Vol.2011-HPC-131 No.1
2011/10/6

データストリーム処理における GPU 統合型 CPU の予備的評価

松浦 紘也[†]　　上野 晃司[†]　　鈴村 豊太郎[†,††]

CPU と GPU をオンダイ統合した APU の登場により，高スループットと低レイテンシの両立が求められるデータストリーム処理での GPU のさらなる活用が期待される．本研究では，マイクロベンチマークと株価加重平均 VWAP，異常検知アルゴリズム FELIX SST でこれを検証し，詳細なプロファイルを行うことで APU の性能特性を評価した．この結果，APU では外付け GPU と比べカーネル起動に 125.7 倍，メモリ送受信には 0.5ms から 2ms のオーバヘッドが見られ，現状ではデータストリーム処理への適用は困難であるが，行列積計算では CPU の 58.6 倍ものスループットがあり，ドライバや実行環境が改善されれば APU の適用が有効であることを確認した．

Preliminary Evaluation of CPU integrated with GPU for Data Stream Processing

Hiroya Matsuura[†],　Koji Ueno[†] and Toyotaro Suzumura[†,††]

APU is a new computing device that integrates GPU with CPU. We expect that APU can achieve higher performance than CPU owing to many computing units and much faster response compared with discrete GPU because APU shares memory controller between CPU cores with GPU cores. Such devices are suitable for Data Stream Processing, a new computing paradigm, which needs both high throughput and low latency. In this paper, we examine APU's performance in detail to determine whether it would be suitable for Data Stream Processing with microbenchmarks, VWAP (Volume Wated Average Price of Stocks), FELIX SST (Change Point Detecting Algorithm) and their profiling. APU achieved 58.6 times faster than CPU on Matrix Multiplication but have 125.7 times longer response time to launch kernel program and 0.5ms – 2ms overhead to transfer data compared with discrete GPU. According to these evaluation results, we conclude that APU can be a suitable for Data Stream Processing if there are improvements of APU drivers and environments.

1. 研究の背景

(1) データストリーム処理

データストリーム処理[1][2]とは，様々な形式のオンラインデータをオンメモリで逐次に処理することで，即時的な応答を実現する計算パラダイムである．MapReduce[3]のような，バッチ処理と呼ばれるデータを全て蓄積してから処理をはじめるという手法に比べ，データストリーム処理では最初のデータが到達し次第計算処理を開始できるため，応答性が重要である計算や，前後の限られたデータや統計情報のみを参照すればよい演算，全データの蓄積が物理的に困難な処理に適している．

このような処理方法は株式自動売買や環境モニタリング，工場の異常検知などで実用化されており，近年重要性が高まっている．

(2) GPGPU

GPGPU(General Purpose GPU)とは，三次元画像処理の計算素子である GPU(Graphic Processing Unit)を汎用の数値計算に応用するという計算手法である．GPU は，CPU に比べ，シンプルだが非常に多くの演算コアとレイテンシは大きいが広い帯域幅のメモリを持ち，多体問題や流体シミュレーション，行列計算などの並列度が高く広帯域のメモリアクセスを行うアプリケーションで効果を発揮する．

(3) 既存のデバイスの問題点と GPU 統合型 CPU の登場

近年，熱やプロセスの問題から CPU 単独での性能向上は難しく，マルチコアによる処理能力向上にも限界があり，計算処理のさらなる高速化には CPU と GPU のような異なる特性のハードウェアを組み合わせたヘテロジニアスプロセッシングへの移行が求められている．

こうした点から GPGPU は今後重要な技術であるが，既存の GPU にも様々な問題点が挙げられる．従来の外付け GPU を汎用計算に用いるには，一旦ホストメモリと呼ばれる CPU 側のメモリ領域からデバイスメモリと呼ばれる GPU 側のメモリへとデータを転送し，GPU での計算を行った後にさらにホストメモリ側へデータを転送する必要がある．このデータ転送は PCI Express 経由で行われるため，転送時間の増大が問題となっている．また，こうしたアーキテクチャではメモリ空間が物理的にもソフトウェア的にも分断されるため，GPGPU への移行の障壁となっている．

こうした背景から，CPU と GPU を 1 つの半導体上に統合した APU(Accelerated Processing Unit)と呼ばれる新たな計算素子が登場した．APU ではメモリコントローラやキャッシュレベルで CPU と GPU が統合されており，共通のメモリにそれぞれのデ

[†] 東京工業大学
Toyo Institute of Technology
[††] IBM 東京基礎研究所
IBM Research Tokyo

ⓒ 2011 Information Processing Society of Japan

バイスが直接アクセスできるため，転送なしでのデータの相互利用が行える．これよりメモリバンド幅は CPU と同程度に制限されてしまうが, 数百単位の演算素子を持つため，並列度の高いアプリケーションでは CPU よりも遥かに高い計算性能を誇る．こうした性能特性から，CPU と外付け GPU のそれぞれの欠点を補うことや，新たな研究・応用領域の開拓が期待されている．(図 1)

また，今までは GPU を追加するには筐体や電源のサポートが必要で，GPGPU 対応環境が限られていたことが問題だったが，APU が広まることにより GPGPU の実行環境が幅広いユーザ層に普及することも併せて期待される．

