2011 年フランス、シャモニーでのコード生成と最適化に関する国際シンポジウムで公開

# インテル® Array Building Blocks:

## 再構成可能なターゲット環境、ダイナミック・コンパイラーおよび組み込み言語

Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu,Dan Zhang

インテル コーポレーション
ソフトウェア & サービス統括部
パフォーマンス & プロダクティビティー・ライブラリー

我々が構築できる大規模なハードウェアの並列性は、それらをプログラムするソフトウェア開発者の能力を超えています。これは、我々の業界に蔓延する問題です。世界中のソフトウェア開発者の多くは、並列プログラミングのエキスパートではなく、彼らのためにコアとベクトル並列によるアプリケーションのコーディング、移植そしてデバッグを容易にすることは、マルチコアやメニーコア・プロセッサー・アーキテクチャーを活用するには不可欠です。しかし、ハードウェアのアーキテクチャーと ISA は、急速に多角化しており、1 つのバイナリーをすべての実行可能なターゲットで効率良く動作させることは困難です。そのため、再構築と動的コンパイルが重要になっています。

この文書では、再構築可能な動的コンパイル・フレームワークを提供するインテル® Array Building Blocks (インテル® ArBB) について紹介します。このシステムでは、マルチコアと異種メニーコアの両方でデータとスレッドの並列性による恩恵を得ることができるよう、標準 C++ の仕様範囲でプログラムを容易に記述そして移植することに注目しています。ArBB は、プログラマーの生産性と最良のパフォーマンスの両方のソリューションに関し顧客の要望に応えるため、他のプログラミング・モデルと相互利用可能です。これは言語仕様、コンパイラーのアーキテクチャー、コード変換そして最適化に貢献します。ArBB の現在のベータリリースから性能データを紹介し、いくつかの主要な分析における影響を示すことで、我々の顧客にとって興味深い多くのベンチマークに向けた変換と最適化を可能にします。

## I. はじめに

並列性は少なくても 20 年の間、プロセッサー・アーキテクチャーの主流となります。プロセッサーの設計者の観点から、増加する並列性 (機能ユニット、メモリーシステム、そしてコア数) は、性能を向上するための安価なメカニズムです。しかし、伝統的な逐次プログラムからマイクロアーキテクチャーが自動的に並列処理をディスパッチすることは、電力の制限と設計の複雑さに制限され、収益逓減の上限に達しました。

ソフトウェアの並列性に注意が向けられるようになったことにより、プロセッサー設計者からソフトウェア開発者へ責任が移行する結果となっています。この傾向は、ソフトウェアにおける明らかな変化をもたらします:

- ベクトル命令が豊富にそしてより広くなることで、SIMD ベクトルの並列性が明示的に (SSE、AVX、インテル® Many-Integrated Core Architecture (MIC) [Skaugen 2010]、Altivec) もしくは、暗黙のうちに行われます(GPUs [Buck 2007, Owens 2005]);
- コアにおける同時マルチスレッド技術の利用 (ハイパースレッディング、MIC、Sun Niagara);
- 単一ダイ上のマルチコア・アーキテクチャーの利用、マルチソケット・システムへの構成可能 (Pentium D、Sun Niagara、Core 2、Core i*、その他);

- 分化されたメモリー、ISA、そして性能上の特徴（Sandy Bridge、AMD Fusion）による異種計算エンジンの利用

これらのアーキテクチャーの傾向は、ハードウェアを有効利用するため、ソフトウェアが並列性を明確に管理することを要求します。しかしそれは非常に困難です。ハイパフォーマンス・コンピューティング (HPC) の技術者は、それらのメカニズムを直接利用する低レベルな並列プログラミングに精通しています。しかし他の多くの開発者は、これらのメカニズムを利用する知識（と設備）が限られています。並列プログラミングでは、独立した計算を隔離する必要があるため、複雑になります。カプセル化などのソフトウェア工学上のメカニズムの必要性、関数インターフェイス向けの最小限の隔離を提供しますが、それらは並列性が備える完全性（正当性）とは同等ではありません。並列性のバグは不完全な分離（つまりデータ競合）に起因し、生産性と製品品質に関わる新たな問題です。それらは、特定のコンテキストが利用されるまでは隠れていて、再現することも困難です。

演算リソースが安価になるに従って、生産性はより重要になります。モジュール化と構成の容易性は、より優れたソフトウェア工学における不可欠な原則です。しかし、多くのプログラミング・モデルでは容易に構成できません。最近のソフトウェア開発ツールは、遅延バインディングによるモジュール化インターフェイスをサポートします（例えば、C++ の継承や仮想関数、動的クラスロード、スクリプト言語におけるダイナミック型、その他）。遅延バインディングはソフトウェア・コンポーネントの柔軟性と表現性を高めます。遅延バインディングはコンパイラーにおける最適化の労力を軽減します。並列化に関するもっとも注目される言語開発 [Buck 2007, Nickolls 2008, Munshi 2008] では、最初に並列カーネルをインライン化することが要求されます。これは、OpenMP のような既存の並列プログラミング・モデルから進歩するものではありません。

並列性によりパフォーマンスが向上することと引き換えに、ソフトウェアは構成が容易でなく、難解で保守を難しくし、ときに移植性を損ないます。これらの観点から、主流なソフトウェア開発における並列性の欠如は、驚くべきものではありません。私たちの経験上、大部分の主流なソフトウェア開発者は、パフォーマンス向上よりも、開発コスト/リスク/時間の軽減に注力します。

インテル® Array Building Blocks (ArBB) の設計は、幾つかの要素によって動機づけられました。将来にわたって利用でき、移植性があるプログラミング・モデルは、ハードウェア・アーキテクチャーとその抽象化の両方に対応できなければいけません。移植性は、コードベースが時間の経過につれ堅牢なものになることを確実にする重大な要素であり、時間とともに生産性が高まり、維持費が下がらなければいけません。また、ソフトウェア工学の習得ができるよう、プログラミング・モデルは、性能向上を放棄することなくモジュール方式をサポートするべきです。

ArBB は、幾つかの主要な技術を統合することでこれらの目標を達成します。第 1 に、ArBB はダイナミック・コード生成に基づくジェネリック・プログラミングをサポートします。これにより、プログラミング・モデルを抽象化したまま、ターゲット・アーキテクチャーに対応したコード生成が可能になります。第 2 に、データ並列にもとづく構造化プログラミング・モデルは、構造上データ競合やデッドロックのような一般的な並列プログラムのエラーを排除することで生産性を高めます。データ並列は、しばしばタスク並列よりスケールします。第 3 に、最適化は ArBB に包括されるため、ソースコードから離れた状態で記述された計算は、演算主体のカーネルに融合することができます。これは、手動でのインライン展開や平坦化を行うことなく高いパフォーマンスを達成しながら、モジュール化とジェネリック・プログラミングをサポートします。

ArBB は、最近インテル [ArBB 2011] としてベータ版がリリースされ、インテル® Parallel Bulding Blocks のコンポーネントの 1 つとなっています。ArBB のインターフェイス設計とコンパイラー・アーキテクチャーは、インテル・ラボで開発された Ct プロジェクトと、カナダの Waterloo 大学で開発された商用 RapidMind プラットフォームが統合されたものです。従って、ArBB は数年に渡る顧客との相互作用と RapidMind における生産経験の特徴を含んでいます。ArBB は、簡単な高レベルの記述より、複

数の並列メカニズムによるコア、スレッド、ベクトル（SIMD）そして、命令レベルの並列性を対象とする統一されたフレームワークを提供します。ArBB プログラミング・モデルでは、同期は暗黙的であり、すべての演算結果は、単一で、決定性があり、シリアルの順番と一貫性があります。この決定論と連続した一貫性は、保守、テスト、そしてデバッグを容易にします。メモリーモデルも分離を前提としており、共有および分散の両方のメモリーモデルをサポートします。ArBB アーキテクチャーは、ダイナミック・インライン展開など斬新な機能をサポートし、並列性により劇的な性能向上を実現します。また、ArBB はコード生成プロセスにおける明示的な操作を可能にするため、コードのオブジェクト化（クロージャー）による表現を可能にします。JIT コード生成自体とそのインターフェイスは、JIT コンパイルの予測タイミングと制御を許すよう設計されています。

ArBB の言語機能と組み込みインターフェイスの説明を始めましょう。次にコンパイル・システムとそのアーキテクチャー説明し、そしてコンパイラーによる最適化の幾つかを紹介します。私たちは、ArBB の主要な言語機能とターゲット・アーキテクチャーに密接した最適化に注目します。最後に、コンパイラーの総合的な性能データを示し、最適化のパフォーマンスへの影響について議論します。

## II. ARBB 組み込み言語

ArBB は、構造化された決定論的なデータ並列プログラミング・モデルをサポートします。それは、組み込まれて、コンパイルされた言語として、API 構文として実装されます。実際にはまさしくライブラリーのように利用され、配備されますが、言語であると考えることができます。

ArBB 型を定義するため、標準 C++ のメカニズムが利用されます。ArBB の型は、スカラー値（整数値や浮動小数点値）とそれらの集合の両方を含みます。そしてプログラマーは、通常の C++ 形式でそれらの型の一連の操作を記述できます。しかし、ビルトイン C++ 型とは異なり、ArBB 型の一連の操作は”キャプチャー”でき、動的にさまざまなターゲットシステム向けのベクトル化されたマシン語に翻訳されます。さらに、ArBB は収集変数間の割り当てに値渡しによるセマンティクスを利用します。収集にまたがる表現式は、常に新しい収集が作成され、全体の入力収集がコピーされるように振る舞います。これはシステムの利用を簡素化します;例えば、関数から浮動小数点値がリターンされるように、収集した結果を返すことができます。さらに重要なことは、収集にはポインターではなく値のセマンティクスがあるため、エイリアシングを排除します。実際には、最適化によりほとんどのコピーは削除されます。

