



ホワイトペーパー

NVIDIA の次世代 CUDA™ コンピュータアーキテクチャ:

Fermi™

目次

GPU コンピューティングの歴史

G80 アーキテクチャ

計算処理とグラフィックスをカバーする NVIDIA の次世代アーキテクチャ、
CUDA（開発コード：「Fermi」）

CUDA の概要

ハードウェア実行

Fermi アーキテクチャの概要

第 3 世代のストリーミング・マルチプロセッサ

512 個の高性能 CUDA コア

16 個のロード/ストア・ユニット

4 個の特殊関数ユニット

倍精度重視の設計

デュアルワープスケジューラ

構成可能な 64KB の共有メモリと L1 キャッシュ

比較表

第 2 世代の並列スレッド実行 ISA

統合アドレス空間によって C++ のフルサポートを実現

OpenCL と DirectCompute に最適化

IEEE 32 ビット浮動小数点精度

述部処理で条件処理パフォーマンスを向上

メモリサブシステムのイノベーション

構成可能な L1 キャッシュと統合 L2 キャッシュを持つ NVIDIA Parallel
DataCache™

GPU として初めて ECC メモリをサポート

高速なアトミックメモリ操作

GigaThread™ スレッドスケジューラ

アプリケーションコンテキストの切り替え速度が 10 倍に向上

コンカレントカーネル実行

NVIDIA Nexus について

まとめ

GPU コンピューティングの歴史

グラフィックス・プロセッシング・ユニット (GPU) は 1999 年に NVIDIA が世界へ送り出したもので、現在、並列プロセッサとして最も広く普及しています。迫真のリアルタイム・グラフィックスを強く求める声を背景に、GPU は、抜きんでた浮動小数点演算パフォーマンスとプログラミング性を持つプロセッサへと進化しました。数値演算のスループットやメモリ帯域幅では CPU を凌駕する性能を持ち、さまざまなデータ並列アプリケーションの高速化に最適なプロセッサとなったのです。

2003 年には、この GPU をグラフィックス以外の用途に活用しようという試みが始まりました。さまざまなデータ並列アルゴリズムが、DirectX、OpenGL、Cg といった高水準のシェーディング言語を使って GPU へとポーティングされました。タンパク質フォールディング、ストックオプションの価格決定、SQL クエリ、MRI 画像再構成などの処理が、GPU で大幅に高速化されたのです。このころはグラフィックス API を通じて汎用処理を行っており、GPGPU プログラムと呼ばれていました。

GPGPU は大幅なスピードアップを実現しましたが、同時にさまざまな問題も浮上しました。第一に、グラフィックス API と GPU アーキテクチャについてプログラマが詳しく知っている必要がありました。第二に、バーテックス座標、テクスチャ、シェーダー・プログラムという形で問題を記述する必要があり、プログラミングが複雑で難しいものとなっていました。第三に、アドレス指定が可能なリードやライトなどの基本的なプログラミング機能も用意されておらず、プログラミング方法が大きく制限されていました。第四に、倍精度演算が(最近まで)サポートされていなかったため、GPU による処理が不可能な科学分野がありました。

このような問題に対応するため、NVIDIA では、二つのテクノロジーを導入しました。一つがグラフィックスと計算処理を統合化した G80 アーキテクチャです(初搭載は GeForce 8800@、Quadro FX 5600@、Tesla C870@でした)。もう一つがソフトウェアとハードウェアをカバーする CUDA アーキテクチャです。こちらは GPU のプログラミングをさまざまな高級言語から行えるようにするものです。これら二つの新技術により、GPU を新しい形で活用できるようになりました。グラフィックス API を通じてグラフィックス専用デバイスをプログラミングするのではなく、汎用超並列プロセッサをターゲットに CUDA エクステンションを持つ C 言語でプログラムを書けるようになったのです。NVIDIA では GPU を活用するこの新しいプログラミングを「GPU コンピューティング」と呼ぶこと

にしましたー初期に行われた「GPGPU」型プログラミングとは大きく異なり、サポートできるアプリケーションも使用できるプログラミング言語も大幅に拡大したのです。

G80 アーキテクチャ

新しい GPU コンピューティング・モデルの先駆けとなった製品は、NVIDIA GeForce 8800 でした。2006 年 11 月に発売された G80 アーキテクチャ搭載の GeForce 8800 には、GPU コンピューティングを可能にするさまざまなイノベーションが採用されていました。

- G80 は GPU として初めて C 言語をサポートしました。この結果、新しいプログラミング言語を学ぶことなく GPU の能力を活用できるようになりました。
- G80 は、それまで分離されていたバーテックス・パイプラインとピクセル・パイプラインを統合し、バーテックスからジオメトリ、ピクセル、計算までの処理を一つのプロセッサで行えるようにした初めての GPU でした。
- G80 は GPU として初めてスカラー・スレッド・プロセッサをサポートしました。この結果、ベクタ・レジスタをプログラマがマニュアル管理する必要がなくなりました。
- G80 には実行モデルとして SIMT (Single-Instruction Multiple-Thread) が導入され、一つの命令で多数の独立スレッドを同時並行に実行可能となりました。
- スレッド間通信に使用する共有メモリとバリア同期の機能も導入されました。

G80 アーキテクチャは、2008 年 6 月にメジャーアップグレードを行いました。第 2 世代の統合アーキテクチャ、GT200 (初搭載は GeForce GTX 280、Quadro FX 5800、Tesla T10 GPU) では、ストリーミング・プロセッサ・コア (のちに CUDA コアへと改称) の数が 128 個から 240 個へと拡大されました。プロセッサのレジスタファイルもサイズが倍増され、同時にオンチップ実行可能なスレッド数が大幅に増加しました。ハードウェアメモリアクセスの結合機能も用意され、メモリアクセスの効率が改善されました。倍精度浮動小数点もサポートされ、ハイパフォーマンスコンピューティング (HPC) や科学計算のニーズに対応できるようになりました。

新しい世代の GPU を開発するにあたり、NVIDIA では、既存アプリケーションのパフォーマンスを高めるとともに GPU のプログラミング性を拡大することを方針としてきました。すぐに効果が感じられるのはアプリケーションの高速化のほうですが、GPU が最も汎用性の高い並列プロセッサへと進化できたのは GPU のプログラミング性が進歩してきたからです。この方針は、GT200 の後継アーキテクチャを開発する際にも堅持しました。

計算処理とグラフィックスをカバーする NVIDIA の次世代アーキテクチャ、CUDA（開発コード：「Fermi」）

G80 の登場以来、最大の進化となるのが Fermi アーキテクチャです。グラフィックスと計算処理を統合した並列プロセッサとはどうあるべきかという方向性を示したのが G80 でした。その G80 のパフォーマンスと機能を拡大したのが GT200 です。これら 2 世代のプロセッサとその関連アプリケーションから得た知見をすべて投入し、まったく新しいアプローチで設計をやり直して世界初のコンピュテーショナル GPU に仕上げたもの、それが Fermi です。Fermi の開発では、まず、G80 や GT200 を使った GPU コンピューティングに対するユーザのフィードバックを集め、以下の点を改善の重要項目としました。

