付けできます。それはページ・ロック・ホスト・メモリでのみ稼動し、もしポインタがページ可能メモリへ入力として渡されたらエラーを返します。

E.8.20 cuMemcpyAtoA()

```
CUresult cuMemcpyAtoA(CUarray dstArray,
                     unsigned int dstIndex,
                     CUarray srcArray,
                     unsigned int srcIndex,
                     unsigned int count);
```

1つの 1 次元 CUDA 行列から他へコピーします。**dstArray** 及び **srcIndex** は CUDA 行列ハンドルと宛先データのインデックスの始めを指定します。結果として、**dstIndex** 及び **srcIndex** は CUDA 行列の送信元と宛先データのインデックスを指定します。それらの値は CUDA 行列の**[0, Width-1]**範囲内です；それらはバイト・オフセットではありません。**Count** はコピーされるバイトの数です。CUDA 行列の要素のサイズは同じフォーマットである必要はありませんが、要素は同じサイズでなければなりません；そしてカウントはサイズにより均等に分割されなければなりません。

E.8.21 cuMemcpy2D()

```
CUresult cuMemcpy2D(const CUDA_MEMCPY2D* copyParam);
CUresult cuMemcpy2DUnaligned(const CUDA_MEMCPY2D* copyParam);
CUresult cuMemcpy2DAsync(const CUDA_MEMCPY2D* copyParam,
                        CUstream stream);
```

copyParam で指定されたパラメタにより 2D メモリ・コピーを実行します。**CUDA_MEMCPY2D** 構造は次のように定義されます：

```
typedef struct CUDA_MEMCPY2D_st {

   unsigned int srcXInBytes, srcY;
   CUmemorytype srcMemoryType;
      const void *srcHost;
      CUdeviceptr srcDevice;
      CUarray srcArray;
      unsigned int srcPitch;

   unsigned int dstXInBytes, dstY;
   CUmemorytype dstMemoryType;
      void *dstHost;
      CUdeviceptr dstDevice;
      CUarray dstArray;
      unsigned int dstPitch;
```

```
    unsigned int WidthInBytes;
    unsigned int Height;
} CUDA_MEMCPY2D;
```

ここで:

- **srcMemoryType** 及び **dstMemoryType** は送信元及び宛先のメモリの型を指定します。結果的に **Cumemorytype_enum** は次のように定義されます；

```
typedef enum CUmemorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03
} CUmemorytype;
```

もし **srcMemoryType** が **CU_MEMORYTYPE_HOST** なら、**srcHost** 及び **srcPitch** はソース・データの（ホスト）ベース・アドレスと適応するための行あたりのバイトを指定します。**srcArray** は無視されます。

もし **srcMemoryType** が **CU_MEMORYTYPE_DEVICE** なら、**srcDevice** 及び **srcPitch** はソース・データの（デバイス）ベース・アドレスと適応するための行あたりのバイトを指定します。**srcArray** は無視されます。

もし **srcMemoryType** が **CU_MEMORYTYPE_ARRAY** なら、**srcArray** 及び **srcPitch** はソース・データのハンドルを指定します。**srcHost**, **srcDevice** 及び **srcPitch** は無視されます。

もし **srcMemoryType** が **CU_MEMORYTYPE_HOST** なら、**dstHost** 及び **dstPitch** はソース・データの（ホスト）ベース・アドレスと適応するための行あたりのバイトを指定します。**dstArray** は無視されます。

もし **dstMemoryType** は **CU_MEMORYTYPE_DEVICE** なら、**dstDevice** 及び **dstPitch** はソース・データの（ホスト）ベース・アドレスと適応するための行あたりのバイトを指定します。**dstArray** は無視されます。

もし **dstMemoryType** が **CU_MEMORYTYPE_ARRAY** なら、**dstArray** は宛先のハンドルを指定します。**dstHost, dstDevice** および **dstPitch** は無視されます。

- **srcXInBytes** 及び **srcY** はコピーのための送信元データのベース・アドレスを指定します。

ホスト・ポインターのための開始アドレスは

```
void* Start =
        (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

デバイス ポインターのための開始アドレスは

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

CUDA 行列のために **srcXInBytes** は行列要素サイズにより均等に分割されなければなりません。

- **dstXInBytes** 及び **dstY** はコピーのための送信元データのベース・アドレスを指定します。

ホスト・ポインタのためのベース・アドレスは

```
void* dstStart =
        (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