ArBB C++ API は、仮想マシン（VM）[Du Toit 2010] 上に構築され、将来代替えのフロントエンドとして利用可能な “ extern C” API を持っています。ArBB VM は、ダイナミック・ライブラリーとして分離され、演算と並列化を実装します。ArBB VM のセマンティクスは、抽象化された並列性に基づいています。ベクトル演算のシーケンスと要素関数の適用による収集により、多くの並列性が隠されています。要素関数はすべての収集要素にマッピングされます。ベクトル演算は収集を直接操作します。

VM は、抽象化され隠匿された並列性の表現式を、API を通して物理マシンの並列メカニズムにマッピングすることに責任があります。現在 ArBB は、ハードウェアに実装される SIMD 命令（ベクトル化）とコア（スレッド）の両方をサポートします。また、それはストリーミング、プリフェッチそしてパイプラインのレイテンシーを隠匿するため、潜在する並列性を利用することがあります。コアの物理的なベクトル幅と数は、VM によって抽象化されます。そのため、ベクトルマシン言語はダイナミックに生成され、ArBB で記述されたコードは複数のベクトル ISA (SSEn、AVX、MIC) にまたがってポータビリティーが備わっています。再コンパイルすることなく利用可能なコア数にスケールします。AVX における scatter/gather のような特殊な命令もサポートされます。また、ArBB の VM は、透過的にリモートメモリーをサポートするメモリーモデルを実装します。これは不要なコミュニケーション・コストを自動的に避けるため、外付けのコプロセッサー（MIC アーキテクチャーなど）のように扱うことができます。

ArBB 言語は、2 つの方法で生産性を改善します。1 つは、隠れた並列性、メモリーの局所性、そして

構造化された演算パターンに焦点を合わせる簡略化されたプログラミング・モデルです [Siu 1996, Skillicorn 1998, Bosch 1998, Bromling 2002, Aldinucci 2007, McCool 2010]。これはソフトウェア開発者のタスクを簡素化します。データ競合やデッドロックといった一般的なソースコードの問題を、決定性のあるパターンを利用することで回避できます。ソースコードは短く、そして問題の数学的表現に近くなり、保守を容易にします。同時に、パフォーマンス上重要な要素である、並列性と局所性を簡潔に表現できます。2 つ目は、コードに移植性があるということです。アプリケーションを新しいプロセッサー (大幅に異なったアーキテクチャーでも) に移植する際、基本的に労力は必要ありません。これは生産性の向上に貢献します。

ArBB C++ インターフェイスの詳細は、サンプルを利用して説明します。リスト 1 から 3 は、マンデルブロ集合を計算する 2 つの実装を示しています。最初は、要素関数を利用したもので、2番目はベクトル演算のシーケンスです。2 つの実装は等価です。リスト 1 は、マンデルブロ集合の 1 ピクセルの出力を、要素関数で演算しています。**要素関数**は概念的にはスカラー演算を行いますが、収集の各要素に適用できます。各要素を個別に計算することで、コアと SIMD レーンにまたがって並列に処理できます。リスト 2 は、要素関数のセットアップと起動を示しています。

```
int max_count; // C++ 非ローカル;値渡し
std::complex<f32> offset; // ArBB 非ローカル;参照渡し
void mandel(
  i32& d,                  // 出力要素
  std::complex<f32> c      // 入力要素
) {
  i32 i;
  c += offset;
  std::complex<f32> z = 0.0f;
  _for (i = 0, i < max_count, i++) {
    _if (abs(z) >= 2.0f) {
    _break;
    } _end_if;
    z = z*z + c;
  } _end_for;
  d = i;
}
```

*リスト1： マンデルブロ集合を要素関数で演算*

*C++ 非局所変数は、ArBB 関数が最初にコンパイルされるとき、値渡しでバインドされます。例えば、キャプチャーされるとき (以下を参照) や、map や call で最初に呼び出される時。ArBB 非局所変数は、参照渡しでバインドされるか、動的に更新されるかもしれません。複素数は C++ の標準テンプレートを利用して表現できます。一般的に ArBB では、多重定義を含む任意のユーザー型がサポートされます。*

```
void mandelbrot1(
  dense<i32,2>& dest,    // 配列出力
  dense<f32> sR,  // 列スケール (入力)
  dense<f32> sC   // 行スケール (入力)
) {
  dense<f32,2> sR2 =
          repeat_col(sR, N_COLS);
  dense<f32,2> sC2 =
          repeat_row(sC, N_ROWS);
  dense<std::complex<f32>,2> tscale;
```

```
  tscale.set<0>(sR2);
  tscale.set<1>(sC2);
  mandel(dest, tscale);
}
void mandelbrot1_call(int* res_arbb) {
  dense<f32> sR;
  bind(sR, scale, N_ROWS);
  dense<f32> sC;
  bind(sC, scale, N_COLS);
  dense<i32,2> dest;
  bind(dest, res_arbb, N_COLS, N_ROWS);
  call(mandelbrot1)(dest, sR, sC);
}
```

*リスト2： 要素関数を呼び出すラッパーコード*

*map 操作は、要素関数を収集された各要素に適用します。呼び出し操作は、並列命令のシーケンスを起動します (1 つの map で構成される)。バインド関数は、ArBB データ空間との間でデータ転送 (in/out) を行う方法の1つです (もう一方はユーザーイテレータ)。*

リスト 3 は、同じ演算を行うもう一つの方法ですが、ここではベクトル命令によって直接収集を操作しています。2 つの方法によるパフォーマンスは、ソフトウェア開発者が適した方法を選択できるよう、同等でなければいけません。

```
void mandelbrot2(
  dense<i32,2>& dest,
  dense<f32> sR, dense<f32> sC
) {
  dense<f32,2> sR2 = repeat_col(sR, N_COLS);
  dense<f32,2> sC2 = repeat_row(sC, N_ROWS);
  dense<std::complex<f32>,2> tscale;
  tscale.set<0>(sR2);
  tscale.set<1>(sC2);
  dest = fill(i32(0), N_COLS, N_ROWS);
  dense<std::complex<f32>,2> z =
  fill(std::complex<f32>(0.f,0.f), N_COLS, N_ROWS);
i32 i;
  _for (i = 0, i < max_count, i++) {
    dense<boolean,2> done = (abs(z) < 2);
    dest = select(done, dest + 1, dest);
    z = select(done, z*z + tscale + offset, z);
  } _end_for;
}
void mandelbrot2_call(int* res_arbb) {
  dense<f32> sR;
  bind(sR, scale, N_ROWS);
  dense<f32> sC;
  bind(sC, scale, N_COLS);
  dense<i32,2> dest;
  bind(dest, res_arbb, N_COLS, N_ROWS);
  call(mandelbrot2)(dest, sR, sC);
}
```

*リスト3： ベクトル命令を利用したマンデルブロ集合の計算。ここでは map を呼び出す代わりに、ベクト*

*ル命令のシーケンスを直接実行しています。これらは早期の exit 最適化を含む要素関数の実装と同様に融合されます。*

リスト4 では、ArBB の先進的な機能を利用しています;クロージャーと呼ばれる、コンストラクトを明示し、コード・オブジェクトを操作する機能を備えています。呼び出し操作はいくつかのステップが一体となっています。最初に特定のポインターで呼び出されると、ArBB は命令のシーケンスをキャプチャーし、IR (中間コード) を作成します。そして ArBB がこれをベクトルマシン言語のシーケンスにコンパイルし、マシンコードをキャッシュしそれらを起動します (可能であれば、複数のコアで同時に)。2 回目以降の呼び出しは、同じ関数ポインターで起動され、ArBB はキャッシュされたマシン語を再利用することで、オーバーヘッドを軽減します。

```
typedef
void F(dense<i32,2>&, dense<f32>, dense<f32>);
  max_count = 100;
  closure<F> mandelA = capture(mandelbrot2);
  max_count = 1000;
  closure<F> mandelB = capture(mandelbrot2);
```

*リスト4：2 つの異なったベクトル実装をキャプチャーするコードによるマンデルブロ集合の例。C++ の非局所変数である max_count は、キャプチャーする間フリーズされます。しかし、異形を作成するための更新と複数回キャプチャーすることができます。いったん構築されると、クロージャーは関数のように呼び出すことができます。呼び出し操作は、実際にはクロージャーを返します。クロージャーはそれが構築される関数の型に基づいて型が決まることに注意してください。動的に型が決まるクロージャーも利用可能ですが、一般的に ArBB では静的に型がチェックされます。*

しかし、これらのステップは必要な時、個別に呼び出すことができ、適切でない場合にはキャッシュを防止することが可能です。キャプチャー API 呼び出しのみが (そして常に) キャプチャー操作を行い、キャプチャーされたコードシーケンスを示すクロージャーから呼び出されたオブジェクトを返します。このクロージャー・オブジェクトは、呼び出しやマップにおいて関数のように起動できます。すべてのキャプチャーは、C++ 非ローカル変数と異なったキャプチャー済みのクロージャーによる非 ArBB 制御フローの効果を“凍結”します。これは、ArBB の異形を生成するためメタプログラミング言語として C++ を利用することを許します。これにより、ジェネリック・プログラミングの強力な書式をサポートします。