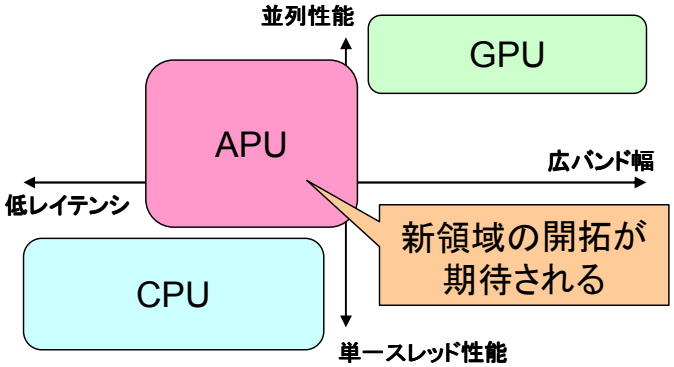


図 1　APU により性能向上が期待される問題領域

2. 本研究の目的と貢献

前章で述べた通り，APU の登場によって既存の CPU や GPU ではカバーできなかった領域への GPGPU の適用が期待できる. CPU では並列度が足りないが外付け GPU への転送時間が許容できない計算処理や，CPU に適した計算部分と GPU に適した計算部分が細分化されておりデータ転送頻度が高い計算処理がこれに当たる．

本研究では, 特に前述のデータストリーム処理への適用を主眼に置いて APU の有効性の調査を行う．データストリーム処理ではネットワーク越しにデータが到来し，到着と同時に即時に計算を行うため，データ転送オーバーヘッドの大きい外付け GPU を用いた先行研究[4][5]では元々粒度の大きい計算処理や高度な最適化を施さなければ GPGPU の恩恵を受けることが難しかった．APU により，こうした制約に囚われず幅広いデータストリーム処理のアプリケーションの高速化が期待される．これを実証するために, マイクロベンチマークと実アプリケーションを実装し APU 環境を総合的に評価することが本研究の目的である．

本研究の貢献としては，マイクロベンチマークと実アプリケーション，プロファイル結果を用いて APU の性能特性を詳細に評価した点が挙げられる. 応答性とスループットの両面に着目した APU の評価結果は未だ存在せず, データストリーム処理という応用領域を背景とした調査も本研究が初めてである．

演算器単位での命令のスループットやメモリバンド幅などは公式にデータが好評されているが，メモリアクセスやカーネル起動のレイテンシなど，データストリーム処理のような高応答性を要求するアプリケーションを考慮した評価は存在せず，本研究の評価結果の有用性は高い．

3. 評価アプリケーションの実装

APU 環境の評価として，メモリ転送とカーネル関数起動のオーバーヘッド測定，行列積計算，VWAP[6]，FELIX SST[7][8]を用いた．行列積計算には AMD APP SDK v2.5 のサンプルの MatrixMultiplication を用い，それ以外は C++と OpenCL で実装した．

外付け GPU との比較のために，マイクロベンチマークの一部は NVIDIA の CUDA でも同様の実装を行った．

実アプリケーションの評価では CPU との比較を行うため, FELIX SST では上野ら[5]による最適化実装を用い，VWAP では OpenCL 実装を CPU で実行した．

3.1 OpenCL

OpenCL とは, Khronos グループにより策定された C/C++を拡張した並列ハードウェア用のプログラミング言語である．OpenCL は異なる計算デバイスやメモリ空間が混在した計算環境の効率的利用を目的としており，デバイス利用のための様々な API，デバイスごとの演算特性やコア数の違い，データの局所性を利用するための機構を備えている．

プログラミングモデルは NVIDIA GPU のための GPGPU 用言語，CUDA のモデルを継承しており，ホスト側でデバイスへのデータ送受信や，デバイス側のコード（カーネル関数）の起動を管理するようになっている．OpenCL でも CUDA と同様に，デバイス内部でのスレッド数や階層キャッシュの各スレッドへの分配もホスト側から細かく指定することが可能になっている．

3.2 VWAP

VWAP とは Volume Weighted Average Price のことで，株価の加重平均計算のことを指す．株式売買を行うべきかどうかの指標として用いられ，一つ一つの計算負荷は軽量だが膨大なデータを即時に処理することが要求される，データストリーム処理の代表的な応用例とである．

ⓒ 2011 Information Processing Society of Japan

先行研究では，CPU，Cell と外付け GPU で VWAP の処理速度を比較しているが，外付け GPU はカーネル関数から GPU メモリへのアクセスレイテンシが大きく，CPU や Cell に劣る結果となっている．APU では CPU と GPU が同一のメモリコントローラを共有するためレイテンシの削減が期待され，CPU に比べ多数の演算コアを持つため計算部分の高速化が期待される．

現実の株式取引ではそれぞれの銘柄は非同期で売買されるが，本研究の実装ではスループットの最大値を測るため，常に取引データがキューにある状態を想定する．加重平均の計算は銘柄(ticker)ごとに行われるため，ticker 数分の計算スレッドを起動する．今回は 10000 回の計算を行い，その実行時間を比較する．