デバイス ポインターのための開始アドレスは

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

CUDA 行列のために、**dstXInBytes** は行列要素サイズにより均等に分割されなければなりません。

- **WidthInBytes** 及び **Height** は2D コピーを実行する幅(バイト内)と高さを指定します。あらゆるピッチは **WidthInBytes** 以上でなければなりません。

cuMemcpy2D()はあらゆるピッチが最大許容以上なら、エラーを返します(SectionE.2.6 の **CU_DEVICE_ATTRIBUTE_MAX_PITCH** を参照下さい)。

cuMemAllocPitch()はいつも **cuMemcpy2D().**と稼動するピッチを戻します。

イントラ-デバイス・メモリ上でコピーします(デバイス ↔ デバイス, CUDA 行列 ↔ デバイス, CUDA 行列 ↔ CUDA 行列)。**cuMemcpy2D()**は **cuMemAllocPitch().** により計算しないピッチのために失敗するでしょう。**cuMemcpy2DUnaligned()**はこの制限がありませんが、かなり遅く **cuMemcpy2D()**がエラーコードを返したケースで稼動するかもしれません。

cuMemcpy2DAsync()は非同期で非ゼロ **stream** 引数を渡すことでストリームとオプションで関連付けできます。それはページ・ロック・ホスト・メモリでのみ稼動し、もしポインタがページ可能メモリへ入力として渡されたらエラーを返します。

E.9 テクスチャ参照管理

E.9.1 cuTexRefCreate()

```
CUresult cuTexRefCreate(CUtexref* texRef);
```

テクスチャ参照を生成し***texRef** のそのハンドルを返します。一度生成されたら、アプリケーションは、参照を割り当てメモリに関連づけるために **cuTexRefSetArray()**か **cuTexRefSetAddress()** をコールしなければなりません。他のテクスチャ参照関数は、フォーマットを指定するのに使用され、メモリがこのテクスチャ参照を経由して読出すときに(アドレッシング、フィルタリングなどを)変換処理するのに使用されます。関数与えるためにテクスチャ序数のテクスチャ参照と関連付けるために、アプリケーションは **cuParamSetTexRef().**をコールするでしょう。

E.9.2 cuTexRefDestroy()

```
CUresult cuTexRefDestroy(CUtexref texRef);
```

テクスチャ参照を破棄します。

E.9.3 cuTexRefSetArray()

```
CUresult cuTexRefSetArray(CUtexref texRef,
                          CUarray array,
                          unsigned int flags);
```

テクスチャ参照 **texRef** へ CUDA 行列 **array** をバインドします。あらゆる前のアドレスまたは CUDA 行列がテクスチャ参照と関連付けられた状態はこの関数により優先します。**flags** は **CU_TRSA_OVERRIDE_FORMAT** へ設定されなければなりません。あらゆる

texRef に前もって拘束した CUDA 行列 は非拘束です。

E.9.4 cuTexRefSetAddress()

```
CUresult cuTexRefSetAddress(unsigned int* byteOffset,
                            CUtexref texRef,
                            CUdeviceptr devPtr,
                            int bytes);
```

テクスチャ参照 **texRef** にリニア・アドレス範囲をバインドします。テクスチャ参照に関連しているあらゆる前のアドレスや CUDA 行列状態もこの関数によって取って代わります。前もって **texRef** に拘束していたあらゆるメモリは非拘束です。ハードウェアがテクスチャ・ベース・アドレスの整列要求を強制してから、**cuTexRefSetAddress()**は必要なメモリから読出すために、テクスチャ・フェッチに適用しなければならない***byteOffset** でオフセットを 1 バイト戻します。それらを **tex1Dfetch()**関数に適用することができるように、このオフセットをテクセル・サイズで分割し、カーネル・テクスチャから読出したものに渡さなければなりません。もし、デバイス・メモリ・ポインタが **cuMemAlloc()**から返ったら、オフセットは 0 になるように保証されます、そして、NULL は **ByteOffset** パラメタとして渡されるでしょう。

E.9.5 cuTexRefSetFormat()

```
CUresult cuTexRefSetFormat(CUtexref texRef,
                           CUarray_format format,
                           int numPackedComponents);
```

テクスチャ参照 **texRef** によって読むようにデータのフォーマットを指定します。**format** と **numPackedComponents** は **CUDA_ARRAY_DESCRIPTOR** 構造の **Format** と **NumChannels** メンバーによく類似しています；それらは各々のコンポーネントのフォーマットと行列要素あたりのコンポーネントの数を指定します。