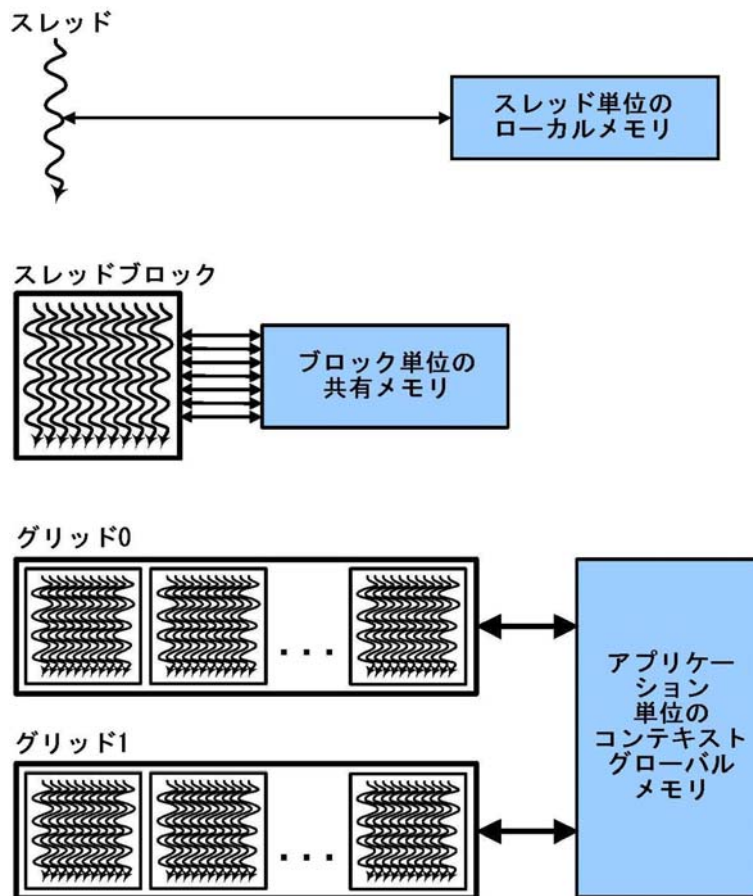
- 単精度浮動小数点の演算はパフォーマンスがデスクトップ CPU の 10 倍にも達していましたが、GPU コンピューティング・アプリケーションの一部においては、倍精度のパフォーマンスをもっと高めて欲しいとの要望がありました。
- データセンターで多数の GPU を使用して GPU コンピューティングが行えるように、ECC メモリに対応する必要がありました。
- GPU 共有メモリを利用できない並列アルゴリズムがあり、きちんとしたキャッシュアーキテクチャが欲しいとの要望がユーザからありました。
- アプリケーションを高速化するため 16KB を超える SM 共有メモリが欲しいとの要望が多く、多くの CUDA プログラマからありました。
- アプリケーションプログラム間でコンテキストを切り替える速度を高めて欲しい、また、グラフィックスと計算処理の相互協調処理の速度を高めて欲しいとの要望がユーザからありました。
- 並列アルゴリズムで使用する、読み込み-変更-書き込みのアトミック処理を高速化して欲しいとの要望がユーザからありました。

このような要望をうけ、Fermi チームでは、計算処理能力を大幅に高めたプロセッサを設計しました。また、アーキテクチャも革新し、プログラミング性と処理効率も大幅に高めることに成功しました。Fermi アーキテクチャには、以下の特長があります。

- **第3世代のストリーミング・マルチプロセッサ(SM)**
 - SM あたり 32 個の CUDA コアと、GT200 の 4 倍を搭載
 - 倍精度浮動小数点の演算能力がピークで GT200 の 8 倍
 - 32 スレッドのワープを 1 クロックあたり 2 つ、スケジューリングしてディスパッチできるデュアルワープスケジューラ
 - 共有メモリと L1 キャッシュのパーティションを自由に構成可能な 64KB の RAM
- **第2世代の PTX(Parallel Thread Execution) ISA**
 - C++フルサポートの統合アドレス空間
 - OpenCL と DirectCompute に最適化
 - IEEE 754-2008 に完全準拠の 32 ビット精度と 64 ビット精度を実現
 - フル 32 ビット整数演算データパス(64 ビット拡張あり)
 - 64 ビットアドレッシングへの移行をサポートするメモリアクセス命令
 - プレディケーション条件実行のパフォーマンスの改善
- **改良されたメモリサブシステム**
 - 構成可変な L1 キャッシュと統合 L2 キャッシュを持つ NVIDIA Parallel DataCache™
 - GPU として初めて ECC メモリをサポート
 - アトミックなメモリ操作のパフォーマンスを大幅に改善
- **NVIDIA GigaThread™エンジン**
 - アプリケーションコンテキストの切り替え速度が 10 倍に向上
 - コンカレントカーネル実行
 - アウトオブオーダーのスレッドブロック実行
 - デュアルオーバーラップメモリ転送エンジン

CUDA の概要

CUDA はハードウェアとソフトウェアをカバーするアーキテクチャで、C、C++、Fortran、OpenCL、DirectCompute などさまざまな言語で書かれたプログラムを NVIDIA GPU で実行することができます。CUDA プログラムは並列カーネルを呼び出します。各カーネルは、複数の並列スレッドによる並列実行となります。スレッドをまとめたものをスレッドブロック、スレッドブロックをまとめたものをグリッドと呼び、プログラムやコンパイラではこれらを単位として取り扱います。実行時は、並列スレッドブロックで構成されたグリッド上に、GPU がカーネルプログラムのインスタンスを生成します。スレッドブロックを構成するスレッド1本、1本がそれぞれカーネルのインスタンス一つを実行するのです。各スレッドは自分が属するスレッドブロックにおけるスレッド ID を持つほか、プログラムカウンター、レジスタ、スレッド単位のローカルメモリ、入力、出力結果を持ちます。



CUDAは、スレッド、ブロック、グリッドという階層構造になっており、メモリ空間もスレッド単位のローカルメモリ、ブロック単位の共有メモリ、アプリケーション単位のグローバルメモリに分かれています。

同時並行で処理を行うスレッドのセットがスレッドブロックです。同じスレッドブロックに属するスレッドはバリア同期と共有メモリにより協調して動作します。スレッドブロックも自分が属するグリッドにおけるブロック ID を持ちます。

スレッドブロックを行列としてまとめたものがグリッドです。グリッドは全体で一つのカーネルを実行し、グローバルメモリからの入力データの読み取り、グ

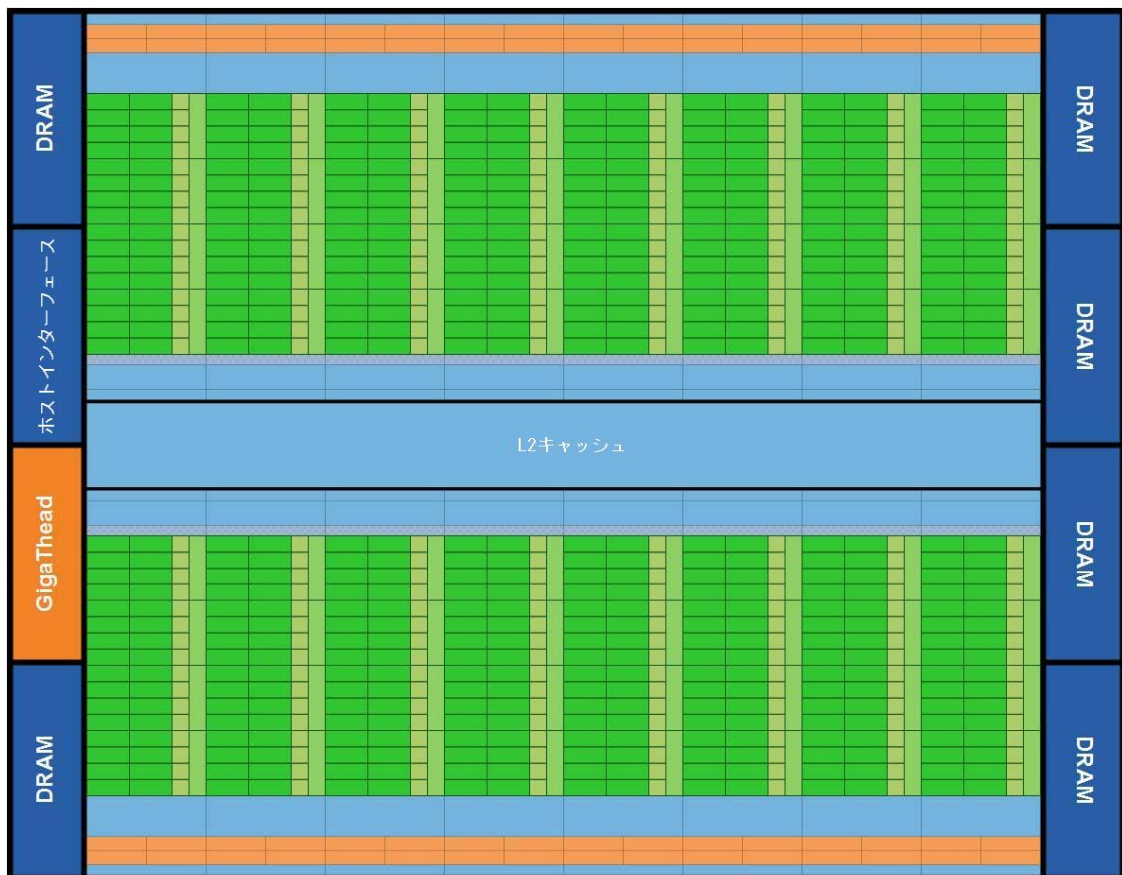
ローバルメモリへの出力データの書き出し、依存関係にあるカーネルコールの同期といった処理を行います。CUDA 並列プログラミング・モデルでは、レジスタ・スピルや関数呼び出し、C の自動配列変数などに使うローカルメモリ空間がスレッドごとに確保されます。スレッドブロックのレベルでは、並列アルゴリズムに必要なスレッド間の通信、データの共有、結果の共有に使う共有メモリ空間がブロックごとに確保されます。複数スレッドブロックで構成されるグリッドは、カーネル全体をカバーするグローバルな同期を行ったあと、グローバルメモリ空間で結果を共有します。