3.3 FELIX SST

FELIX SST(Feedback impLIcit kernel approXimation SST)[8]とは，特異スペクトル変換(SST：Singular Spectrum Transformation)[7]という特異値分解（SVD：Singular Value Decomposition）を用いた変化点検知アルゴリズムを，近似により改良したアルゴリズムである．SST は過去と現在のデータを比べることでモデル不用の変化点検知を実現し，高い異常検出性能を誇るが，密行列に対して特異値分解を行うため，計算量が行列サイズの三乗に比例と非常に大きくなってしまう．FELIX SST では行列圧縮を行うことで計算時間を抑えながら SST の高い検出性能を維持することができ，工場や環境モニタリングシステムなどの異常検知アルゴリズムとして，データストリーム処理の有用なアプリケーションとなっている．

GPU 側の実装は，上野らの先行研究[5]による NVIDIA GPU 用の CUDA 実装を OpenCL に移植し，性能評価を行った．本実装では，行列圧縮までを GPU で行い，変化度スコアの計算を CPU で行っている．FELIX SST を GPU で実行する際，1024 以上の巨大な行列では外付け GPU が CPU よりも高速だが，それを下回る小規模な行列に対しては CPU の処理時間を上回ってしまう．先行研究では，GPU 計算部分にタスク並列を導入し，小規模な行列を同時に並列して計算することで CPU よりも高速な処理を実現しているが，タスク並列の実現には GPU のアーキテクチャに依存した高度な最適化が必要となってしまう．APU を導入することにより，複雑な最適化を施すことなく CPU と GPU の両方で高速化が困難な処理要求を満たすことが期待される．

CPU での処理には CBLAS と LAPACK を用いた Native 実装と OpenCL 実装の両方を用い，OpenCL の APU 実行版と計算時間を比較する．

4. 性能評価

現在 OpenCL に対応した APU は AMD 社のものしか存在しないため，評価にはこれを用いる．AMD E-series はローエンド向けの製品で，ミドルレンジ向けである A-series でも比較実験を行うべきだが，今回は機材の都合がつかず実験は見送った．また，マイクロベンチマークの比較対象として NVIDIA の外付け GPU を用いる．

APU 環境は，Linux 環境と Windows 環境のそれぞれで評価を行い，AMD の OpenCL 実行環境である AMD APP SDK のバージョン間の性能比較も行った．

4.1 評価環境

(1) ハードウェア環境

ハードウェア環境は，AMD の APU である E-350 を搭載した APU 環境と、NVIDIA の外付け GPU である Tesla C1060 を搭載した CUDA 環境の 2 つを用意した．

APU 環境は，APU として 1.6GHz 2 コア L2 1MB の CPU と 492MHz 80sp L2 64kB の GPU を統合した E-350 を搭載し，シングルチャネル DDR3 2GB のうち 1.5GB を CPU 側，512MB を GPU 側に割り当てる．

CUDA 環境は，Phenom 9850 2.5GHz 4 コア L2 2MB，デュアルチャネル DDR2 8GB を搭載し，外付け GPU は Tesla C1060 240sp 1.296GHz GRAM 4GB となっている．

(2) ソフトウェア環境

APU 環境は Linux と Window での比較を行う. Linux 環境は, Scientific Linux 6.1 kernel 2.6.32 AMD64，gcc 4.4.5，OpenCL 1.1 対応の AMD APP SDK v2.4 と v2.5，AMD GPU ドライバに Catalyst 11.7 で，Windows 環境は Windows 7 64bit，Visual C++ 2010，AMD APP SDK v2.4，Catalyst 11.4 となっている．

CUDA 環境は，CentOS 5.3 kernel 2.6.18 AMD64，gcc 4.1.2，CUDA 3.2，devdriver 3.2 を用いる．

特に断りがない場合，APU 環境は SDK v2.5 を利用し，実験結果は 10 回実行して特異点を上下 2 つ取り除き平均を取った値を利用している．また，プロファイル結果の一部に$から始まる文字列があるが, モジュールのパス名や一時フォルダ内に生成される乱数名など枠に収まりきらない重要でない情報を省略したことを表している．

4.2 マイクロベンチマーク

APU の並列計算のスループット，各種応答性能を測定するためにマクロベンチマークによる性能評価を行った．

(1) 行列積

APU 環境の CPU と GPU の計算性能を比較するために，行列積を用いてスループットの評価を行った. 単精度の正方行列の 1 辺の長さ n を 8 から 1024 まで 2 倍ずつ変化させ，FLOPS 値を測定した．行列サイズ n が 1024 までなのは，それ以上試した場合でも FLOPS 値の改善が見られなかったためである．GPU のローカルキャッシュは利用していない．

GPU 実行では行列サイズが増えるほどスループットも向上しており，n=1024 のときに最大値の 42.02GFLOPS を記録している．CPU は n=128 での 0.71GFLOPS をピークに性能が低下しているが，E-350 CPU の L1-D キャッシュは 32kB なので，キャッシュミスの影響と思われる．(図 2)

ⓒ 2011 Information Processing Society of Japan

E-350 の GPU 性能は CPU と比較すると 58.6 倍ものスループットを達成しており，行列計算のように十分な並列度を得られるアプリケーションでは CPU よりも遥かに優れた性能を発揮することが確認できる．