E.9.6 cuTexRefSetAddressMode()

```
CUresult cuTexRefSetAddressMode(CUtexref texRef,
                                int dim, CUaddress_mode mode);
```

テクスチャ参照 texRef の与えられた次数にアドレッシング・モード **mode** を指定します。もし **dim** がゼロなら、アドレッシング・モードはテクスチャからフェッチに使われる関数の最初のパラメータへ適用されます（Section4.4.5 を参照下さい）。もし、**dim** が1なら2番目以降のパラメータです。**CUaddress_mode** は下記にように定義されます；

```
typedef enum CUaddress_mode_enum {
    CU_TR_ADDRESS_MODE_WRAP = 0,
    CU_TR_ADDRESS_MODE_CLAMP = 1,
    CU_TR_ADDRESS_MODE_MIRROR = 2,
} CUaddress_mode;
```

このコールが **texRef** がリニア・メモリへの拘束であるなら効き目がないことを留意下さい。

E.9.7 cuTexRefSetFilterMode()

```
CUresult cuTexRefSetFilterMode(CUtexref texRef,
```

```
                         CUfilter_mode mode);
```

テクスチャ参照 **texRef** を経由してメモリを読出しているとき、フィルタリング・モード **mode** が使用されるように指定します。**CUfilter_mode_enum** は次のように定義されます：

```
typedef enum CUfilter_mode_enum {
    CU_TR_FILTER_MODE_POINT = 0,
    CU_TR_FILTER_MODE_LINEAR = 1
} CUfilter_mode;
```

このコールが **texRef** がリニア・メモリへの拘束であるなら効き目がないことを留意下さい。

E.9.8 cuTexRefSetFlags()

```
CUresult cuTexRefSetFlags(CUtexref texRef, unsigned int Flags);
```

テクスチャ参照を経由して返されたデータの振舞いを制御するためにオプションでフラッグを指定します。有効なフラッグは：

- **CU_TRSF_READ_AS_INTEGER** はテクスチャ・プロモート整数データから[0, 1]の範囲で浮動小数点ポイント・データを持っているデフォルトの振舞いを抑制します；
- **CU_TRSF_NORMALIZED_COORDINATES** は座標範囲[0, Dim)のテクスチャを持っているデフォルトの振舞いを抑制します。ここで Dim は CUDA 行列の幅と高さです。代わりに、テクスチャ座標[0, 1.0)は行列次数の全体の幅を参照します。

E.9.9 cuTexRefGetAddress()

```
CUresult cuTexRefGetAddress(CUdeviceptr* devPtr, CUtexref texRef);
```

***devPtr** のテクスチャ参照 **texRef** へベース・アドレス拘束に返すか、もしテクスチャ参照があらゆるデバイス・メモリ範囲へ拘束しないなら、**CUDA_ERROR_INVALID_VALUE** を返します。

E.9.10 cuTexRefGetArray()

```
CUresult cuTexRefGetArray(CUarray* array, CUtexref texRef);
```

***array** のテクスチャ参照 **texRef** へベース・アドレス拘束に返すか、もしテクスチャ参照があらゆる CUDA 行列へ拘束しないなら、**CUDA_ERROR_INVALID_VALUE** を返します。

E.9.11 cuTexRefGetAddressMode()

```
CUresult cuTexRefGetAddressMode(CUaddress_mode* mode,
                                CUtexref texRef,
                                int dim);
```

テクスチャ参照の次数 **dim** へ対応しているアドレッシング・モード ***mode** に返します。現在は **dim** が 0 か 1 の値のみ有効です。

E.9.12 cuTexRefGetFilterMode()

```
CUresult cuTexRefGetFilterMode(CUfilter_mode* mode,
                               CUtexref texRef);
```

returns in ***mode** the filtering mode of the テクスチャ参照 **texRef**.