ArBB によるメタプログラミング形式は、テンプレート形式のメタプログラミングとは異なり、より強力です。”テンプレート・メタプログラミング” は、幾つかのハイパフォーマンス C++ ライブラリーで利用されています。[Veldhuizen 1999, Abrahams 2004]. C++ におけるテンプレート・メタプログラミングでは、ユーザー定義のコードシーケンスをより効率良い形式に変換するため、テンプレートを書き直します。そのようなテンプレート・ライブラリーでは、テンプレートの書き換え規則が C++ のコンパイル時に C++ コードを操作する “ 偶然の関数型言語” として利用されます。しかしながら、テンプレート・メタプログラミングは、C++ のコンパイル時間を増やし、ライブラリーを複雑にします。これは、実行可能な最適化の手段を制限します。加えて、エンドユーザーはライブラリーのソースを参照しなければいけません。対照的に、ArBB ではランタイム時に実行コードを生成します。そして、C++ から生成されたコードを操作できます。生成メタプログラミングは [Herrington 2003, Czarnecki 2000] 、メタプログラミングの形式を指す用語です。

テンプレートは C++ API において ArBB に利用されますが、よりジェネリックで、引数を利用できるシステムを提供するため、一般目的に限られます。ArBB を利用したライブラリーの開発者は、ソースコードを提供する必要はありません。それらのライブラリーは、事前コンパイルされたバイナリー形式で提供します。そのような場合でも、ArBB を利用するライブラリー開発者は、関数ポインターベースのコールバック

およびユーザーのライブラリーによって提供された仮想関数のオーバーロードの両方の（動的な）インライン展開などによる強力な最適化を実装しできます。以降に幾つかのサンプルを示します。表 I に補足事項として、サンプルで利用される ArBB の操作をまとめます。詳細については、ArBB のドキュメント [ArBB 2011] を参照してください。

## III. コンパイラー・システムの概要

インテルでは、生産性を向上させることで、プログラマーが任意の言語を使ってできる限り自由にコーディングできるようになると考えています。そのため、さまざまなフロントエンドをサポートすることは我々の長期的なビジョンの 1 つです。例えば、Python* [Clyther 2010]、Microsoft* .NET、金融サービス業界で使用されている金融言語、シェーダー言語などのフロントエンド、そして、マルチコア、メニューコア、各種 ISA (SSE、AVX、MIC) を含む広範なターゲット・アーキテクチャーをサポートすることが重要です。

表 I Array Building Blocks キーワード

| キーワード | 説明 |
|---|---|
| repeat_row, repeat_col | 行または列を複製して 1D 配列を 2D 配列に拡張します。 |
| fill | すべての要素を同じ値で初期化した新しい配列を作成します。 |
| *collection*.set<*i*> | 構造体のコレクション（配列）の *i* 番目のコンポーネントを初期化します。 |
| bind | C 配列と ArBB のコレクションをバインドします。 |
| replace | コレクションの指定された要素を新しい値で更新し、更新したコレクションを返します。 |
| _for, _end_for, _while, _end_while, etc. | クロージャーにキャプチャー可能な制御フロー構造の種類です。通常の C 制御はクロージャーが構築される場合のみ実行されます（汎用プログラミングで役立ちます）。ArBB 型のキャプチャーした値は ArBB 制御フローでのみ利用できます。 |
| uncaptured<T>::type | 指定された ArBB 型に対応する C 型を探します。 |
| f32, f64, i32, u32 | 単精度/倍精度浮動小数点数、符号付き/符号なし 32 ビット整数に対応するスカラー ArBB 型です。 |
| dense<*T*,*D*> | 要素の型が *T* (単精度スカラー ArBB 型またはその構造体) で、次元 *D* の密配列を表します。コレクションのサイズはランタイムによって異なります。割り当てには値渡しによるセマンティクスを使用します。 |
| add_reduce | コレクションの最後の次元の全要素を足して 1 つの要素にまとめ、コレクションの次元を 1 つ減らします。1D 入力にはスカラー、2D 入力には 1D コレクション、3D 入力には 2D コレクションを返します。ほかの結合演算子に対する一般化もあります。 |
| add_scan, add_iscan, など | 並列前置操作（排他的および内包的）です。ほかの結合演算子に対する一般化もあります。 |
| pack, unpack | コレクションから要素を選択し、新しいコレクションに隣接するように配置します。または、その逆の操作です。 |
| select | ブール型のコレクションを基に、2 つの入力コレクションのどちらかから要素を取得して、新しいコレクションを生成します。 ?: 演算子のベクトルバージョンです。 |
| *collection*[u] | コレクションの位置 u からランダムに読み取ります（gather）。 |
| call | 一連の命令（通常は、並列命令）を呼び出します。引数には、C 関数（キャプチャー済み、コンパイル済み、およびキャッシュ済み）かクロージャーを指定できます。 |
| capture | C 関数を呼び出し、呼び出される一連の ArBB 操作を記録して |

| | |
|---|---|
| | closure を構築します。 |
| map | 形状が同じ 1 つ以上のコレクションの要素に対して関数を複製して並列命令を生成します。 |
| closure<*F*> | シグネチャー *F* のキャプチャー済み関数を表す型です。 |
| shift, neighbor | 指定された現在の位置からのオフセットにあるコレクションの要素を返します。 |

インテルでは、OpenCL や LLVM などの標準バックエンドもサポートしたいと考えています。これは、フロントエンド言語とバックエンド・ターゲットの爆発的組み合わせを伴う “M × N” 問題を引き起こします (図 5 を参照)。その対策として、ArBB VM へのオープン・インターフェイスを作成することで [Du Toit 2010]、共通のコンパイラー・インフラストラクチャーに対して、フロントエンドごとに個別のインターフェイスを提供し、さまざまなバックエンドの中から選択できるようにします。

前述のように、ArBB は実際にはライブラリーのように使用され、標準コンパイラーで動作します。そのため、ソースコードを変更したり、C++ コンパイラーを呼び出さなくても、ArBB 動的ライブラリーを更新して、配備するだけで新しいターゲットがサポートされます。

以下に ArBB のシステム動作の概要を示します。C++ コンパイラーは、ヘッダーファイルとテンプレートを使用して call や map などの組み込み言語関数、そして ArBB 型のコンストラクターや操作を、ArBB 動的ライブラリーのエントリーポイントへの呼び出しとして表現します。

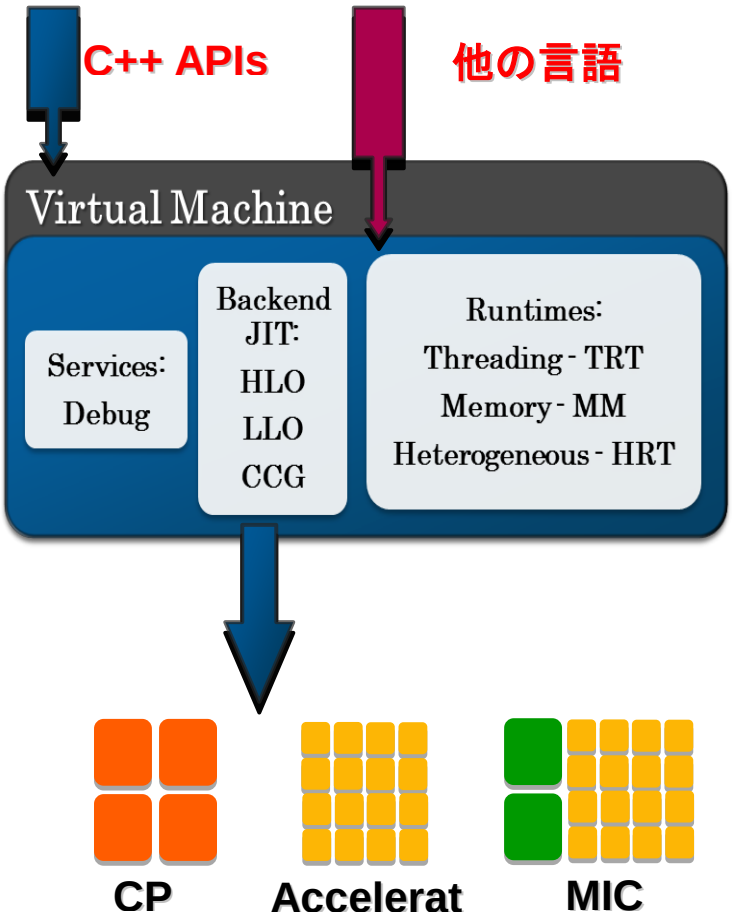



*図 1: ArBB アーキテクチャー。バックエンドは高レベルな最適化機構 (HLO)、低レベルな最適化機構 (LLO)、コンバージ・コード・ジェネレーター (CCG) からなります。スレッディング・ランタイム (TRT) はインテル® スレッディング・ビルディング・ブロックを使用して構築され、インテルおよびサードパーティー製*

*の並列化ツールとリソース・マネージャーを共有します。異種ランタイム（HRT）を使用してコードのロードと実行を調整し、1 つまたは複数の（リモートも含む）アクセラレーション・デバイスとの間でデータを移動します。ベクトル/スカラーメモリー空間のガーベジ・コレクションはメモリー・マネージャーによって行われます。*

call（または capture）が呼び出されると、ArBB ライブラリーは“キャプチャー・モード”になり、操作を指定する C++ 関数を呼び出します。キャプチャー・モードでは、ArBB 型に対して直ちに操作を実行する代わりに、ArBB 型に対する操作の呼び出しシーケンスのトレース（および ArBB 制御フローからのトークン）を記録し、計算を指定する内部表現（**IR**）の作成に使用します。これらの関数からリターンすると、ターゲットの JIT コードが生成され、再コンパイルすることなく再利用できるようになります。ArBB コードを含むネイティブ C/C++ コードは、最初のキャプチャーでのみ実行されます。

