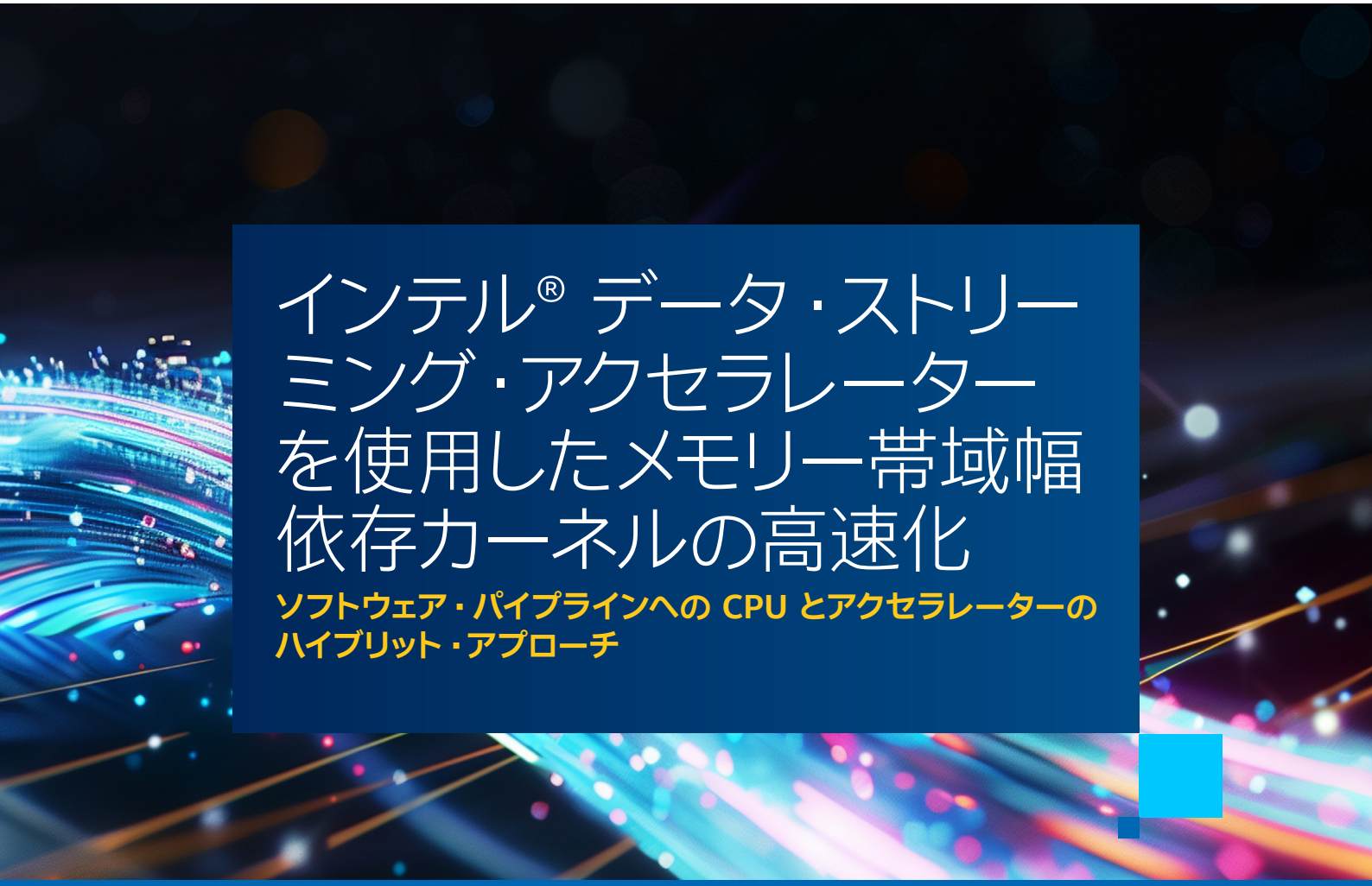

インテル® データ・ストリーミング・アクセラレーターを使用したメモリー帯域幅依存カーネルの高速化

ソフトウェア・パイプラインへの CPU とアクセラレーターのハイブリット・アプローチ

Vamsi Sripathi インテル コーポレーション ソフトウェア・イネーブリング & 最適化エンジニア

インテル® データ・ストリーミング・アクセラレーター（インテル® DSA）は、第 4 世代インテル® Xeon® スケーラブル・プロセッサー以降で利用できる、高性能なデータコピーおよび変換アクセラレーターです。メモリーコピー、比較、フィルなどのさまざまなデータ操作をサポートします。インテル® DSA は、作業キューと記述子を使用して CPU ソフトウェアと対話します。記述子には、操作のタイプ、ソースアドレスとデスティネーション・アドレス、データ長など、目的の操作に関する情報が含まれています。詳細については、The Parallel Universe 第 53 号の「ダイレクト・メモリー・アクセスの先へ：インテル® データ・ストリーミング・アクセラレーターによるデータセンター・コストの削減」を参照してください。