E.9.13 cuTexRefGetFormat()

```
CUresult cuTexRefGetFormat(CUarray_format* format,
                           int* numPackedComponents,
                           CUtexref texRef);
```

テクスチャ参照 **texRef** への CUDA 行列拘束のコンポーネント数、フォーマット ***numPackedComponents** 及び***format** に返します。もし、**format** か **numPackedComponents** がヌルなら、それは無視されます。

E.9.14 cuTexRefGetFlags()

```
CUresult cuTexRefGetFlags(unsigned int* flags, CUtexref texRef);
```

テクスチャ参照 **texRef** のフラッグ***flags** に返します。

E.10 OpenGL 相互運用性

E.10.1 cuGLInit()

```
CUresult cuGLInit(void);
```

OpenGL 相互運用を初期化します。これは他のあらゆる OpenGL 相互運用関数を実行する前にコールされなければなりません。もし必要とした OpenGL ドライバが不可能なら失敗するでしょう。

E.10.2 cuGLRegisterBufferObject()

```
CUresult cuGLRegisterBufferObject(GLuint bufferObj);
```

CUDA によるアクセスのための ID **bufferObj** のバッファ・オブジェクトを登録します。この関数は CUDA がバッファ・オブジェクトにマップできる前にコールしなければなりません。これを登録している間に、バッファ・オブジェクトは OpenGL ドローイング・コマンドのためのデータ・ソースを除き、あらゆる OpenGL コマンドにより使うことはできません。

E.10.3 cuGLMapBufferObject()

```
CUresult cuGLMapBufferObject(CUdeviceptr* devPtr,
                             unsigned int* size,
                             GLuint bufferObj);
```

ID **bufferObj** バッファ・オブジェクから現在の CUDA コンテクストのアドレス空間へマップし、***devPtr** と***size** で結果として起こるマッピングのベース・ポインタとサイズを返します。

E.10.4 cuGLUnmapBufferObject()

```
CUresult cuGLUnmapBufferObject(GLuint bufferObj);
```

CUDA によるアクセスのための ID **bufferObj** のバッファ・オブジェクトを非マップします。

E.10.5 cuGLUnregisterBufferObject()

```
CUresult cuGLUnregisterBufferObject(GLuint bufferObj);
```

CUDA によるアクセスのための ID **bufferObj** のバッファ・オブジェクトを非登録します。

E.11 Direct3D 相互運用性

E.11.1 cuD3D9Begin()

```
CUresult cuD3D9Begin(IDirect3DDevice9* device);
```

Direct3D デバイス **device** の相互運用性を初期化します。この関数は **device** からのあらゆるオブジェクトを CUDA がマップできる前にコールしなければなりません。アプリケーションは **cuD3D9End()**がコールされるまで Direct3D デバイスによる頂点バッファが所有していたのをマップできます。

E.11.2 cuD3D9End()

```
CUresult cuD3D9End(void);
```

Direct3D デバイスが以前 **cuD3D9Begin()**で指定されている状態で相互運用性を結論づけます。

E.11.3 cuD3D9RegisterVertexBuffer()

```
CUresult cuD3D9RegisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

CUDA によるアクセスのために Direct3D 頂点バッファ **VB** を登録します。

E.11.4 cuD3D9MapVertexBuffer()

```
CUresult cuD3D9MapVertexBuffer(CUdeviceptr* devPtr,
                               unsigned int* size,
                               IDirect3DVertexBuffer9* VB);
```

Direct3D 頂点バッファ **VB** を現在の CUDA コンテクストのアドレス空間にマップし、***devPtr** 及び ***size** の結果として起こるマッピングのベースポインターとサイズを返します。

E.11.5 cuD3D9UnmapVertexBuffer()

```
CUresult cuD3D9UnmapVertexBuffer(IDirect3DVertexBuffer9* VB);
```

CUDA によるアクセスのための頂点バッファ VB を非マップします。

E.11.6 cuD3D9UnregisterVertexBuffer()

```
CUresult cuD3D9UnregisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

CUDA によるアクセスのための頂点バッファ VB を非登録します。

E.11.7 cuD3D9GetDevice()

```
CUresult cuD3D9GetDevice(CUdevice* dev, const char* adapterName);
```

*dev で EnumDisplayDevices か IDirect3D9::GetAdapterIdentifier()より得られたアダプター名 adapterName に対応するデバイスを返します。

Appendix F. テクスチャ・フェッチ

この Appendix ではテクスチャ参照（Section4.3.4 を参照下さい）の前の属性に依存する、Section4.4.5 でテクスチャ関数によって返された値を計算するのに使用される方法を述べます。

テクスチャ参照へのテクスチャ拘束は 1-次数テクスチャのためのテクセルか 2-次数テクスチャのためのテクセルの行列として表されます。それはテクスチャ座標 x 及び y を使ってフェッチされます。T を記述するのに、それを使用できる前にテクスチャ座標は T の有効なアドレス範囲の中に落ちていなければなりません。アドレッシング・モードはテクスチャ座標 x の範囲外がどのように有効な範囲に再マップされるかを指定します。