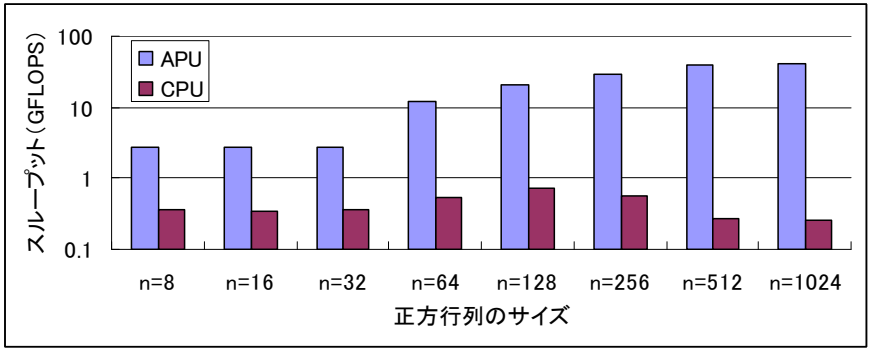


図 2　行列積のスループットの APU と CPU の比較（対数グラフ）

(2) カーネル起動レイテンシ

GPU で計算処理を行う場合，CPU コードであるホスト側から GPU コードであるカーネルプログラムを起動しなくてはならない．GPU でバッチ処理を行う場合や，CPU のみでデータストリーム処理を行う場合にはこうしたレイテンシの影響は考慮せずに済む．しかしながら，データストリーム処理ではデータの到着のたびに GPU へデータを転送し，カーネルを起動しなくてはならないため，CPU-GPU 間通信のボトルネックも考慮する必要がある．

APU では上記の制約を改善するために CPU と GPU を同一ダイ上に統合したが，その効果を確かめるために NVIDIA の外付け GPU を用いた CUDA 環境と比較する．1 スレッドで起動し計算処理を行わないカーネル関数を 10000 回実行するプログラムを OpenCL と CUDA でそれぞれ作成し，1 回分のカーネル起動時間を評価する．スレッド数を 256 まで増やした結果も測定したが，起動時間に有意な差は見られなかった．

表 1 はカーネル起動時間をまとめたものだが，APU は Tesla の 125.7 倍という莫大なレイテンシが発生している．この原因を探るため，プロファイルを行った．

表 1　E-350 と Tesla のカーネル起動レイテンシ

	E-350 SDK	Tesla
経過時間	153.896 [usec]	1.223475 [usec]

samples	%	app name	symbol name
8279	34.3584	libaticaldd.so	/usr/lib64/libaticaldd.so
3272	13.5790	fglrx	/fglrx
2858	11.8609	vmlinux	native_safe_halt
1966	8.1590	libamdocl64.so	$SDK_PATH/lib/x86_64/libamdocl64.so
366	1.5189	libc-2.12.so	memcpy
297	1.2326	libc-2.12.so	GI_memset

図 3　APU のカーネル起動のプロファイル結果（1%以上のもの）

図 3 は OProfile によるプロファイル結果である．プロファイル結果は占有率が 1%以上のものだけを抜粋しており，CPU 使用率は 1 コアではユーザ空間が 100%に達しており，もう 1 コアはほぼアイドル状態だった．この結果より，AMD 関連のライブラリと GPU ドライバ，カーネルの native_safe_halt システムコールが大きな割合を占めており，ドライバのオーバーヘッドや不適切な halt により起動時間が大幅に増加しているのではないかと推測される．

現状では外付け GPU にも劣る結果であり，データストリーム処理での利用は困難だが，ハードウェアとしては進歩的な構成を取っているため，今後のドライバやライブラリなどのソフトウェア環境の整備が待ち望まれる．

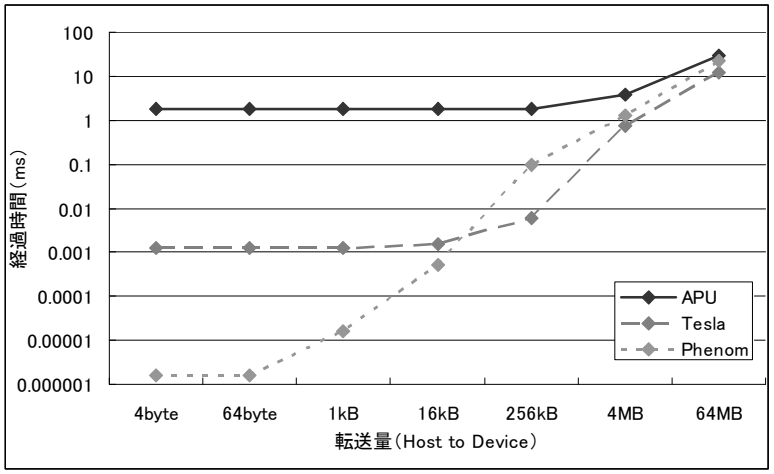
(3) メモリ送受信

カーネル起動と同様に，CPU-GPU 間メモリ送受信のオーバーヘッドを CUDA 環境と比較する．通信サイズは，4byte から 64MB まで 16 倍ずつの刻みで実験した．Tesla 以外に CUDA 環境の CPU でのメモリコピー関数 memcpy によるデータコピーとも比較する．