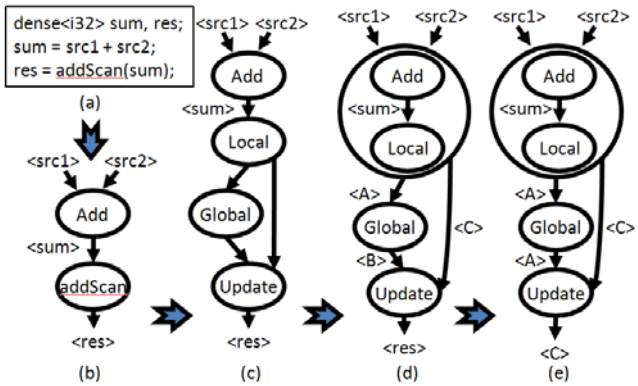
ネイティブコードと ArBB コードは異なる論理メモリー空間で処理を行うため、これらのコード間の依存関係は bind などの ArBB 演算子や読み取り/書き込み範囲イテレーターによってのみ発生します。データのコピーと同期は暗黙で行われるため、これらの操作のメカニズム、そして必要性は実装に委ねられます。これは CUDA [Nickolls 2008] にあるように、パフォーマンスを最適化し、生産性を高める上で重要です。

## IV コードの最適化

ArBB は、中間言語（IR）をビルドし最適化して、さまざまなレベルのパフォーマンスを向上します。高レベルな最適化機構（HLO）は、アーキテクチャーに依存しない最適化を実行し、メモリー使用、スレッディング・オーバーヘッド、冗長計算を減らし、データの局所性とアフィニティーを高めます。低レベルな最適化機構（LLO）は、HLO の IR を SIMD およびスレッドにより並列化し、プラットフォームに依存しないコードを生成します。そして、コンバージ・コード・ジェネレーター（CCG）は、ターゲット・プラットフォーム向けに最適化されたバイナリーを生成します。原則として、解析フェーズの一部はオフラインで行うことができますが、マシン固有のフェーズやランタイムデータに依存するフェーズは動的に行う必要があります [Nuzman 2011]。

### A. 高レベルな最適化機構

HLO には 40 種類の最適化があり、それらは 3 つのフェーズに分類できます。斜体で示されている最適化については、セクション VI.D で例を使って説明します。図 6 (a) は、HLO の IR からほかのフェーズの最適化された形式への変換方法を示した ArBB サンプルプログラムです。HLO は最初に、図 6 (b) で示すように高レベルなセマンティクスを保持した IR をビルドします。



```
Procedure decompose(graph g)
  for each node n in g
    if n is a hierarchy node
      decompose(n->subgraph);
    else
    decompose(n);

Procedure decompose(node n)
  if opclass(n)=element-wise
    output(n) = local(input(n))
  else if opclass(n)=reduce
    if (n is 2D or 3D) return;
    /* 次のコードシーケンスを生成 */
    temp = localReduce(input(n));
    output(n) = globalReduce(temp);
  else if opclass(n)=scan
    if (n is 2D or 3D) return;
    /* 次のコードシーケンスを生成 */
    <temp1,temp2> = localScan(input(n));
    temp3 = globalScan(temp2);
    output(n) = updateScan(temp1,temp3);
  else if opclass(n)=permute
    if opkind(n)=pack
      /* 次のコードシーケンスを生成 */
      temp1 = localPack(input(n));
      <output(n),temp2> = globalPack(temp1);
      output(n) = updatePack(input(n),temp2);
    else if opkind(n)= scatter
      /* 次のコードシーケンスを生成 */
      temp = localScatter(input(n));
      output(n) = globalScatter(temp);
```

*図 2: HLO 変換。(a) オリジナルコード (b) 高レベルな IR (c) サブプリミティブに分割 (d) 融合された IR (e) 非 SSA IR*

```
  else … /* その他の permute の種類は省略 */
```

*リスト 3: サブプリミティブ分割アルゴリズム*

## 1. フェーズ 1

HLO のフェーズ 1 では、静的な単一割り当て形式 (SSA) の変換、コピーの伝播、不要なコードの排除、定数の畳み込みなどの典型的なコンパイラーによる最適化を実行します [Allen 2002]。HLO は IR を変換するときに 3 つの最適化を繰り返し実行します。

フェーズ 1 では ArBB 固有のいくつかの最適化を行います。*map の SIMD 化*は、スカラー形式で記述された map 関数をベクトル化された call 関数に変換してデータ並列を活用します。

サブプリミティブ分割は、高レベルなプリミティブ IR をサブプリミティブと呼ばれる 3 種類の低レベルな IR (*すなわち*ローカル、グローバル、更新) に分割します。ローカル・サブプリミティブは、ローカルタスク内の要素単位の操作を実行します。グローバル・サブプリミティブは、すべての並列タスク (*例*: リダク

ションやスキャン) のローカル・サブプリミティブの結果を収集し伝播するため、同期が必要になります。更新サブプリミティブは、グローバル・サブプリミティブの結果を使用して、各タスクの最終結果を計算します。

データ並列操作を関数サブプリミティブに分割することで、通信と同期のパターンが同じものをグループ化し、それらを一緒に実装できます。その結果、コンパイラーが自動で ArBB 演算子を結合できるようになります。高レベルなプリミティブがサブプリミティブに分解されると、高レベルなセマンティクスは保持されません。リスト 7 は、高レベルなアルゴリズムでサブプリミティブ分割を表したものです。関数に含まれるすべての ArBB プリミティブからなる評価ユニット (EU) のすべてのノードをスキャンします。操作クラスによって、高レベルなプリミティブは異なるサブプリミティブの集合に分割されます。図 6 (c) は、図 6 (b) のサブプリミティブ分割の結果です。例えば、'add' は要素単位の演算子でローカル・サブプリミティブにのみ分割でき、'addScan' はローカル、グローバル、更新サブプリミティブに分割できます。

## 2. フェーズ 2

HLO のフェーズ 2 では、低レベルなサブプリミティブの最適化を実行します。これには、*融合*、*形状の融合*、*異なる型の融合*、並列ループ解析、マスクの融合、シフトの融合、ブールの軽減が含まれます。

融合は ArBB において最も重要なパフォーマンス最適化手法の 1 つです。すべての ArBB 演算子はデータ並列であるため、ベクトルオペランドの各要素をシーケンシャルまたは並列に反復するには、1 つまたは複数のループあるいはループの入れ子が必要です。このようなループを*暗黙の*ループと呼びます。融合は複数の演算子をグループ化し、ループの入れ子として実装できるようにします。融合の主な目的は、メモリー使用に加えて、同期と並列化のオーバーヘッドを減らすことです。いくつかの ArBB 演算子を融合することで、すべての演算子でメモリーから中間表現を読み取り/書き込む必要がなくなり、代わりに中間表現を SIMD レジスターに格納して、次の演算子にパイプラインを介して送ります。さらに、データ並列タスクは、融合したほうがタスクの粒度が粗くなり、スケールしやすくなります。

現時点では、並列タスクの実装を容易にするため、融合可能な演算子には依存関係がなく、スレッド間の同期を必要としないものでなければなりません。デフォルトでは、要素数が同じベクトルに対する演算子のみ融合します。融合された ArBB 操作のリストを '融合ノード' といいます。形状解析を実装することで、融合ノードにある暗黙のループを収集し、データの局所性が向上するようにループの順序を最適化します。多次元ベクトルの形状の代わりに、ArBB 演算子の暗黙の形状を解析します。例えば、要素単位の演算子はオペランドの次元数に関係なくすべて 1D で、shift 演算子は行方向、列方向ともに 2D です。融合ノードのすべての形状は 1 つに統合され、ヒューリスティックを使用して、その特定の次元ごとに並列化、SIMD 化、ブロッキングの決定と適用が行われます。

ArBB の形状解析では、従来のループの最適化の多くを行うことができます。

リスト 8 に ArBB の融合アルゴリズムを示します。**EU** は JIT 評価ユニットで、ArBB 演算子の集合を示す graph IR メンバーを持ちます。最初に、各ノードの融合関連データの構造体を初期化します。要素サイズの互換性の確認、ループ構造の実装を表す入出力融合シンボルの初期化、融合の型 (形状) の確認を行います。次に、IR をリニアスキャンして、貪欲法で融合可能なコード領域を再帰的に探し、融合マップを作成します。制御フローバリアによって融合が妨げられるまで、融合の形状も保持します。融合マップの解析が終わったら、階層ノードを作成し、その下にすべての融合可能なノードを配置します。

図 6 (c) に融合を適用した結果が図 6 (d) です。'add' ノードと 'local' ノードはともに要素単位の演算子であるため融合することができます。しかし、'global' は同期を必要とするため、同じ融合ノードに含めることができません。

*異なる型の融合* 変換は、次のように要素数が同じで要素型の幅が異なる操作を融合します。SIMD レジスターにパックされた単精度演算と倍精度演算を融合する例について考えてみましょう。コレクションの