ハードウェア実行

CUDA ではスレッドを階層構造で取り扱いますが、GPU 側にはこれに対応するプロセッサ階層が存在します。スレッドを実行するのはストリーミング・マルチプロセッサ (SM) を構成する CUDA コアなどの実行ユニット、スレッドブロックは SM、カーネルグリッドは GPU 全体というわけです。SM では、スレッドを 32 本ごとのグループとして実行しますが、これをワープと呼びます。ワープ実行を気にせずスレッドをプログラミングするだけで、機能を実現することができます。ただし、同じワープに属するスレッドが同じコードパスをたどり、近接したアドレスのメモリにアクセスするようにプログラミングすると、パフォーマンスが大きく向上します。

Fermi アーキテクチャの概要

Fermi アーキテクチャで作られた最初の GPU は、30 億個のトランジスタで作られており、512 個の CUDA コアがあります。CUDA コア 1 個が、1 秒間に、浮動小数点命令あるいは整数演算命令 1 つを実行してスレッド 1 本の処理を進めます。512 個の CUDA コアは 32 コアずつまとめて SM を構成するため、SM の数は 16 となります。メモリパーティションは 64 ビットが 6 つあり、メモリインターフェースは 384 ビットで最大 6GB の GDDR5 DRAM メモリまでサポートできます。GPU と CPU をつなぐホストインターフェースは PCI-Express です。GigaThread グローバルスケジューラは、スレッドブロックを SM のスレッドスケジューラへと分配する機能を担当します。



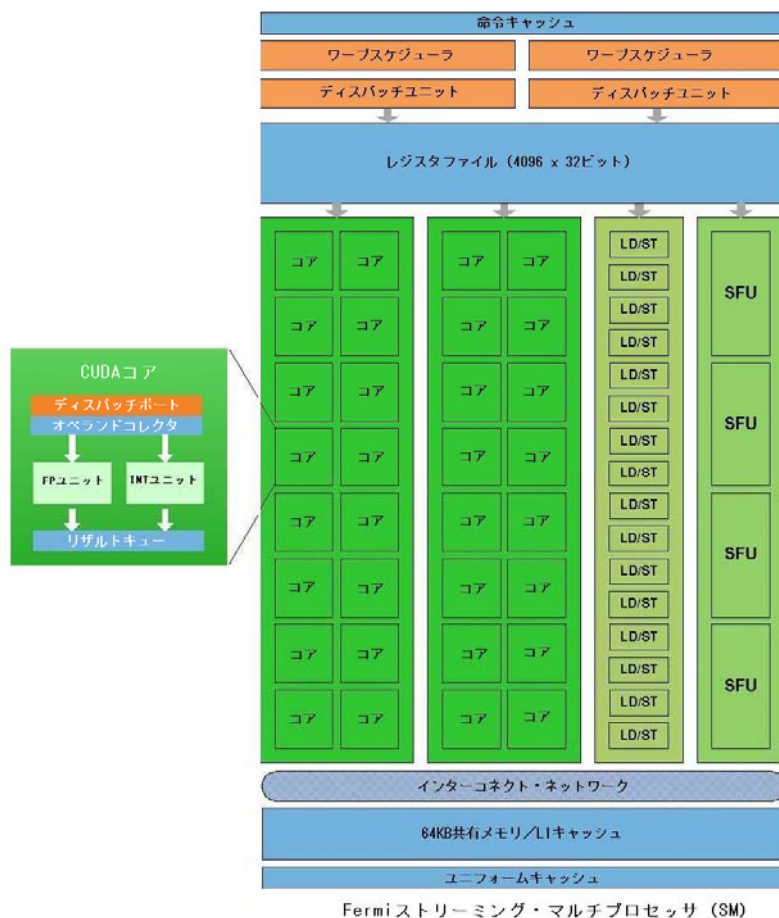
共通L2キャッシュを囲むように配置されているのが、FermiのSM、16個です。
各SMは長方形をしており、オレンジの部分がスケジューラーとディスパッチャー、
緑の部分が実行ユニット、薄い水色の部分がレジスタファイルとL1キャッシュです。

第3世代のストリーミング・マルチプロセッサ

第3世代のSMはアーキテクチャにさまざまなイノベーションが導入されており、かつてないほどパワフルになるとともにプログラミング性と効率も過去最高のレベルになっています。

512 個の高性能 CUDA コア

各 SM には、従来の 4 倍、32 個の CUDA プロセッサが用意されています。各 CUDA プロセッサには、完全パイプライン化された整数演算ユニット (ALU) と浮動小数点演算ユニット (FPU) が一つずつ用意されています。今までの GPU では、浮動小数点演算に IEEE 754-1985 を使用していましたが、Fermi アーキテクチャでは新しい IEEE 754-2008 規格を採用しているため、単精度演算で



も倍精度演算でも FMA (Fused Multiply-Add) 命令が使えます。FMA の乗算・加算では最終段階においてのみ丸め処理が行われ加算時に精度の低下がない分、従来の MAD (Multiply-Add) 命令よりも優れています。つまり、演算を別々に行うよりも FMA で処理した方が精度が高くなるのです。GT200 に搭載されていたのも倍精度 FMA でした。

GT200 では、整数演算を行う ALU の精度が、乗算時、24 ビットしかありませんでした。そのため、整数演算時に複数命令でエミュレーション・シーケンスを行う必要がありました。Fermi では整数演算用 ALU を設計しなおし、標準的なプログラミング言語で要求される 32 ビットの精度をあらゆる命令について確保しました。新設計の ALU は、64 ビットの拡張精度演算も効率的にサポートできるように最適化されています。サポートしている命令の種類も論理演算、シフト、ムーブ、コンペア、コンバート、ビットフィールド抽出、反転挿入、ビット・カウントなど多様です。