図 4 はメモリ送受信の評価結果の図である．APU は 16kB 以下では経過時間が一定で非常に遅く，CPU とメモリコントローラを共用していることを考えると，ドライバなどのオーバーヘッドが 0.5ms から 2ms 程度存在すると考えられる．Tesla でも 16kB 以下で同等の傾向が見られるが，1us 程度と APU に比べ非常に小さく抑えられている．CPU は 4byte を除くと線形に転送時間が増えており，特に 16kB 以下の小規模な転送では最も高速だが，それ以上では Tesla が最も高速という結果になっている．これは APU や CPU 内でのコピーだと読み書きが衝突するのに対し，外付け GPU では外部の GPU メモリと読み書きが分散されるからである．

APU はキャッシュの影響を受けない 4MB 以上の領域でも CPU や外付け GPU と比べ，3 倍から 5 倍程度の処理時間が発生しており，これを確かめるためにプロファイルを行った．

ⓒ 2011 Information Processing Society of Japan



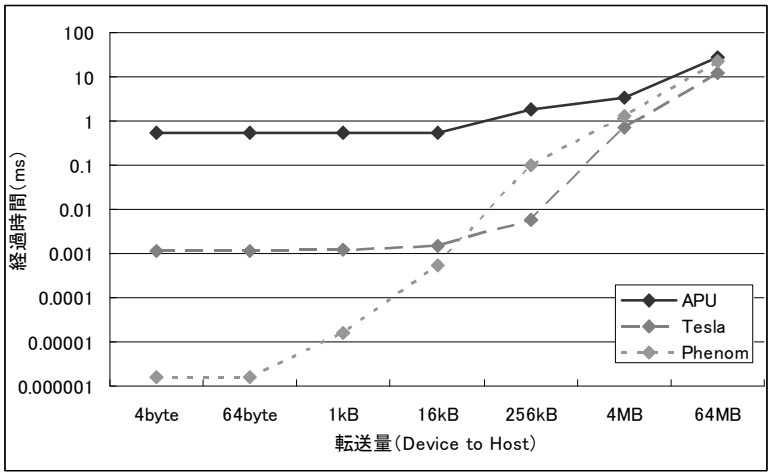


図 4　データ送受信の評価結果（対数グラフ）

samples	%	app name	symbol name
176292	45.4540	fglrx	/fglrx
120336	31.0267	libaticaldd.so	/usr/lib64/libaticaldd.so
9851	2.5399	vmlinux	put_compound_page
9207	2.3739	vmlinux	native_safe_halt
8145	2.1001	vmlinux	follow_page
6111	1.5756	libamdocl64.so	$SDK_PATH/lib/x86_64/libamdocl64.so

図 5　APU メモリ受信のプロファイル結果（1%以上のもの）

図 5 がプロファイルの実行結果である．実験では全体で 1 コアの 100%を利用し，そのうち OS カーネルが 50%前後を占めていた．AMD のドライバ，OpenCL 関連が多くの時間を占めており，カーネル起動と同様にこれらのプロセスが全体の速度低下を招いていることがわかる．特に，GPU ドライバは半分近くの処理を占有しており，Linux の GPU ドライバである fglrx と libaticaldd.so の部分がなくなるだけでもかなりの高速化が期待できる．

本章で詳細な比較を行ったので，以降の測定ではカーネル起動とメモリ送受信の CPU，GPU との比較は行わない．また，APU のカーネル起動時間は今後 Tesla 以下に抑えられる可能性があり，純粋な計算時間の比較を行うため，APU の計算時間を CPU，GPU と比較する際にはカーネル起動時間を除いたものを利用する．

4.3 VWAP

OpenCL で実装した VWAP を用い，計算する ticker 数を変えて評価を行う．ticker 数は 1，10，20，40，80，160，320，640 で，10000 回計算時の合計で比較する．

VWAP を比較する環境は，SDK v2.4 での Linux 環境と Windows 環境，Linux 環境の CPU 実行と APU 実行についての比較を実施する．Windows 環境を評価した理由は，Linux 環境に比べより高速なメモリ転送がサポートされているためである．Linux 環境と Windows 環境の比較には SDK v2.4 を，Linux 環境での CPU と APU の比較には SDK v2.5 を用いた．