形状が同じ場合のみ（要素数が同じであるため）融合できます。また、すべての処理が 1 つのループに融合されるため、単精度と倍精度の反復回数は同じでなければなりません。倍精度値は幅が単精度値の 2 倍で、SIMD レジスターごとの要素数が単精度値の半分なので、倍精度ループを 2 倍でアンロールして、単精度ループと組み合わせます。

```
Procedure initFuseInfo(graph g)
  for each node n in g
    if n is a hierarchy node
      initFuseInfo(n->subgraph);
    else
      setElemSize(n);
      setFuseSymbol(n);
      setFuseType(n);
Procedure collectFuseInfo(graph g)
  for each node n in g
    if n is a controlFlowBarrier
      stopAllFusionResults(n);
    else if fusable(n, fuseShape)
      updateFuseShape(fuseShape, n);
      insertInFusionMap(fuseMap, n);

Procedure transformFusion(graph g)
  for each fuse region r in fuseMap
    fnode = createFuseNode(r);
    setFuseSymbol(fnode, r->FuseSym);
    linkFuseNodeToGraph(fnode, g);

Procedure fuseGraphRecursive(graph g)
  for each node n in g
    if n is a hierarchy node
      fuseGraph(n->subgraph);
Procedure fuseGraph(graph g)
  collectFuseInfo(g);
  transformFusion(g);
  fuseGraphRecursive(g);


Procedure fuse (EU eu)
  initFuseInfo(eu->graph);
  fuseGraph(eu->graph);
```

*リスト 4: 高レベルな融合アルゴリズム*

並列ループ解析は、IR を検証してループ運搬の依存がないものを並列化可能としてマークします。この解析結果に基づき、後で LLO はデータ並列にするか、タスク並列にするかを決定します。この暗黙の並列性は、map 関数などのその他の表現から明示的な並列性を引き出すのにも使用されます。
マスクの融合による最適化は、同じマスクを持つ一連のマスク演算子を、最終結果を除いてアンマスク演算子に変換します。これにより、マスク演算子の処理に時間のかかるマイクロアーキテクチャーにおいてその使用が最小限に抑えられます。これは、マップ関数の変換後に IR を最適化する場合に特に役立ちます。マップ関数の変換により、ベクトル演算に含めるベクトル要素の選択と制御フローのエミュレートのため、マスク演算子が多数追加されるからです。

シフトの融合は、一連の shift 演算子を算術規則に従ってマージし最適化します。例えば、右へ 1 シフトと右へ 2 シフトを続けて行うのは、右へ 3 シフトを行うのと同じです。本質的に shift 演算子では並列タスク間に依存関係があるため、シフトの融合が有効でない限り一連の shift 演算子を融合することはできません。

ArBB のその他のデータ型とは異なり ArBB のブール型ではビット幅を指定しないため、ブールの軽減はブール型をそれが定義されている演算と同じ幅の内部ブール型に変換します。これにより、一義化によるキャスト演算子も最小限に抑えられます。

## 3. フェーズ 3

HLO のフェーズ 3 では、主にメモリーの最適化とループの順序解析を行います。HLO のフェーズ 1 と 2 は、IR を SSA 形式で保持します。メモリーの最適化は、サイズが同じでその有効範囲がオーバーラップしない限り、異なるベクトルが同じメモリー空間を共有できるようにします。これにより、合計メモリー使用量は減りますが、IR は非 SSA 形式になります。図 6 (e) はメモリーを最適化した結果です。コレクション <A> と <B>、コレクション <C> と <res> はそれぞれ同じメモリー空間を共有できます。

ループの順序解析は、_for、_while、_do などの明示的なループと融合ノードにある暗黙のループそれぞれのデータ再利用の機会を解析し、データの局所性を向上することで、これらのループの最適な順序を決定します。

## B. LLO

LLO は並列化と SIMD 化の両方を行います。関数をアウトライン化することで並列性を実装します。アウトライン化したタスクは、引数のマーシャリング解除後に、それぞれに割り当てられた処理を取得します。これは、OpenMP* の実装と同じです [Brunschen 2000]。バックグラウンドでスレッディング・ランタイム (TRT) が呼び出され、アウトライン化されたタスクがスポーンされます。

現在 ArBB では、データ並列性と「並列 for」形式のタスク並列性の 2 種類の並列性をサポートしています。そのため、セクション IV.A.2 の説明のように、融合ノードの共通の形状を分割するか、反復空間を分割します。並列領域のサイズを増やすことで並列化のオーバーヘッドを分散できるため、タスク並列性は _for が融合ノードを囲んでいる場合にのみ使用します (その逆の場合には使用しません)。並列 _for ループ内にある融合ノードの入出力引数は、_for 構造外で無効な場合、タスクに対してプライベートとなります。タスク・プライベート変数はローカル・ストレージで管理されます。また、LLO はアライメントの有無にかかわらずデータ分割方法を選択します。多くの演算子では、ベクトルを分割した後もアライメントされているベクトル部分にアクセスできるように、各タスクにアライメントされたデータを渡す必要があります。しかし、グローバルな集約とスキャンでは、各タスクは 1 つのローカルスカラーの合計を受け取り/引き渡すため、分割方法でアライメントを考慮する必要はありません。LLO は、タスクの粒度を管理し、スケーラビリティーの問題に対応します。主に次の 2 種類の同期を行います。

- pack、reduce、*scan* などの一部の演算子のグローバルフェーズ [ArBB 2011] では、ローカルの結果をマージするのに同期ポイントが必要です。
- コンパイラーは、細粒度でのタスク間の依存性の追跡をサポートし、メリットがある場合にはバリアの代わりに依存関係にあるタスク間でのみ同期の使用を有効にします。

ArBB の SIMD 化プロセスには、効率の良い SIMD アルゴリズムの実装、剰余処理、アライメント

処理、スケーラビリティー・バランスの保持が含まれます [Zhao 2005]。LLO は各ローカル、グローバル、更新サブプリミティブに対して効率の良いアルゴリズムを実装します。剰余処理は、ベクトルレジスターに格納されないデータセットの境界要素を処理します。剰余処理には 2 通りの方法があります: (1) マスクを使用して有効な要素からの結果のみ保持する。(2) スカラーコードを生成してスカラーループ内で各要素を個別に処理する。ソフトウェアを再利用するために、ほとんどの場合は 1 つ目の方法を採用します。SSE/SSE2 命令ではアライメント処理が必要です。LLO は、融合ノードの入力引数と出力引数の両方のアライメントの開始メモリーアドレス (不変) を保持します。残りのデータはほぼすべてレジスターに格納されるため、順列操作を除く多くの操作ではアライメントの必要はありません。

LLO はステンシル・アプリケーションでよく見られる shift 操作を最適化します。通常、shift 操作はアライメントされていないアドレスから読み取ります。SIMD 化のメリットを最大限に活用するためには、アライメントされたアクセスの SIMD ループをピーリングして、アライメントされていないアクセスの SIMD ループはそのまま残します。コンパイル時に shift 操作のシフト距離が分からない場合は、ループピーリングするべき場所を特定できないため、その shift 操作をほかの shift 操作と融合することはできません。ただし、シフト距離としてループ帰納変数が使用されている場合は、シフト方向を解析し最大シフト量を計算することでアライメントされたアクセスの SIMD ループがピーリングできるようになり、shift と _for ループを融合することができます。

## V. 関連情報

ArBB で使用されているデータ並列プログラミング・モデルは、APL [APL] や C [Hillis 1986] をはじめとする以前あったいくつかの言語に似ています。ArBB は命令型言語ですが、セグメント (入れ子された) 配列 [Blelloch 1990, 1993, 1996] (ここでは詳細は省略) などの主な機能のいくつかは NESL 関数型データ並列言語にインスピレーションを受けています。

[Chatterjee 1993] は、VCODE と呼ぶスタックベースの中間言語 (IR) に基づくデータ並列性について検証しています。この論文で使用されている IR には、ArBB と多くの類似点があります。どちらのシステムも、入れ子したデータ並列タイプと順列やスキャンを含む並列演算子をサポートしています。そして、彼らの中間表現もグラフ表記で、我々のものに似ています。ArBB の解析と変換には、サイズの推測、アクセスの推測、クラスター分割、ストレージの最適化など、彼らの手法のほとんどが含まれています。ただし、いくつかの点で [Chatterjee 1993] とは異なります。VCODE [Chatterjee 1993] とは違い 、ArBB はクロージャーなどの関数型言語の機能を取り入れつつ、命令セマンティクスを採用し、C/C++ などの主流な言語を拡張します。また、ArBB は_for や _if などの命令制御フロー構造がある場合でも ArBB 演算子を融合できます。そのため、さまざまな粒度で融合を行うことができ、以下に示すような優れたハイパフォーマンスを引き出すことができます。ArBB は、タスク並列性とデータ並列性 (スレッディングと SIMD) をすべて 1 つの統合されたフレームワークで実現します。そのため、タスク並列性、データ並列性、またはその組み合わせの中から最も効果的なものを選択することができます。

入れ子されたデータ並列性は Haskell [Chakravarty 2001] でもサポートされています。パターンベースの並列プログラミングの強い影響についてはすでに紹介しましたが [Cole 1989, Gamma 1994, Siu 1996, Skillicorn 1998, Bosch 1998, Massingill 1999, Bromling 2002, Tan 2003, Cole 2004, MacDonald 2002, Mattson 2004, Aldinucci 2007, McCool 2010]、典型的な並列ワークロードで使用されるパターンに関するカリフォルニア大学バークレー校の研究にも大きな影響を受けました [Asanovic 2006]。ArBB の先行システムとして、RapidMind [McCool 2006]、そして RapidMind の先行システムである Sh [McCool 2002, 2004a, 2004b] と Ct [Ghuloum 2007] があげられます。最近の Python のコード特殊化を使用した、生産性に優れたデータ並列言語への取り組みでも類似点が見られます [Catanzaro 2010a, Catanzaro 2010b]。言語機能としてのユーザー定義によるコード生成のサポートは ML [Lee 1996]、テンプレート・メタプログラミング [Veldhuizen 1999, Abrahams 2004]、および生成的メタプログラミング [Herrington 2003] から