16 個のロード/ストア・ユニット

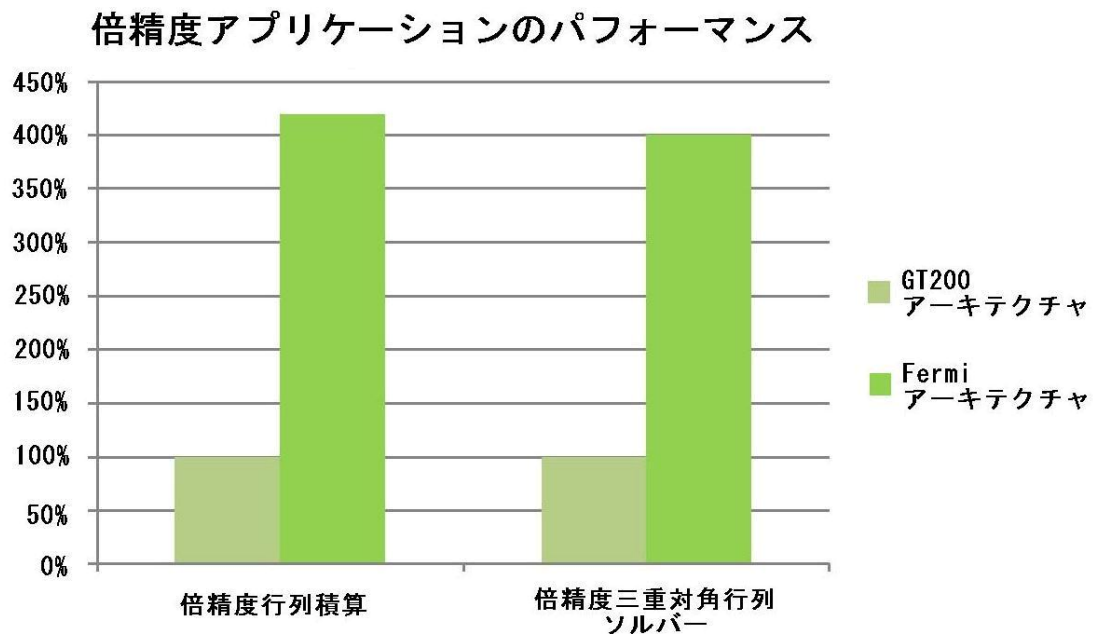
各 SM には、16 個のロード/ストア・ユニットが用意されており、1 クロックあたり 16 スレッド分のソースアドレスと宛先アドレスを計算することができます。各アドレスのデータは、サポートするユニットがキャッシュや DRAM に対してロードやストアを行います。

4 個の特殊関数ユニット

特殊関数ユニット (SFU) とは、正弦関数、余弦関数、相反関数、平方根などの超越命令を実行するユニットです。各 SFU が 1 クロックで 1 スレッドの命令 1 個を実行するため、8 クロックでワープが実行できます。SFU パイプラインはディスパッチユニットから切り離されているため、SFU が命令を実行している間にディスパッチユニットから他の演算ユニットに命令を発行することができます。

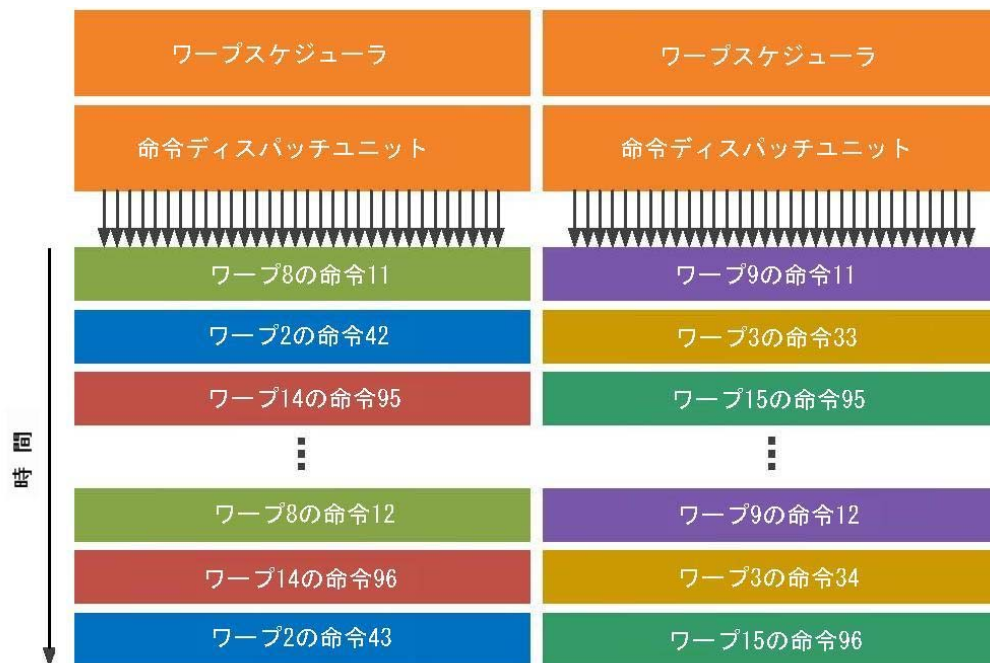
倍精度重視の設計

線形代数、数値シミュレーション、量子化学などの HPC アプリケーションでは倍精度算術演算が重要となります。Fermi アーキテクチャは、倍精度についてかつてないほど高いパフォーマンスの実現を重視した設計となっています。1 個の SM が 1 クロックで FMA 演算を 16 回も行うことが可能で、GT200 アーキテクチャと比べものにならないほどの性能を発揮します。



デュアルワープスケジューラ

SMは、32本の並列スレッドをグループ化したワープを単位にスレッドのスケジューリングを行います。SMにはワープスケジューラが2個と命令ディスパッチユニットが2個あり、2つのワープを並列に実行することができます。Fermiのデュアルワープスケジューラでは、2つのワープを選択し、1ワープあたり1つの命令を実行できるのです。ディスパッチ先となるのは、16個のコア、16個のロード/ストア・ユニット、4個のSFU、いずれかのグループとなります。各ワープは独立に実行されるため、命令ストリーム内における依存関係をスケジューラ側でチェックする必要がありません。このようにエレガントなモデルでデュアル実行を可能としているため、Fermiはピークパフォーマンスに近い性能をハードウェアから引き出せるのです。



大半の命令がデュアル実行に対応しています。整数命令 2 つ、浮動小数点命令 2 つ、あるいは、整数、浮動小数点、ロード、ストア、SFU 命令の組み合わせなどのデュアル実行が可能です。倍精度命令を他の演算と組み合わせてデュアル実行することはできません。

構成可変な 64KB の共有メモリと L1 キャッシュ

アーキテクチャ関連のイノベーションの中で、GPU アプリケーションのプログラミング性とパフォーマンスの改善に大きく貢献したものの一つがオンチップの共有メモリです。共有メモリがあれば同じスレッドブロックに属するスレッドが連携し、オンチップデータを再利用してオフチップのトラフィックを大幅に削減することができます。高性能な CUDA アプリケーションが実現した背景には、共有メモリの存在があるのです。

G80 と GT200 では、SM ごとに 16KB の共有メモリが用意されていました。Fermi アーキテクチャでは SM ごとのオンチップメモリを 64KB に拡大し、その構成を共有メモリ 48KB と L1 キャッシュ 16KB あるいは共有メモリ 16KB と L1 キャッシュ 48KB から選べるようにしました。

従来のアプリケーションも、共有メモリをよく使うもの、特にメモリアクセス帯域がネックとなる問題の場合、共有メモリを 3 倍にするとパフォーマンス

が大きく改善します。共有メモリをソフトウェアで管理するキャッシュとして使っていたアプリケーションの場合、スレッド連携用に少なくとも 16KB の共有メモリを使いつつ、ハードウェアキャッシュによってすっきりさせることが可能です。新方式最大のメリットは、共有メモリを使用しないアプリが L1 キャッシュの恩恵を享受できる点です。最小限の時間と労力で高性能な CUDA プログラムが実現できるのです。

比較表

GPU	G80	GT200	Fermi
集積トランジスタ数	6 億 8100 万個	14 億個	30 億個
CUDA コア数	128	240	512
倍精度浮動小数点 演算能力	×	30 FMA 演算/クロック	256 FMA 演算/クロック
単精度浮動小数点 演算能力	128 MAD 演算/クロック	240 MAD 演算/クロック	512 MAD 演算/クロック
ワーブスケジューラ /SM	1	1	2
特殊関数ユニット (SFU) /SM	2	2	4
共有メモリ/SM	16KB	16KB	48KB/16KB (構成可能)
L1 キャッシュ/SM	×	×	16KB/48KB (構成可能)
L2 キャッシュ/SM	×	×	768KB
ECC メモリのサポート	×	×	○
同時実行カーネル数	×	×	最大 16
ロード/ストアのアドレス 幅	32 ビット	32 ビット	64 ビット

第 2 世代の PTX (Parallel Thread Execution) ISA

Fermi アーキテクチャでは、命令セットとして並列スレッド実行 (PTX) 2.0 を初めてサポートしました。PTX とはローレベルの仮想マシンと ISA (Instruction Set Architecture) で構成されており、並列スレッド・プロセッサの操作をサポート