VWAP では，CPU 実行も OpenCL で書かれた同一のコードを用いて評価する．

ⓒ 2011 Information Processing Society of Japan

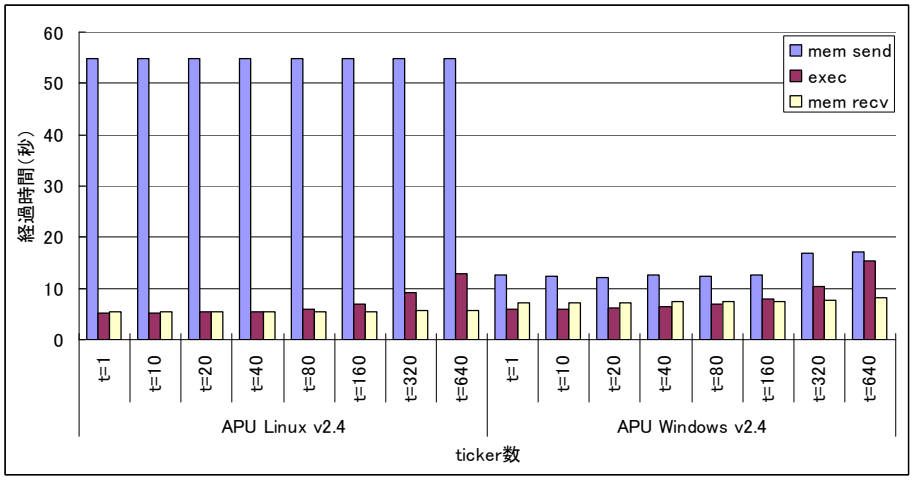


図 6　Linux 環境と Windows 環境の比較

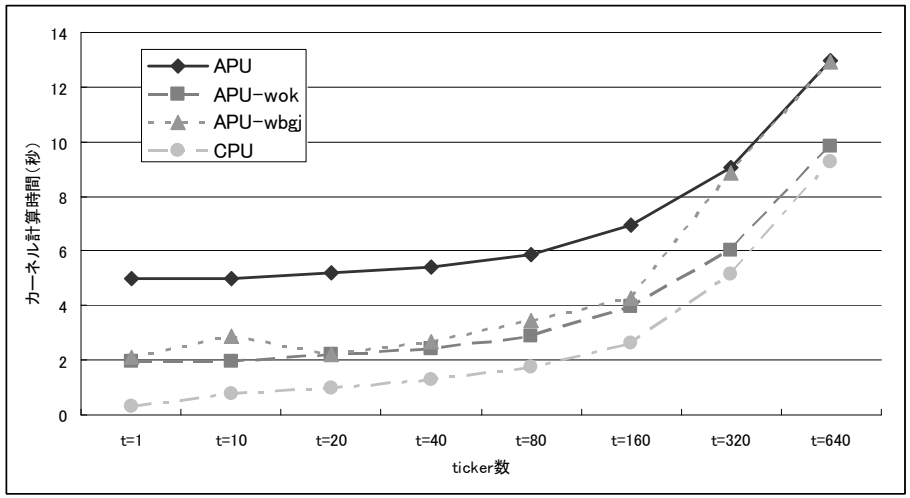


図 7　APU と CPU のカーネル計算時間の比較

図 6 は，SDK v2.4 で Linux 環境と Windows 環境での CPU-GPU 間メモリ送受信と VWAP 計算カーネルの実行時間をまとめたものである．ホストからデバイスへの送信は，Linux では 5 倍近くかかっている．制約付きながらホストのメモリ領域をデバイス側へコピーせずに使えるという Zero Copy という機能も Linux ではサポートされていない．逆に，デバイスからホストへのコピーとカーネル部分は僅かながら Linux が高速という結果になっており，OS や SDK などの実行環境により結果に差が出ることが判明した．

図 7 は，APU と CPU のカーネル計算部分の実行時間を比較したものである．APU と CPU は，それぞれ OpenCL コードのカーネル計算部分 10000 回の計算時間を積算したものだ．APU-wok は，APU の結果から事前に測定した空のカーネルの起動オーバーヘッドを引いたもので，これは将来的に APU のカーネル起動が Tesla 以下のコストになった場合を想定したものだ．APU-bgj は，VWAP 計算時に別プロセスで CPU に負荷をかけたものを指す．負荷には bash の while ループと echo コマンドを用い，常に 1 コアの CPU 使用率が 100%になっていることを確認した．現在の SDK では，今回の VWAP OpenCL 実装のように CPU にほとんど負荷がかからないアプリケーションではカーネルのオーバーヘッドが大きくなるという現象が報告されており，これを確認するためにこのような実験を行った．

APU は CPU に比べ 4〜5 秒ほど処理時間が余分にかかっているが，カーネル起動時間を無視した APU-wok とその差は 1 秒以内にまで縮まる．AMD E-series では演算ユニットが 80 しか存在しないため，ticker 数に応じて処理時間が増えてしまっているが，今後登場するより多くのコアを積んだ APU ならば，ticker 数がさらに多い場合に CPU を上回る可能性がある．

APU-wbgj は ticker が 160 以下では APU-wok と同等の処理時間となっており，負荷によって OpenCL のレイテンシが低下することが確認できる．しかし，ticker が 320 以上では負荷なしの APU と同じ処理時間に戻っており，GPU スレッド数が多い場合には効果が薄いようだ．

CPU での OpenCL 実行はネイティブ実装に比べ性能は劣るが，VWAP のような計算量が非常に小さいアプリケーションでも十分な並列度とデータがあれば，APU のようなデバイスも一部有効であるかもしれない．しかしながら，APU を実環境で利用するにはメモリ転送やカーネル起動のオーバーヘッドなど，解決しなければならない課題は非常に多い．

4.4 FELIX SST

3000 回 SST 計算を実行した際の全ての処理時間の合計を APU OpenCL 実装，CPU OpenCL 実装と CPU Native 実装で比較する．ただし，OpenCL の初期化部分は除く．