ヒントを得ました。

## VI. 性能データ

ここでは、性能データを基に次の 3 つの点を実証します。

- ArBB は高品質なコードを生成し、パフォーマンスも優れています。
- SIMD とスレッドによる並列性を利用して大幅なスピードアップを達成します。
- ArBB の最適化により、適用したコード領域でパフォーマンスが大幅に向上します (これは、ArBB プログラミング・モデルにとって最も重要なことです)。

最初の 2 つは ArBB コードと C コードを使用して、ArBB の異なる最適化レベルでのスピードアップを比較します (セクション VI.C)。最適化の影響はセクション VI.D で説明します。

### A. ワークロード

この解析では、インテル® ArBB ベータ版に含まれる 17 個のワークロードを使用しました [ArBB 2011]。使用したワークロードの詳細は表 II を参照してください。これらのワークロードは異なるアプリケーション・ドメインにわたっており、データ並列性の活用に関するお客様からの意見と要件を基に選択しました。

### B. 解析方法

それぞれのサンプルのカーネルは、(スカラー) C と ArBB の 1 つ以上の表現 (カーネル) を使ってコーディングしています。カーネルの実行時間は、ウォームアップ実行後 (コンパイルとキャッシュのウォームアップが終わった後) に測定しました。インテル® ArBB ベータ版には小さなデータセットと大きなデータセットがありますが、ここでは大きなデータセットを使用しました。4 コア、2 スレッド、3.30 GHz、32GB メモリー、8MB キャッシュの Nehalem ソケットを 1 つまたは 2 つ使用し (以下を参照)、スイッチの比較は RedHat* EL 6.0 Beta 1 で、全体のスピードアップの測定は Windows Server* 2008 SP2 で行いました。すべてのワークロードは 64 ビットでコンパイルしています。

スピードアップは、ベースラインと比較対象カーネルを 10 回ずつ実行し、それぞれの最短時間を基に計算しました。いくつかの外れ値を除いて、ベースラインでは実行間の差異は 2% 以内でした。測定された結果はすべて統計的に有意なものでした。タスクを 1 ハードウェア・スレッドにつき 4 タスクに分割したときに実行間の差異が最小になったため、この粒度を使用しました。以上が、スイッチ設定の比較に使用した構成です。全体のスピードアップの比較には、より一般的な 2 ソケット、4 コア、2 スレッド、1 スレッドにつき 1 タスクという構成を使用しました。

表 II: ARBB ベータ 3 に含まれるサンプルのワークロード

| | | |
|---|---|---|
| 融資 | binomial-tree | ヨーロピアン・オプション価格の数値格子 |
| | black-scholes | ヨーロピアン・オプション価格の解析法。オプションで多項式の評価と近似を行う。 |
| | monte-carlo | 不規則に変動する価格からブラックショールズ公式を使用して金融オプションを計算する確率統計的手法。オプションで乗法合同法 (MCG) を使用して一連の乱数を生成できる。 |
| | poisson-solver | モンテカルロ法によりポアソン関数の解を求める (MCP ソルバー)。線形合同数法 (LCG) で生成した一連の乱数を使用。 |

| | | |
|---|---|---|
| グラフィック | raytracing 1 | カメラを通して画像平面から光源までの光線を追跡し、リアルな画像を作成するカーネル。2D 配列内の各ピクセルに対して、光線-三角形交差の近似を求め、証明計算を使ってピクセルシェードの評価を行う。 |
| | raytracing 2 | raytracing1 の光線-三角形交差を、光線と交差するグリッドセルの三角形のみに制限したバージョン。つまり、一定の空間分割を使用することで高速化したもの。 |
| | mandelbrot | 二次多項式写像を使用したフラクタル・データ・セットの生成。 |
| イメージ処理 | convolve | 離散ガウス関数を使用した 2D イメージの畳み込み。 |
| | gauss-convolve | convolve に似ているが、ステンシルサイズが異なり、奇数のステンシルサイズを仮定しない。 |
| | sobel | 画像強度の勾配 (変化) を使用した 2D イメージのエッジ検出フィルター。 |
| 医療 | 3D-dilate | 3D グレースケール・イメージに適用される膨張のモルフォロジー処理。 |
| | 3D-erode | 3D グレースケール・イメージに適用される侵食のモルフォロジー処理。3D 膨張と似ているが、出力がわずかに異なる。 |
| | 3D-gauss-convolve | 離散ガウス関数を使用した 3D イメージの畳み込み。 |
| | back-projection | CAT スキャンからの入力データを使って画像再構成を行う手法。 |
| 地震 | 3dstencil | 逆時間移動 (RTM: Reverse Time Migration) で使用される畳み込み。 |
| | convolution | 地震画像の 1D/2D 畳み込み。 |
| | kirchhoff | 地表下の地震波の速度が一定であると仮定した一般的なキルヒホフ移動。 |

使用したスカラー・コンパイラーは、Microsoft* Visual C++* コンパイラー 2008 (VCC)、インテル® C/C++ コンパイラー 12.0 0 (ICC)、gcc* 4.4.3 (GCC) です。パフォーマンス関連のコンパイラー・スイッチは、我々の開発環境で使用しているものと同じで、合理的なビルド時間で優れたパフォーマンスが得られるものを使用しました。O2、-msse2、-fno-stack-protector (オーバーラン保護なし)、-fno-strict-aliasing (ANSI エイリアシング規則を無効にする)、-fp-speculation=fast (浮動小数点演算のスペキュレーションを行う)。

## C. 全体のパフォーマンス

ArBB には、独自の最適化レベル ARBB_OPT_LEVEL=O2 と O3 があります。ArBB の O2 は、SIMD 化を含むすべての最適化を適用しますが、シングルスレッドで実行します。O3 では、O2 に加えてスレッドによる並列性が追加されます。

表 III は、ArBB の最適化レベルを使用して、各種コンパイラーで生成したコードをシングルスレッドで実行した場合の C ベースラインに対するスピードアップの相乗平均をまとめたものです。ベースラインの C コードには、ベクトル化を促進するような変更は特に加えていません。図 9 は、Windows* 上で ICC および VCC コンパイラーを使用してコンパイルしたシングルスレッドの C ベースラインに対する、ArBB の O2 と O3 を使用したケースのスピードアップです。Linux* でも同様の結果が得られました。ワークロードに複数のカーネルがある場合は、スピードアップが最高の ArBB 実装のものを使用し、カーネルを区別するためにそれぞれ k1、k2 のようにラベルを付けました。ワークロードは図 9 のように、最も一般的なベースラインである VCC のスピードアップが昇順になるようにソートしています。一般に、このようにソートしたものを S 曲線と呼びます。

表 III: 各種コンパイラーにおけるシングルスレッドの C ベースラインに対する ArBB 最適化レベル 2 (O2) と 3 (O3、スレッディング) での相乗平均および最大スピードアップ。

| | | Windows* | | Linux* | |
|---|---|---|---|---|---|
| | | ICC | VCC | ICC | GCC |
| 相乗平均 | O2 | 1.4x | 2.0x | 1.7x | 2.1x |
| | O3 | 4.3x | 6.1x | 4.4x | 5.8x |
| 最大 | O2 | 8.5x | 16.9x | 9.9x | 10.9x |
| | O3 | 59.2x | 131.9x | 38.8x | 54.9x |

表 III のスピードアップの相乗平均は、O2 ではすべてのコンパイラーで 1 を大きく上回っています (1.4x-2.1x)。つまり、ArBB を使用したほうが SIMD ハードウェアをより有効活用していることが分かります。また、図 9 の S 曲線から、すべてのスピードアップが 1 を超えていることが分かります。通常 O3 のスピードアップ (8.5x-16.9x) は、メモリーバインドでない場合はスレッドによる並列性が効率良く使用され、直線的に増加します。例えば、ランダム収集を行う kirchhoff や条件文と隣接値へのアクセスを含む convolve において優れたスケーリングが見られます。本ベータリリースでは、ベクトルコピーを減らすことでメモリー・ボトルネックを軽減しスレッドのスケーリングを向上させるなどのいくつかの最適化がまだ実装されていません。 超線形スピードアップは、メモリーのトラフィックと (融合などに関連した) 制御オーバーヘッドを減らすことで達成できます。

## D. 最適化の影響

セクション IV で述べたように、ArBB の最適化のいくつかは、言語とコンパイラー・アーキテクチャーによっては興味深い結果が得られます。このセクションでは、最適化の一部を無効にし、スピードアップを測定して、パフォーマンスへの影響 (最適化を無効にした場合の時間/最適化を有効にした場合の時間) を検証します。検証結果のうち、ここで詳細に取り上げるものを図 10 に示します。また、右側にベータ版に含まれるサンプル全部の相乗平均と最大スピードアップを示します。さらに、説明を補足するために、使用したサンプルのソースコードも示します: mandelbrot k1 (リスト 3)、kirchhoff k2 (リスト 11)、sobel k2 (リスト 12)。