ートしています。PTX 命令はプログラムがインストールされる時、GPU ドライバによってマシン語の命令へと変換されます。

PTX の主な目的は以下のとおりです。

- 複数世代の GPU をカバーする一貫した ISA を提供する。
- コンパイルしたアプリケーションで GPU からフルパフォーマンスを引き出す。
- マシンに依存しない ISA を C、C++、Fortran などのコンパイラ言語で使えるようにする。
- コード配布ができる ISA をアプリケーションやミドルウェアの開発者に提供する。
- PTX をターゲットマシンにマッピングするコード生成プログラムやコード変換プログラムを最適化できる共通 ISA を提供する。
- ライブラリや高性能カーネルの手書きコーディングをしやすいようにする。
- 並列コア数が数個から多数までのさまざまな GPU をカバーできるスケーラブルなプログラミング・モデルを提供する。

PTX 2.0 には、GPU のプログラミング性、正確さ、パフォーマンスを大幅に改善するさまざまな新機能が搭載されています。たとえば、IEEE 完全準拠の 32 ビット浮動小数点演算、全ての変数とポインタに適用される統合アドレス空間、64 ビットのアドレス指定、OpenCL および DirectCompute 用の新規命令などです。特に重要なポイントは、C++ プログラミング言語をフルサポートした点です。

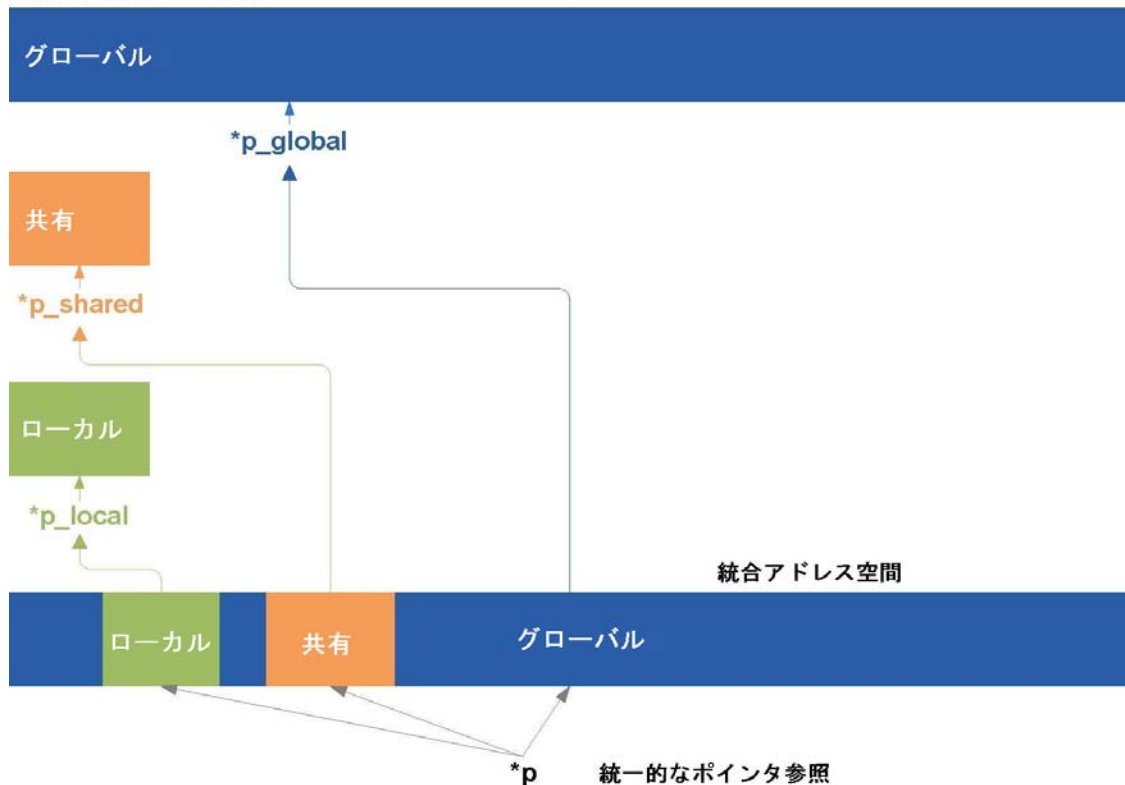
統合アドレス空間によって C++ のフルサポートを実現

アドレス空間は、従来、3 つに分かれていましたが (スレッド単位のローカルメモリ、ブロック共有メモリ、グローバルメモリ)、Fermi と PTX 2.0 ISA では 1 つの統合アドレス空間としてロード操作やストア操作が行えるようになりました。PTX 1.0 では、ロード命令やストア命令でどのアドレス空間に対するものなのかを指定していました。つまり、値を読み込む、あるいは書き込むターゲットとなるアドレス空間は、コンパイル時に確定していました。これに対し、C 言語や C++ 言語のポインタはターゲットとするアドレス空間が必ずしもコンパイル時に確定せず、ランタイムに動的に決まるケースがあるため、このポインタを完全に実装することが困難な状況となっていました。

PTX 2.0 となって 3 つのアドレス空間が連続した 1 つの統合アドレス空間になりました。ロード命令やストア命令もローカルメモリ、共有メモリ、グローバルメモリのいずれかに対するものから、同じ命令セットでアドレス空間全体

が操作できるように拡張されました。統合アドレス空間は 40 ビットで、アドレスできるメモリサイズの上限は 1 テラバイトです。また、ロード/ストアの ISA は、将来の拡張をみこして 64 ビットのアドレス指定もサポートしています。

分断されたアドレス空間



統合アドレス空間を実装した結果、Fermi で C++ プログラムを完全にサポートできるようになりました。C++ では、すべての変数と関数がオブジェクト内に置かれており、オブジェクトはポインタで渡されます。PTX 2.0 では、統合ポインタにより、どのメモリ空間に対してもオブジェクトを渡せるようになりました。ポインタの参照は、Fermi ハードウェアに搭載されたアドレス変換ユニットが適切なメモリ空間へ自動的にマッピングします。

このほか、C++ の仮想関数、関数ポインタもサポートしましたし、オブジェクトの動的な割り当てや再割り当てに用いる 'new' 演算子と 'delete' 演算子もサポートしました。C++ で例外の処理に用いる 'try' と 'catch' もサポートしました。

OpenCL と DirectCompute に最適化

OpenCL と DirectCompute は、CUDA プログラミング・モデルと密接な関係があります。スレッド、スレッドブロック、スレッドブロックのグリッド、バリア同期、ブロック単位の共有メモリ、グローバルメモリ、アトミック演算など、基本的な概念が同じなのです。第 3 世代の CUDA アーキテクチャである Fermi は、当然、これらの API に最適化されています。それだけでなく、Fermi では、フォーマット変換を伴う OpenCL や DirectCompute のサーフェイス命令をハードウェアでサポートし、グラフィックスプログラムと計算プログラムから同一データに対して簡単に処理が行えるようになっています。PTX 2.0 ISA は、ビット・カウント、アペンド、ビット反転などの DirectCompute 命令もサポートしています。

IEEE 32 ビット浮動小数点精度

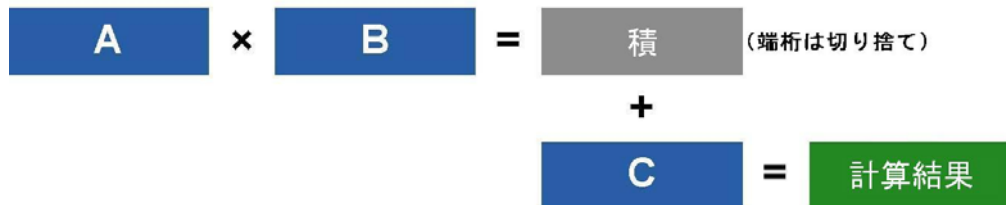
単精度浮動小数点命令は、IEEE 754-2008 規格で定められた 4 種類の丸めモード(最近隣、ゼロ、正の無限大、負の無限大)をすべてサポートしたほか、デフォルトで非正規化数もハードウェア処理できるようになりました。