もし x が非正規化なら、クランプ・アドレッシング・モードだけをサポートし、もし $x < 0$ なら x を 0 に、もし $N \leq x$ なら $N - 1$ に取り替えます。もし、x が正規化なら：

- クランプ・アドレス・モード内で x が $x < 0$ なら 0 に、$1 \leq x$ なら $1 - \frac{1}{N}$ に取り替えます、
- ラップ・アドレス・モード内でが x が $frac(x)$ に取り替えます。ここで、$frac(x) = x - floor(x)$ で、$floor(x)$ は最大整数で x 以上ではありません。

残りの Appendix で、x 及び y は T の有効なアドレッシング範囲pに再マップされる非正規化テクスチャ座標です。x 及び y は $x = N\hat{x}$ 及び $y = M\hat{y}$.のように正規化テクスチャ座標 $\hat{x}$ 及び $\hat{y}$ から配信されます

F.1 直近ポイントのサンプリング

このフィルタリング・モードで、テクスチャ・フェッチにより返される値は

- 1-次数テクスチャのために $tex(x) = T[i]$
- 2-次数テクスチャのために $tex(x, y) = T[i, j]$
- ここで $i = floor(x)$ 及び $j = floor(y)$ です。

Figure F-1 では $N = 4$ の1-次数テクスチャのための直近ポイントのサンプリンを示しています。整数テクスチャのために、テクスチャ・フェッチにより返された値は、[0.0, 1.0]へオプションで再マップできます（Section4.3.4.1 を参照下さい）。

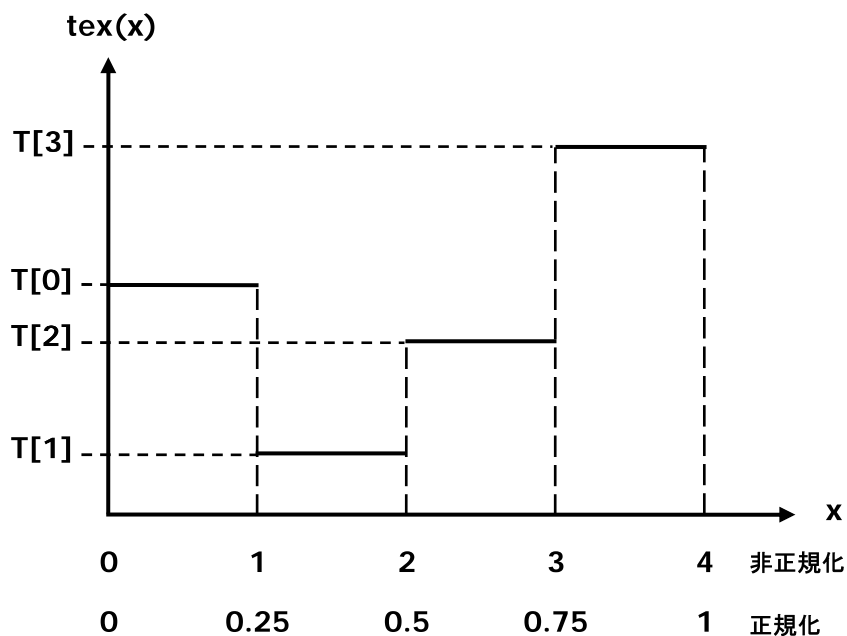


Figure F-1. 4テクセルの1-次数テクスチャの直近サンプリング

F.2 リニア・フィルタリング

このフルタリング・モードで、浮動小数点テクスチャのためにのみ可能で、テクスチャ・フェッチにより返される値は

- 1-次数テクスチャのために $tex(x) = (1-\alpha)T[i] + \alpha T[i+1]$
- 2-次数テクスチャのために
  $tex(x, y) = (1-\alpha)(1-\beta)T[i, j] + \alpha(1-\beta)T[i+1, j] + (1-\alpha)\beta T[i, j+1] + \alpha\beta T[i+1, j+1]$
- ここで:
- $i = floor(x_B)$, $\alpha = frac(x_B)$, $x_B = x - 0.5$,
- $j = floor(y_B)$, $\beta = frac(y_B)$, $y_B = y - 0.5$.