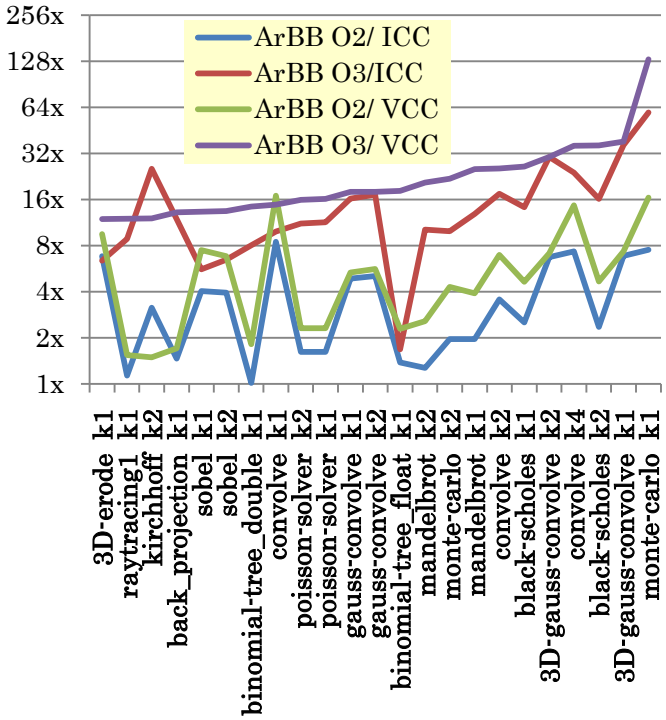



*図 5: ArBB 最適化レベル O2 (SIMD) とO3 (SIMD+スレッディング) を使ってインテルと Microsoft* の C/C++ コンパイラーでビルドしたケースのスピードアップ (目盛りは $\log_2$ 単位)。S 曲線は VCC の O3 スピードアップが昇順になるようにソートしています。*

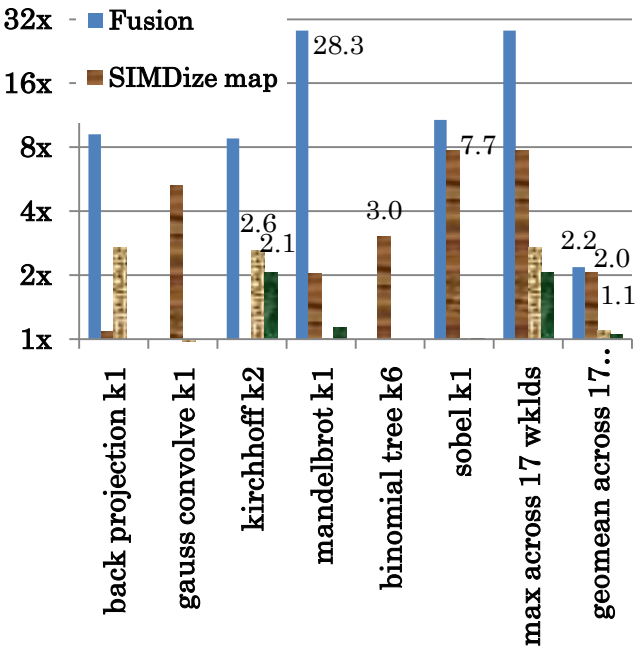



*図 6: 選択したワークロードとベータ版に含まれるサンプル全体における融合関連の最適化のスピードアップへの影響*

融合の最適化では、mandelbrot k1 (リスト 3 の ArBB コード) で 28.3x のスピードアップが見られます。すべての演算子を融合することで、*done*、*dest*、*z*z* などの一時変数を、ローカルメモリーではなく、ベクトルレジスターに格納します。また、_for ループの融合により、ループ終了後も有効なすべての出力ベクトルに対して、_for ループのそれぞれの反復でメモリーの割り当てと解放を行う代わりに、同

じメモリーハンドルを共有できるようになるため、パフォーマンスが大幅に向上します。さらに、局所性を向上するため、構造体の配列 (AoS) から配列の構造体 (SoA) への変換を std::complex の実部と虚部に適用します。

リスト 11 は kirchhoff サンプルコードです。一般的な ArBB 演算子だけでなく、_for ループ全体が融合されていることが分かります。*異なる型の融合*では、f32 型と usize (ターゲットにより符号なしの 32 ビットまたは 64 ビット) 型が混合する融合コード領域で 2.6x のスピードアップが得られました (図 10 を参照)。さらに、カーネルには、2 つの異なる形状 (1D と 2D) に対する計算が含まれています。形状の融合は kirchhoff に含まれる形状を変更する演算子を融合し、ループ全体の融合を可能にします。これにより、 2.1x のスピードアップが得られました (図 10 を参照)。形状の融合を行わなかった場合、それぞれの形状は別々のコード領域となり、融合コード領域は 2 つになっていたでしょう。

```
_for (jcx = (uncaptured<usize>::type)0,
      jcx < NX_LEN, jcx ++) {
  f32 cx = f32(jcx) * dx;
dense<usize,2> vIt = (dense<usize,2>)(sqrt((vX - cx) * (vX - cx) + vZ * vZ) *
reciVdt + 0.5f);
dense<usize,2> vIu = fill(jcx, nx, nt);
  dense< array<usize, 2>, 2> vIdx;
  vIdx.set<0>(vIt);
  vIdx.set<1>(vIu);
  vModl += vData[vIdx];
} _end_for;
```

*リスト 7: 地震の復元を行うための **kirchhoff** サンプルコード*

sobel サンプルコードはリスト 12 に示します。sobel k2 のベクトル化実装では、スカラー実装と比べて 7.7x のスピードアップを記録しました。map 関数のベクトル化実装では効率の良い 2D のベクトル演算子を使用しています。例えば、neighbor 演算子はメモリーからスカラー要素を 1 つだけロードするので、ベクトル演算ほど最適化されていない可能性があります。ArBB は、セクション IV.B で述べたように、neighbor 演算子を SIMD 化および効率良く最適化された shift2D 演算として内部的に実装します。一方、sobel カーネルの単純なスカラー実装では、入れ子されたループが 2 つ追加され、それぞれのベクトル要素に対するスカラー演算はそのまま残ります。

### E. コード・ジェネレーターでのループ解析の使用

ArBB は各種ターゲット・プラットフォームをサポートしているため、MIC アーキテクチャー向けのすべての SSE、AVX、ISA コードを生成できます。HLO と LLO から CCG へ渡されるメタデータ情報の一部は、ループが SIMD 化されているか、あるいはループの実行に時間がかかるかどうかといった、ループに関する情報です。反復回数は動的に決定されますが、ArBB は動的に処理されるため、より正確な情報を提供することができます。このメタデータ情報は、時間のかかるループ内では使用されず、その前後で使用される SIMD またはスカラーレジスターの有効範囲を分割すべきかどうかを検証するのに使用します。この情報を利用することで、2 つの binomial-tree カーネルにおいて、ワークロード全体で相乗平均では 6.8%、最大では 60% の効果がありました。

```
f32 sobelX2(f32& src)
{
  // 相対インデックスを使用してピクセル 'src' の周りの
  // 3 x 3 ボックスの ピクセルを収集
  return neighbor(src, -1, -1) + f32(2.0f) *
    neighbor(src, 0, -1) + neighbor(src, 1, -1) +
    f32(-1.0f) * neighbor(src, -1, 1) +
    f32(-2.0f) * neighbor(src, 0, 1) +
    f32(-1.0f) * neighbor(src, 1, 1);
}

f32 sobelY2(f32& src)
{
  // 相対インデックスを使用してピクセル 'src' の周りの
  // 3 x 3 ボックスの ピクセルを収集
  return neighbor(src, -1, -1) * f32(-1.0f) +
      neighbor(src, 1, -1) +
      neighbor(src, -1, 0) * f32(-2.0f) +
      neighbor(src, 1, 0)   * f32(2.0f) +
      neighbor(src, -1, 1) * f32(-1.0f) +
      neighbor(src, 1,   1);
}

template <typename T>
void sobel3x3Mp(f32 src, T& res)
{
assert(typeid(typename uncaptured<T>::type) != typeid(double));

f32 x = sobelX2(src);
f32 y = sobelY2(src);

// 三項演算子 'select' を使用して
// X と Y の大きい方を選択
f32 voxel = select(abs(x) > abs(y), x, y);

// 結果がしきい値を満たしているか検証 [0, max<T>()]
typedef typename uncaptured<T>::type scalar_t;
voxel = max(f32(0.0f), voxel);
voxel = min(voxel,
  f32(std::numeric_limits<scalar_t>::max()));
res = voxel;
}
```

*リスト 8: エッジ検出フィルター **sobel** サンプルコード*

## VII. まとめと総論

インテル® ArBB は、C++ (およびその他の言語) に組み込んでデータ並列とタスク並列を効率良く簡単に指定することができます。プログラマーが*どのように行うのか* ではなく*何を行うのか* を指定できるため、コードの開発、デバッグ、移植、保守にかかる総所有コスト (TCO) を削減できます。本書では、次の点について紹介しました。

- スレッドとベクトル (SIMD) を利用して並列化を行うための統一されたフレームワーク
- 細かい仕様やマイクロアーキテクチャーの詳細を知らなくても、高レベルな抽象化表現を使用することで、メモリーモデル (統合/分割/リモート)、単一プロセッサー (最大でメニーコア・アーキテクチャーまで)、さまざまな幅の SIMD およびベクトル ISA の利用をコンパイラーに指示する方法
- 関数の入れ子に伴うコストを減らし、再構築、将来性、ダイナミック・インライン展開を可能にする先進的な動的コンパイラー・アーキテクチャー
- コンテキストに応じてコンパイルを明示的に管理し、強力な特殊化、汎用プログラミング、決定的な JIT コンパイル・パフォーマンスを実現するための新しいキャプチャー・メカニズム
- データアクセスにおける明確に定義された暗黙の同期
- データ競合状態による並列プログラミングの不具合をすべて排除する、論理的に別々のデータ空間により保証された安全性
- この言語とターゲット・アーキテクチャーにおいて初めて適用された、あるいは新しい方法で適用されたコード生成の最適化 (特に融合)
- 並列処理を構造化した決定論的なパターンに基づくプログラミング・モデル。ステンシルのような一般的なアプリケーション・パターンのビルトインサポート