非正規化数とは、使用する浮動小数点システムで表現可能な最小の正規化数とゼロの間に存在する数のことです。従来の GPU では非正規化数を切り捨ててゼロとしていたため、精度の低下を招いていました。一方 CPU は、非正規化数の計算を例外処理としてソフトウェア的に実現するのが普通で、数千サイクルも消費してしまいます。Fermi の浮動小数点ユニットはハードウェアで非正規化数を処理するため、パフォーマンスを落とすことなく、ゼロへと徐々にアンダーフローする処理が行えます。

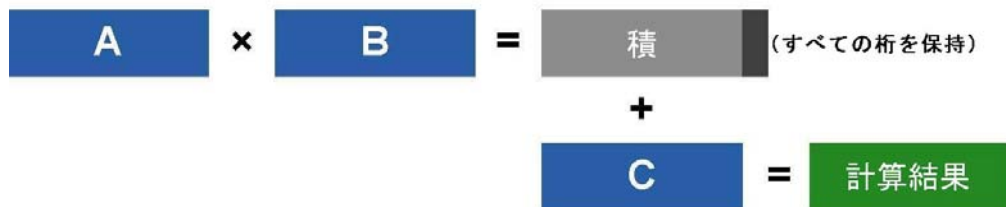
コンピュータグラフィックスや線形代数、科学的計算では、2 つの数字を掛けあわせ、その答えに第 3 の数字を足すという処理をよく行います。式で書くと、 $D = A \times B + C$ という処理です。従来の GPU には、この処理を高速化するため、2 回の演算を 1 クロックで行える MAD (Multiply-Add) 命令が用意されていました。MAD 命令では、まず切り捨て方式で乗算を行い、最近隣の偶数に丸める形で加算を行います。これに対し、Fermi では、32 ビットの単精度浮動小数点と 64 ビットの倍精度浮動小数点の両方について新しく FMA (Fused Multiply-Add) 命令を実装しました(GT200 は、倍精度についてのみ FMA をサポートしていました)。FMA は中間段階で精度を落とすことがないため、MAD 命令よりも高い精度で計算が行えます。計算精度が向上すると、細かく入り組んだジオメトリのレンダリングを行う、高い精度で反復計算を行う、正確な丸め処理による除算や平方根を高速で処理するなどさまざまなアルゴリズムにとって

大きなメリットとなります。

Multiply-Add (MAD):



Fused Multiply-Add (FMA)



プレディケーションで条件処理パフォーマンスを向上

プレディケーションをネイティブハードウェアサポートし分岐スレッド管理で活用していましたが、Fermi ISA では、この述部処理を命令レベルからも利用できるようになりました。プレディケーションが利用できると短い条件処理コードのセグメントを分岐命令オーバーヘッドなしで効率的に処理することが可能になります。

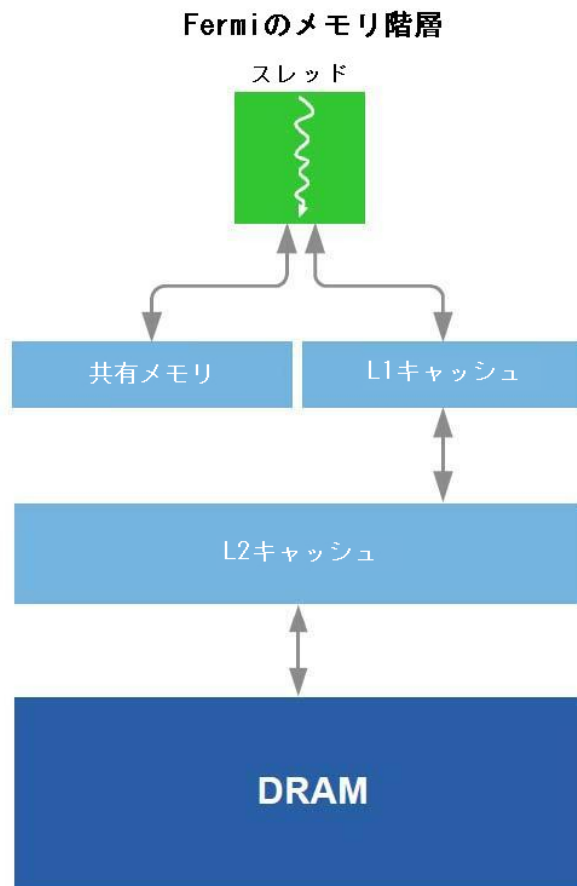
メモリサブシステムのイノベーション

構成可変な L1 キャッシュと統合 L2 キャッシュを持つ NVIDIA Parallel DataCache™階層

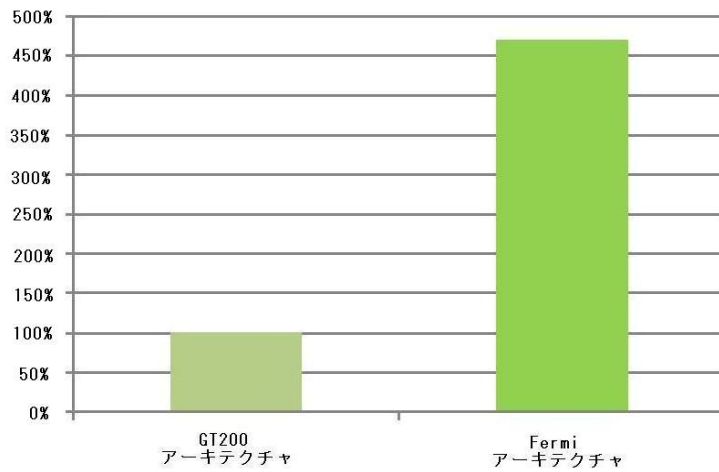
さまざまな分野で経験してきた何百種類もの GPU コンピューティング・アプリケーションを通じ、多くの課題で共有メモリが有効ではあるものの、すべての課題に適切というわけではないことが明らかになりました。必然的に共有メモリへマッピングするアルゴリズムもありますが、キャッシュを必要とするアルゴリズムもありますし、共有メモリとキャッシュを組み合わせなければならないものもあります。つまりメモリ階層としては、共有メモリとキャッシュ、

両方のメリットを併せ持ち、パーティションをプログラマが決められるものがベストとなります。Fermi のメモリ階層は、どちらにも対応しています。

ロード/ストアの操作について適切なキャッシュ階層を実装するには、難しい課題を解決する必要がありました。従来の GPU アーキテクチャは、テクスチャ操作に使用する読み取り専用の「ロード」パスとピクセルデータの出力に使用する書き込み専用の「エクスポート」パスを持つ形でした。しかしこれでは、読み出しと書き込みが順番に実行されることが前提となる一般的な C スレッドや C++スレッドが実行しにくいのです。たとえば、レジスタオペランドをメモリにスピルし、それを読み戻す処理で RAW (Read after Write) ハザードが生じます。つまり、読み取りと書き込みのパスが分離されていると、読み取り命令を実行する前に書き込みの「エクスポート」パスを明示的にフラッシュしないといけないかも知れません。更に読み取りパスのどのキャッシュも書き込みデータとコヒーレンシーが取れていません。

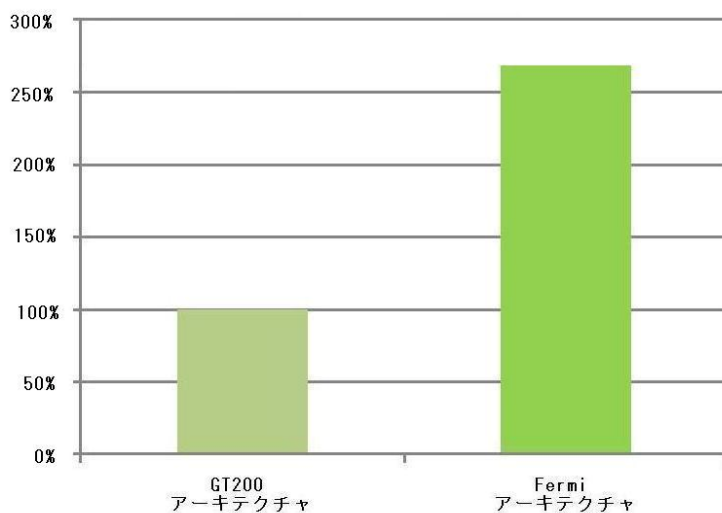


共有メモリを使用する基数ソート



Fermiで共有メモリに48KBを割り当てると、基数ソートの実行速度がGT200の4.7倍に上昇しました。

PhysXによる凸面への流体衝突の計算



Fermiが持つキャッシュの恩恵が特に大きいのは、流体シミュレーションなどの物理演算アルゴリズムです。凸面衝突の計算速度は、FermiがGT200の2.7倍でした。