α 及び β は関数値の 8 ビット付 9 ビット固定ポイント・フォーマットにストアされます。

Figure F-2 では $N = 4$ の 1-次数テクスチャのための直近ポイント・サンプリングを表しています。

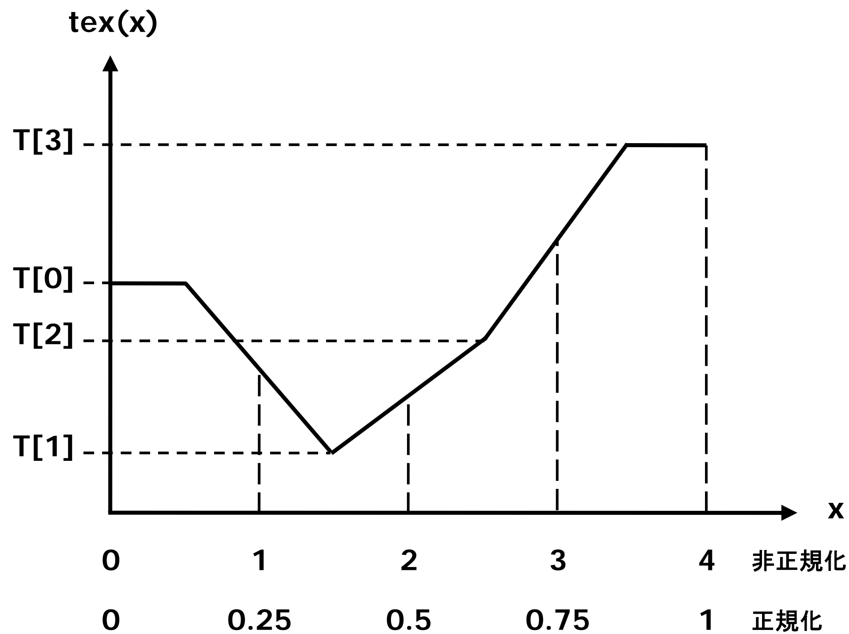


Figure F-2. クランプ・アドレッシング・モードで4テクセルの 1-次数テクスチャのリニア・フィルタリング

F.3 参照テーブル

参照テーブル $TL(x)$ ここで間隔 $[0, R]$ の x スパンは、$TL(0) = T[0]$ 及び $TL(R) = T[N-1]$ の順序を確実にするために $TL(x) = tex(\frac{N-1}{R}x + 0.5)$ として実装できます。

Figure F-3 はテクスチャ・フィルタリングから $N = 4$ の 1-次数テクスチャから $R = 4$ または $R = 1$ の参照テーブルへの実装の使用を示します。

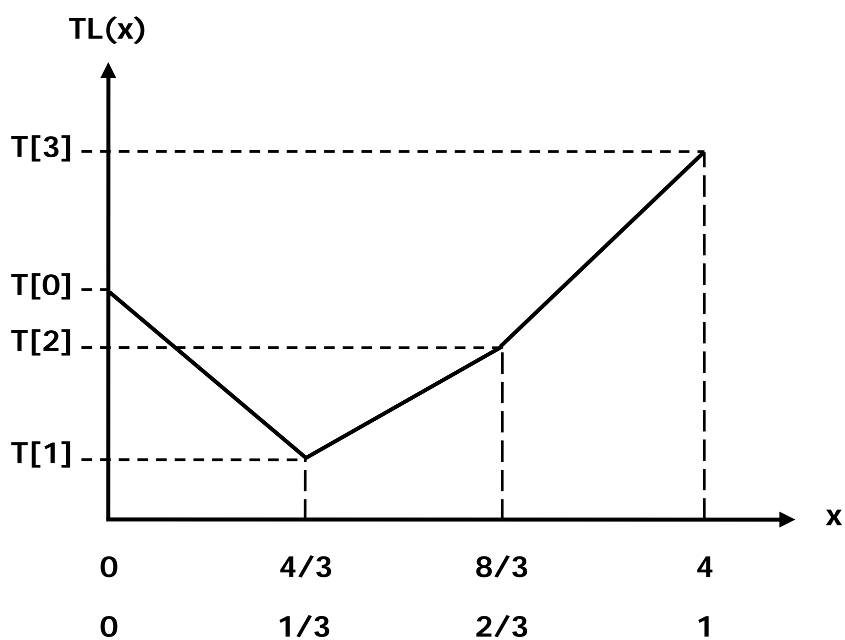


Figure F-3. リニア・フィルタリングを使っている 1-次数の参照テーブル

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support デバイス s or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, GeForce, Tesla, and Quadro are trademarks or registered trademarks of NVIDIA Corporation. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2007 NVIDIA Corporation. All rights reserved.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com