ArBB の性能は各種コンパイラーで実証されました。SIMD 命令とコアを活用することでベクトルとスレッドによる並列性を実現し、シングルコア・プロセッサーでは相乗平均スピードアップが 1.4～2.1x、8 デュアルスレッド対応コアでは相乗平均スピードアップが 4.3～6.1 という結果が得られました。さらに、8 論理 CPU と 4 SIMD レーンだけで 100x を超えるスピードアップを達成しました。

ArBB は新しい最適化に加えて、既存の最適化を新しい方法で適用します。各種最適化の内容は、サンプルコードを使って説明し、使用したサンプルコードとインテル® ArBB ベータ版に含まれるサンプルコードのパフォーマンス・データを紹介しました。要素関数での SIMD の利用、形状関連の演算子の融合、異なるデータ型に対する演算の融合などの最適化ではそれぞれパフォーマンスが 2 倍以上になりました。そして、融合では約 30x のスピードアップが測定され、融合が並列パフォーマンスを向上する上で最も重要であることが分かりました。

## VIII. 今後の取り組み

今後取り組みたい課題として、仮想マシン・インターフェイス (MIC アーキテクチャーへの処理のオフロード方法 [Skaugen 2010])、クラスターへの ArBB の拡張方法などがあります。

インテル® ArBB はまだベータ版であり、今後さらなるパフォーマンスの最適化が期待できます。メモリーバンド幅によってパフォーマンスが制限される場合が多くありますが、現在インテルではメモリー・トラフィックを減らしてスケーラビリティーを向上させる方法について調査しています。

## 参考文献

[Abrahams 2004] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming*, Addison-Wesley, 2004.

[Aldinucci 2007] M. Aldinucci and M. Danelutto, *Skeleton-based parallel programming: Functional and parallel semantics in a single shot*, Comput. Lang. Syst. Struct., 33(3-4), 2007, pp. 179-192.

[Allen 2002] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures.,* Morgan Kaufmann Publishers, San Francisco, 2002.

[APL] APL: A Programming Language. URL: http://www.users.cloud9.net/~bradmcc/APL.html

[ArBB 2011] Intel® Array Building Blocks Documentation, URL : http://software.intel.com/en-us/articles/intel-array-building-blocksdocumentation/, もしくは http://software.intel.com/en-us/articles/intel-arraybuilding-blocks/, 2010

[Asanovic 2006] K. Asanovic et al, *The Landscape of Parallel Computing Research: A View from Berkeley*, EECS Department, University of California, Berkeley EECS-2006-183, 2006.

[Blelloch 1990] G. E. Blelloch, *Vector models for data-parallel computing*, MIT Press, 1990.

[Blelloch 1993] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein and M. Zagha, *Implementation of a portable nested data-parallel language*, PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, 1993, pp. 102-111

[Blelloch 1996] G. E. Blelloch, *Programming parallel algorithms*, Commun. ACM, 39(3), 1996, pp. 85-97.

[Bosch 1998] J. Bosch. *Design patterns as language constructs*. Journal of Object-Oriented Programming, 11(2):18-32, 1998.

[Bromling 2002] S. Bromling, S. MacDonald, J. Anvik, J. Schaefer, D.Szafron, K. Tan, *Pattern-based parallel programming*, Proceedings of the International Conference on Parallel Programming (ICPP'2002), August 2002, Vancouver Canada, pp. 257-265.

[Brunschen 2000] Brunschen, C., Brorsson, M.: OdinMP/CCp - a portable implementation of OpenMP for C. Concurrency - Practice and Experience 12 (2000) 1193–1203

[Buck 2007] I. Buck, *GPU Computing: Programming a Massively Parallel Processor*, Proceedings of the International Symposium on Code Generation and Optimization, March 11-14, 2007.

[Catanzaro 2010a] Catanzaro, B. , Garland, M. and Keutzer, K.*Copperhead: Compiling an Embedded Data Parallel Language.* Technical Report UCB/EECS-2010-124, EECS Department, University of California, Berkeley, September 16, 2010.

[Catanzaro 2010b] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovi´c, J.Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. *SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization.* Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, 2010.

[Chakravarty 2001] Manuel M. T. Chakravarty and Gabriele Keller and Roman Lechtchinsky and Wolf Pfannenstiel. *Nepal --- Nested Data Parallelism in Haskell*. Proc. 7th International

Euro-Par Conference, volume 2150 of Lecture Notes in Computer Science, pp 524-534. Springer-Verlag, 2001.

[Chatterjee 1993] *Compiling nested data-parallel programs for sharedmemory multiprocessors.* ACM Trans. Program. Lang. Syst. 15(3), July 1993, pp 400-462.

[Clyther 2010] *Clyther: Python language extension for OpenCL.* URL: http://clyther.sourceforge.net/, 2010.

[Cole 1989] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT Press, 1989.

[Cole 2004] M. Cole. *Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming,* Parallel Computing, 30(3), pp. 389-406, March 2004

[Czarnecki 2000] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, 2000, ACM Press/Addison-Wesley.

[Du Toit 2010] S. Du Toit, A Data-parallel Virtual Machine, URL: http://software.intel.com/en-us/articles/data-parallel-vm/, 2010

[Gamma 1994] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Ghuloum 2007] A. Ghuloum, E. Sprangle, J. Fang, G.Wu, and X. Zhou. *Ct: A Flexible Parallel Programming Model for Tera-scale Architectures.* Technical Report White Paper, Intel Corporation, 2007.

[Herrington 2003] J. Herrington, *Code Generation in Action*, 2003, Manning Publications.

[Hillis 1986] W. D. Hillis and J. Guy L. Steele. Data *parallel algorithms*. Communications of the ACM, 29(12):1170–1183, 1986.

[Lee 1996] P. Lee and M. Leone, *Optimizing ML with run-time code generation*, Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation, p.137-148, May 1996.

[MacDonald 2002] S. MacDonald, J. Anvik, S. Bromling, D. Szafron, J. Schaeffer and K. Tan. *From patterns to frameworks to parallel programs*, Parallel Computing, 28(12);1663-1683, 2002.

[Massingill 1999] M. Massingill, T. Mattson, and B. Sanders. *A pattern language for parallel application programs*. Technical Report CISE TR 99-022, University of Florida, 1999.

[Mattson 2004] T. Mattson, B. Sanders, B. Massingill, *Patterns for Parallel Programming*, 2004, Pearson Education.

[McCool 2002] M. McCool, Z. Qin, and T. Popa, *Shader metaprogramming*, HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2002, pp. 57-68.

[McCool 2004a] M. McCool, S. Du Toit, T. Popa, B. Chan and K. Moule, *Shader algebra*, ACM Trans. Graph., 23(3), 2004, pp. 787-795.

[McCool 2004b] M. McCool and S. Du Toit, *Metaprogramming GPUs with Sh*, 2004, AK Peters.

[McCool 2006] M. McCool. Data-Parallel Programming on the Cell BE and the GPU Using the

RapidMind Development Platform. GSPx Multicore Applications Conference, 9 pages, 2006.

[McCool 2010] M. McCool. *Structured Parallel Programming with Deterministic Patterns*, HotPar 2010 (2nd USENIX Workshop on Hot Topics in Parallelism), Berkeley, CA, 14-15 June 2010.

[Munshi 2008] A. Munshi, *OpenCL Specification Version 1.0*, The Khronos Group, 2008.

[Nickolls 2008] J. Nickolls, I. Buck, M. Garland, and K. Skadron. *Scalable parallel programming with CUDA.* Queue, 6(2):40–53, Mar/Apr 2008.

[Nuzman 2011] D. Nuzman, I Rosen, S. Dyshel, A. Zaks, E. Rohou, K. Williams, A. Cohen, and D. Yuste, *Auto-Vectorize Once, Run Everywhere,* Proceedings of the International Symposium on Code Generation and Optimization, 2011

[Owens 2005] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krueger, A. E. Lefohn, and T. J. Purcell, *A survey of general-purpose computation on graphics hardware*, Eurographics 2005, State of Art Report.

[Siu 1996] S. Siu, M. De Simone, D. Goswami, and A. Singh. *Design patterns for parallel programming*. Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96), pp. 230–240, 1996.

[Skaugen 2010] K. Skaugen, *HPC Technology - Scale Up and Scale Out*, Keynote, International Supercomputing Conference, 2010

[Skillicorn 1998] D. B. Skillicorn and D. Talia, *Models and languages for parallel computation*, ACM Comput. Surv., 30(2), 1998, pp.123-169.

[Tan 2003] K. Tan, D. Szafron, J. Schaeffer, J. Anvik and S. MacDonald, *Using generative design patterns to generate parallel code for a distributed memory environment*, PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, 2003, pp 203-215.

[Veldhuizen 1999] T. L. Veldhuizen, *C++ templates as partial evaluation*, In ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, 1999.

[Zhao 2005] Yuan Zhao, Ken Kennedy. *Scalarization on Short Vector Machines*. Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005. Pages: 187-196. ISBN:0-7803-8965-4