そのため Fermi アーキテクチャでは、ロードとストアの両方をカバーする形でメモリリクエストの統合パスを実装し、SMごとに L1 キャッシュを持たせるとともにあらゆる操作(ロード、ストア、テクスチャ)をカバーする統合 L2 キャッシュを用意しました。SMに用意された L1 キャッシュは共有メモリとして使うようにも、ローカルメモリやグローバルメモリのキャッシュとして使うようにも構成可能です。64KBのメモリを、共有メモリ48KB と L1 キャッシュ16KB とするか、共有メモリ16KB と L1 キャッシュ48KB とするかが選べるのです。共有メモリに48KBを割り当てると、電気力学的シミュレーションなど共有メモリの利用が多いプログラムは実行速度が3倍程度まで上昇

します。メモリアクセス状況があらかじめわからない場合には L1 キャッシュに48KBを割り当てておけば、DRAMへの直接アクセスよりも格段に高いパフォーマンスを得ることができます。

いずれの構成でも、L1 キャッシュは、複雑なプログラムにおける一時的なレジスタ・スピルもキャッシュしてくれます。従来の GPU では DRAM に対して直接レジスタがスピルするため、アクセスレイテンシが増大していました。L1 キャッシュを搭載した結果、一時的なレジスタの利用が多い場合ほど大幅にパフォーマンスが向上するようになりました。

Fermi にはまた、ロード、ストア、テクスチャのすべてをカバーする統合 L2 キャッシュ、768KB が用意されています。L2 キャッシュ搭載により、GPU 全体でデータを効率的かつ高速に共有することが可能になりました。このキャッシュ階層の恩恵が特に大きいのは、物理演算ソルバー、レイトレーシング、疎行列の乗算など、データアドレスが事前に確定しないアルゴリズムです。複数の SM から同じデータを読む必要があるフィルターや畳み込み演算のカーネルにも大きなメリットがあります。

GPU として初めて ECC メモリをサポート

Fermi は、GPU として初めて、メモリ上データの ECC (Error Correcting Code) による保護をサポートしました。ECC は、ハイパフォーマンスコンピューティング環境でデータの完全性を確保したいと考える GPU コンピューティング・ユーザーから要望されていた機能です。特に、医療関連の画像処理や大規模なクラスタコンピューティングといった分野で強いニーズがあります。

メモリに記憶されたビットが自然界に存在する放射線の影響で変化し、ソフトウェアを引き起こすことがあります。1 ビットのソフトウェアを検出し、システムが影響を受ける前に修正するのが ECC テクノロジーです。使用するシステムの数に比例して放射によるエラーの可能性が高くなるため、大規模なクラスタインストールでは ECC が不可欠だと言えます。

Fermi は SECDED (Single-Error Correct Double-Error Detect) ECC コードをサポートしており、データアクセス時、ハードウェアに 1 ビットエラーがあればそれを修正することができます。SECDED ECC はまた、2 ビットエラーならば必ず、マルチビットエラーもその多くを検出・報告してくれるため、データがおかしくなったら処理を中断し、プログラムを実行しなおすことが可能です。

レジスタファイル、共有メモリ、L1 キャッシュ、L2 キャッシュ、DRAM メモリのすべてが ECC で保護される Fermi は、HPC アプリケーション分野における最もパワフルな GPU であるとともに最も信頼性の高い GPU なのです。Fermi はまた、チップ間の伝送時にもデータをチェックする業界規格もサポートしています。NVIDIA の GPU は、すべて、CRC チェックとデータリンク層におけるリトライを行う PCI Express 規格をサポートしているのです。Fermi はさらに、これとよく似た GDDR5 規格 (EDC と呼ばれます) もサポートしており、メモリバスでデータを伝送する際にも CRC チェックとリトライを行うことができます。

高速なアトミックメモリ操作

同時並行で処理を進めるスレッドが共有データ構造に対して読み込み-変更-書き込み処理を正しく行えなければならないため、並列プログラミングにおいてはアトミックなメモリ操作が重要となります。加算、最小値、最大値、比較-交換などの演算は、他のスレッドから割り込まれることなくデータの読み出しから変更、書き込みまでの処理が完了できなければならないため、アトミック演算と呼ばれます。アトミックなメモリ操作は、並列ソート、縮約演算のほか、スレッド実行のシリアライズに使われるロックなしでデータ構造を並列に構築する場合など、さまざまな場面で利用されます。

Fermi ではアトミック処理のハードウェアによるサポートが増えたほか、L2 キャッシュも搭載されたため、アトミック演算のパフォーマンスが GT200 に対して 20 倍にも向上しました。

GigaThread™ スレッドスケジューラ

2 レベルの分散型スレッドスケジューラも、Fermi アーキテクチャに搭載された重要なテクノロジーです。チップレベルでは、グローバルな処理分配エンジンがスレッドブロックを SM へと割り振り、SM レベルではそれぞれに実装されたワープスケジューラが 32 スレッド単位のワープを実行ユニットへと割り振ります。G80 に導入された第 1 世代の GigaThread は、12,288 スレッドをリアルタイムに管理する能力を持っていました。Fermi アーキテクチャでは、スレッドスループットを高めただけでなく、コンテキストスイッチングの高速化、コンカレントカーネル実行、スレッドブロックスケジューリングの改善なども実現しました。

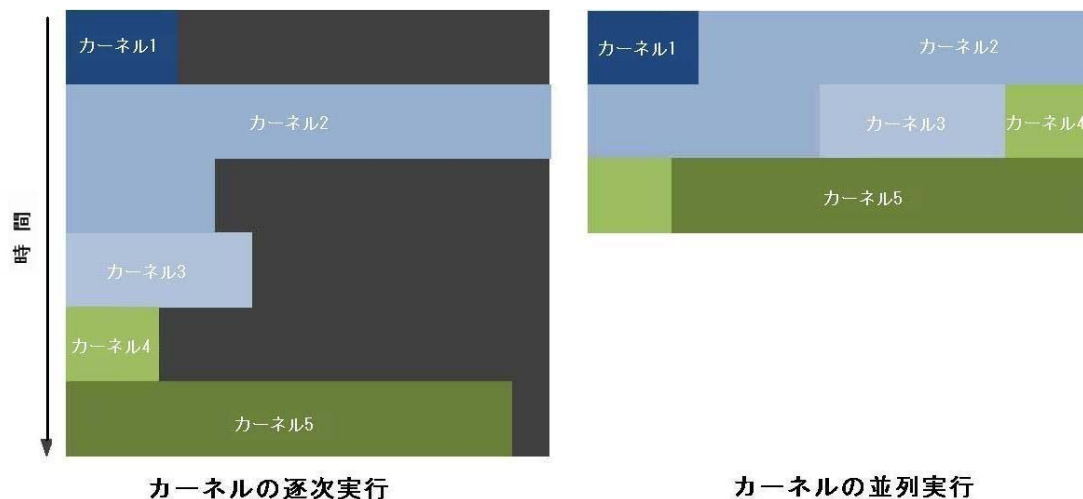
アプリケーションコンテキストの切り替え速度が 10 倍に向上

CPU と同じように GPU もコンテキストスイッチングによってマルチタスクをサポートしています。つまり、プロセッサのリソースを時間で分割し、各プログラムに割り当てているのです。Fermi パイプラインはアプリケーションコンテキストの切り替えに要する時間が 25 マイクロ秒以下と、従来の GPU と比較して大幅に最適化されています。この結果、パフォーマンスが大きく向上しただけでなく、グラフィックスと PhysX アプリケーションが緻密な連携を行う場合など、カーネル間通信を頻繁に行うアプリケーションを開発できるようにもな