また，APU OpenCL 実装，CPU OpenCL 実装，CPU Native 実装それぞれのプロファイル結果も併せて掲載する．

ⓒ 2011 Information Processing Society of Japan

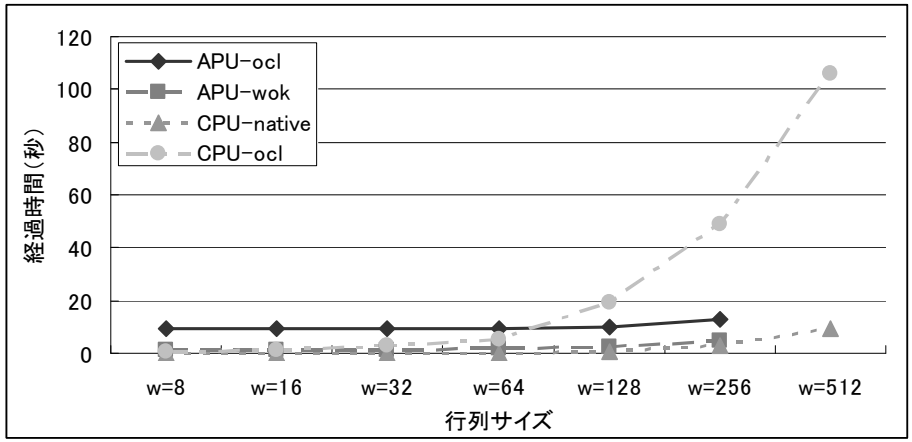


図 8　FELIX SST の実行時間の比較

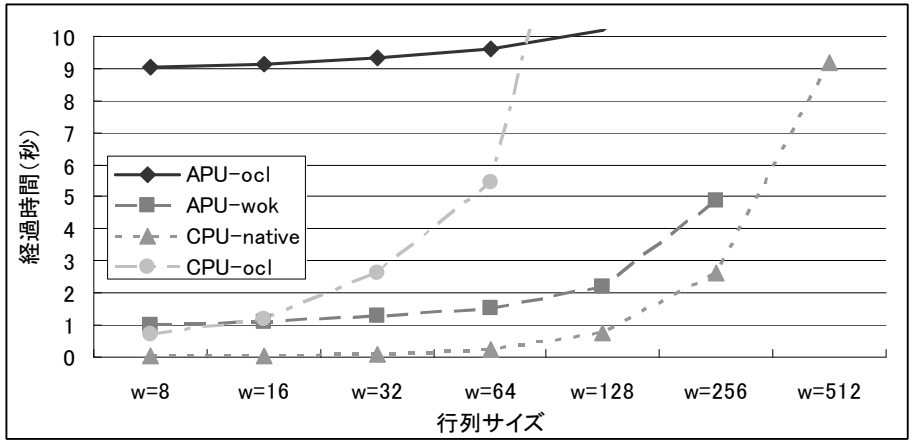


図 9　図 8 の 10 秒以下の部分の詳細

図 8，図 9 は FELIX SST の実行時間の比較である．CPU-ocl の実行時間が非常に長いため，10 秒以下の部分も併せて掲示する．OpenCL 実行（APU-ocl，CPU-ocl）における行列のウィンドウサイズが 256 までなのは，AMD の実行環境におけるローカルグループのスレッド数の上限が 256 であるためで，OpenCL そのものはそれ以上の数のスレッド数もサポートしている．APU-wok は VWAP の実験時と同様に，CPU-GPU 間データ転送時間と事前に測定した空のカーネル実行時間の両方を引いた値を仮想的に用いている．

CPU-ocl は OpenCL 環境が CPU に最適化されていないため，w=128, 256 では APU-ocl にも劣る結果となっている．APU-ocl はそのままではメモリ送受信部分の影響が大きすぎて比較できないが，APU-wok は CPU-native に対して最大 2 倍程度の実行時間まで迫っている．Native 実装以上の速度が出ない理由として，カーネル関数内部から共用メモリへのアクセス時間が大きいことが原因と考えられる．

samples	%	image name	app name	symbol name
29769	31.9822	fglrx	fglrx	/fglrx
23410	25.1504	vmlinux	vmlinux	walk_system_ram_range
11442	12.2927	vmlinux	vmlinux	native_safe_halt
5562	5.9755	libaticaldd.so	libaticaldd.so	/usr/lib64/libaticaldd.so
2464	2.6472	libamdocl64.so	libamdocl64.so	$SDK_LIB

図 10　FELIX SST APU-ocl のプロファイル結果（1%以上のもの）

samples	%	image name	app name	symbol name
514468	64.1244	OCLcAQTAg.so	OCLcAQTAg.so	/tmp/$FILE_NAME
255692	31.8700	libamdocl64.so	libamdocl64.so	$SDK/libamdocl64.so

図 11　FELIX SST CPU-ocl のプロファイル結果（1%以上のもの）

samples	%	app name	symbol name
40341	91.6862	cpu_sst_native	$FILE_NAME
437	0.9932	vmlinux	native_safe_halt
369	0.8387	cpu_sst_native	ATL_saxpy_xp1yp1aXbX
300	0.6818	cpu_sst_native	ATL_sscal_xp1yp0aXbX
290	0.6591	cpu_sst_native	ATL_sdot_xp1yp1aXbX

図 12　FELIX SST CPU-native のプロファイル結果（上位 5 つ）

ⓒ 2011 Information Processing Society of Japan

図 10，図 11，図 12 はそれぞれ APU-ocl，CPU-ocl，CPU-native のプロファイル結果である．CPU 実行の両者では，計算プログラムが上位を占めており，効果的にプロセッサが使われていることがわかる．それに比べ，APU ではドライバの fglrx や walk_system_ram_range などの計算部分以外に実行時間が奪われており，こうした部分の改善が今後必要と思われる．