りました。

コンカレントカーネル実行

Fermi はコンカレントカーネル実行をサポートしています。つまり、同じアプリケーションコンテキストのカーネル、複数を一つの GPU で並列実行できるのです。コンカレントカーネル実行がサポートされると、数多くの小カーネルを実行して GPU を無駄なく活用することが可能になります。たとえば、PhysX プログラムから呼び出す流体ソルバーと剛体ソルバーが、逐次実行ではスレッド・プロセッサの半分しか利用できないとします。このような場合でも、Fermi アーキテクチャでは同じ CUDA コンテキストのカーネル、複数を並列実行できるため、GPU リソースを無駄なく活用できるわけです。コンテキストスイッチングのパフォーマンスも改善されていますから、アプリケーションコンテキストが異なるカーネルも高い効率で逐次実行されます。

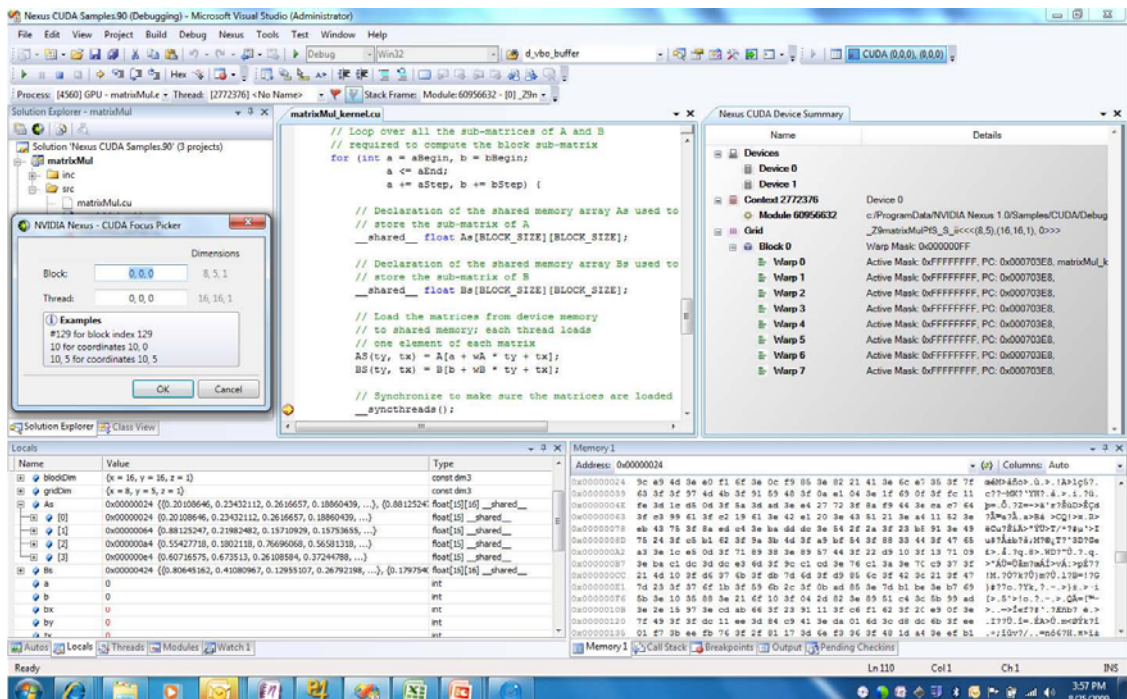


NVIDIA Nexus について

NVIDIA Nexus は、CUDA C、OpenCL、DirectCompute による大規模並列アプリケーションの開発に特化した初の開発環境です。Nexus を使えば、Microsoft Windows における統合アプリケーション開発環境として最も普及している Microsoft Visual Studio において、並列対応ハードウェアのソースコードに対するデバッグとパフォーマンス解析が行えるようになります。従来は CPU コードと GPU コードの間で開発効率に大きな差がありましたが、今後は違いがなくなるのです。

Nexus が登場した結果、CPU コードの記述とデバッグに使うツールとインターフェースで GPU ソースコードの記述とデバッグが行えるようになりました。ソースブレークポイントやデータブレークポイントの設定、メモリ検査などが行えるのです。並列に稼働する数千ものスレッドからスレッド 1 本に着目してデバッグする、並列スレッド全体の計算結果をシンプルかつ効果的に可視化するなど、大規模並列処理の管理ツールが使えるなど、Visual Studio の機能拡張も実現されます。

Nexus は、CPU と GPU、両方を活用するコプロセッシングアプリケーションの開発に最適な環境です。両方のプロセッサからパフォーマンスに関係するイベントと情報を集め、時系列で相関関係がわかる形にして開発者に提示してくれます。あるサブシステムやプロセッサだけを見るのではなく、システム全体でアプリケーションの挙動とパフォーマンスを見ることができるのです。



NVIDIA Nexus 統合開発環境

まとめ

NVIDIA では、16 年にわたり、世界最速のグラフィックスプロセッサの構築に専念してきました。そのため、GPU コンピューティングの先駆けとなった G80 アーキテクチャも、それを大幅に改良した GT200 も、グラフィックスの世界に根ざした設計となっていました。これに対し、NVIDIA が提唱する新しい方向性を示すもの、それが Fermi アーキテクチャです。GT200 の単なる後継としてではなく、GPU というものの役割、目的、能力を根底から考え直すことから Fermi は生まれたのです。

Fermi チームは実行ユニットを増やすという楽な道を選ばず、GPU コンピューティングに伴う難しい問題への対応に注力しました。データの局所性が重要となることから、Fermi では、2 レベルのキャッシュ階層を用意するとともにメモリに対するロード/ストアのパスを統合しました。倍精度演算はスーパーコンピューター並みまでパフォーマンスを強化しましたし、アトミック演算も 20 倍と高速化しました。包括的な ECC サポートも実現し、ハイパフォーマンスコンピューティング市場を重視する姿勢を改めて示しました。

ソフトウェア方面については、世界で最も広く使われているオブジェクト指向のプログラミング言語、C++をサポートしました。また、大規模並列 GPU コンピューティングのアプリケーション開発を念頭に設計された世界初の統合開発環境、Nexus も完成しました。

優れたパフォーマンスと機能、プログラミング性を併せ持つ Fermi アーキテクチャは、GPU コンピューティングに次なる大変革をもたらすことになるでしょう。

注記

解説、見解、NVIDIA のデザイン仕様、リファレンスボード、ファイル、図面、診断、リスト、その他のドキュメントなど、本ホワイトペーパーで提供する情報は(以下、集合的あるいは個別に「マテリアル」と呼称します)、すべて、「現状のまま」を条件に提供するものです。NVIDIA は、本マテリアルについて、明示的、暗示的、法定的を含む保証を一切行わず、また、権利の不侵害、商品性、および特定目的への適合性に関するあらゆる黙示保証を明示的に放棄するものとします。

記載された情報の正確性、信頼性には万全を期しておりますが、これらの情報の使用の結果として、もしくはこれらの情報の使用に起因して第三者の特許またはその他の権利の侵害が発生しても、NVIDIA Corporation は一切の責任を負わないものとします。暗示的に、もしくは NVIDIA Corporation が所有する特許または特許権に基づき、付与されるライセンスは一切ありません。本書に記載の仕様は予告なしに変更されることがあります。本書は、過去に提供されたすべての情報よりも優先されます。NVIDIA Corporation の製品は、NVIDIA Corporation の明示的な書面による許可なくしては、生命維持装置の重要な部品として使用することはできません。

商標について

NVIDIA、NVIDIA ロゴ、CUDA、FERMI、GeForce は、米国およびその他の国における NVIDIA Corporation の商標または登録商標です。その他の会社名および製品名は、各社の登録商標または商標です。

Copyright

© 2009 NVIDIA Corporation. All rights reserved.