4.5 評価のまとめ

APU は行列積では CPU より遥かに優れた処理性能を誇るが，データ転送やカーネル起動など，応答性に関わる部分では外付け GPU に比べカーネル起動に 125.7 倍，256kB 以下のメモリ送受信では 0.5ms から 1ms もの応答時間が発生している．内部設計や電気的には外付け GPU より応答性に優れるため，ドライバやその他ソフトウェアが発表されてから日が浅く，チューニングが進んでいないためだと考えられる．これはプロファイル結果からも明らかで，今後のソフトウェア層の改善が期待される．

5. 関連研究

5.1 CPU，GPU，Cell での VWAP の性能比較

本研究と同様に，VWAP の性能を各デバイスで評価したという研究[6]である．並列演算デバイスをデータストリーム処理に応用するという点では本研究と類似しているが，この論文中では株式の銘柄を内部行列として保持し，行列から各スレッドがデータを引いてくるという設計になっている．これより，メモリへのランダムアクセスの性能が最も影響するベンチマークとなっており，データストリーム処理の実アプリケーションで想定される負荷を網羅しているとは言い難い．また，APU 登場以前の論文であるため，こちらは未評価である点も本研究とは異なっている．

5.2 複合型計算システムにおける実行時自動チューニング

OpenCL が対象とするデバイスは，デバイスごとに最適なスレッド数やローカルメモリ量が異なっており，この研究ではこれらの自動最適化を提案・評価している[9]．特に，GPU 用に設計されたプログラムはしばしば CPU では性能が出ないことは本研究の評価結果からも明らかで，この研究ではこうした問題も取り扱っている．こうした技術を本研究の対象領域と組み合わせることで，1 つのコードだけでデータストリーム処理の負荷やノードの状況に応じた最適化を行う機構が実現できる．

5.3 GPU による IP パケット処理

PacketShader[10]という，ソフトウェアルータの GPU 実装を構築・評価したという研究．高バンド幅と高並列を要求するルーティング処理を，GPU にオフロードすることによって CPU のみを利用する場合よりも遥かに高い処理性能を実現している．応答性と並列処理性能の両立が必要な分野という意味で，本研究が対象とするアプリケーション領域に含まれると言える．

6. 結論と今後の展望

本研究では，既存の CPU や外付け GPU とは異なる性能特性を持つ APU のデータストリーム処理への適用を目的としマイクロベンチマークと実アプリケーションによって，スループットと応答性能の予備的評価を行ったが，現状の APU 環境では当初想定したような性能特性が望めないことが明らかになった．しかしながら，今後の SDK の改良やアーキテクチャの更新により，メモリアクセスやカーネル起動のレイテンシは改善される余地がある．行列積の結果からも，演算性能自体は同クラスの CPU を凌駕しているので，ドライバや実行環境の改善が待ち望まれる．

本評価で明らかになった APU の各種レイテンシが Tesla 以下に短縮された際には，データストリーム処理をはじめとする即時性と処理性能の両立が重要な分野での APU が活用されていくだろう．

今後の展望として，今回実験に用いたものよりも新しい世代の APU での実験や，どのような入力データレートや計算負荷ならば APU や GPU を CPU の代わりにデータストリーム処理に用いることができるかを評価するための問題定式化，こうした定式化を用いたヘテロジニアス環境下でのデータストリーム処理の動的最適化機構の実装と評価などが挙げられる．

参考文献

1) Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, MyungCheol Doo, and Kun lung Wu. “SPADE: the System S Declarative Stream Processing Engine”, *SIGMOD*, 2008.
2) Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Yoonho Park, and Chitra Venkatramani. “SPC: A distributed, scalable platform for data mining”, *DM-SSP*, 2006
3) Jeffrey Dean and Sanjay Ghemawat, “MapReduce: Simplified Data Processing on Large Cluster”, *OSDI*, 2004
4) 森田 康介，鈴村 豊太郎，「データストリーム処理を用いた変化点検知アルゴリズム SST の GPU による性能最適化」，*DE 研究会*，2010
5) 上野 晃司，鈴村 豊太郎，「データストリーム処理における GPU タスク並列を用いたスケーラブルな異常検知機構の実現」，*インターネットコンファレンス*，2010
6) Scott Schneidert, Henrique Andrade, Bugra Gedik, Kun-Lung Wu, and Dimitrios S. Nikolopoulos, "Evaluation of streaming aggregation on parallel hardware architectures", *DEBS*, 2010
7) 井手 剛，井上 恵介，「非線形変換を利用した時系列データからの知識発見」，*データマイニングワークショップ*，2004
8) 井手 剛，「行列の圧縮による変化点検出の高速化」，*IBIS*，2006
9) 滝沢 寛之，「複合型計算システムにおける実行時自動チューニング」，*ATRG*, 2010
10) Sangjin Han, Keon Jang, KyoungSoo Park and Sue Moon, “PacketShader: a GPU-Accelerated Software Router”, *SIGCOMM*, 2010

ⓒ 2011 Information Processing Society of Japan