多くの科学および商用アプリケーションは、最適なパフォーマンスを実現するため、高性能な CPU と高いメモリー帯域幅を必要とします。CPU は世代を重ねるごとに、コアの追加、SIMD 幅の増加、新しい ISA 拡張などのマイクロアーキテクチャー機能によって性能が大幅に向上していますが、DRAM 帯域幅の向上はこれまで CPU の改良に追従できていませんでした。DRAM からデータをフェッチする速度が CPU 実行ユニットのストールを防ぐのに十分ではないため、実際のアプリケーションでは期待される性能向上が得られない「メモリーウォール」と呼ばれる現象に長年悩まされてきました。

CPU には、ロード / ストアキューとスーパーキュー形式のアーキテクチャー・バッファーと、メモリーウォールの効果を軽減するハードウェア・プリフェッチャーがありますが、各 CPU コアがハードウェア・キューに保持できるエントリーの数は限られているため、コア数が少ない場合はメモリー帯域幅への影響は限定的です。DRAM からデータをフェッチする必要がある場合、各メモリー要求が DRAM レイテンシーによりバッファースロットを長時間占有するため、キューがボトルネックになります。この対策として、利用可能な DRAM 帯域幅を完全に飽和させるため、より多くの CPU コアを使用して同時にメモリーアクセス要求を生成するのが一般的です。ただし、コア数が少ない状態で高いメモリー帯域幅を実現することは依然として困難です。

この記事では、インテル® DSA の長所を CPU と組み合わせて補完し、メモリー依存カーネル（つまり、メモリー・サブシステムが演算のデータオペランドを CPU コアに供給できる速度によってパフォーマンスが決まる操作）を高速化する手法について説明します。

パフォーマンスの測定には、標準の STREAM ベンチマーク（英語）を使用し、CPU のメモリー・パフォーマンスの特性を把握します。STREAM は、Copy、Scale、Add、Triad の 4 つのカーネルで構成されており（**表 1**）、これらのカーネルはすべてメモリー依存です。

カーネル	演算	リードバイト数（キャッシュをバイパスするストアを使用）	ライトバイト数	FLOPS
COPY	A[i] = B[i]	8	8	0
SCALE	A[i] = scalar × B[i]	8	8	1
ADD	C[i] = A[i] + B[i]	16	8	1
TRIAD	C[i] = A[i] + scalar × B[i]	16	8	2

表 1. STREAM カーネルの特性

インテル® DSA は強力なデータコピー / 変換エンジンとして機能しますが、データオペランドに対する乗算、加算、積和演算（FMA : Fused Multiply-Add）などの算術演算の実行はサポートしていません。そのため、インテル® DSA だけでは、メモリーと演算が混在するアプリケーション・カーネルを実行できません。ただし、インテル® DSA は、演算によって生成されるデスティネーション・データの場所（DRAM または CPU 上の最終レベルキャッシュ（LLC））を制御する独自の機能を提供します。例えば、DRAM にあるソースバッファーから、DRAM（CPU キャッシュをバイパス）または LLC にあるデスティネーションにデータをコピーできます。

インテル® DSA のキャッシュ書き込み機能を、DRAM から LLC へのプロキシー・ハードウェア・プリフェッチ・エンジンとして使用し、CPU コアで演算を処理できます。このソリューションは、インテル® DSA のデータ転送（DRAM から LLC へ）と、LLC から CPU レジスターへの非同期コピーによる CPU 計算を効率的にオーバーラップすることで機能します。デフォルトでは、インテル® DSA は LLC の約 14MB の部分に書き込むことができます（第 4 世代インテル® Xeon® スケーラブル・プロセッサー上の 15 ウェイの LLC のうち 2 ウェイ、有効サイズ = 2/15 * LLC のサイズ = 14MB）。**図 1** は、CPU のみのアプローチと CPU + インテル® DSA のハイブリッド・ワークフローの高レベルの違いを示しています。

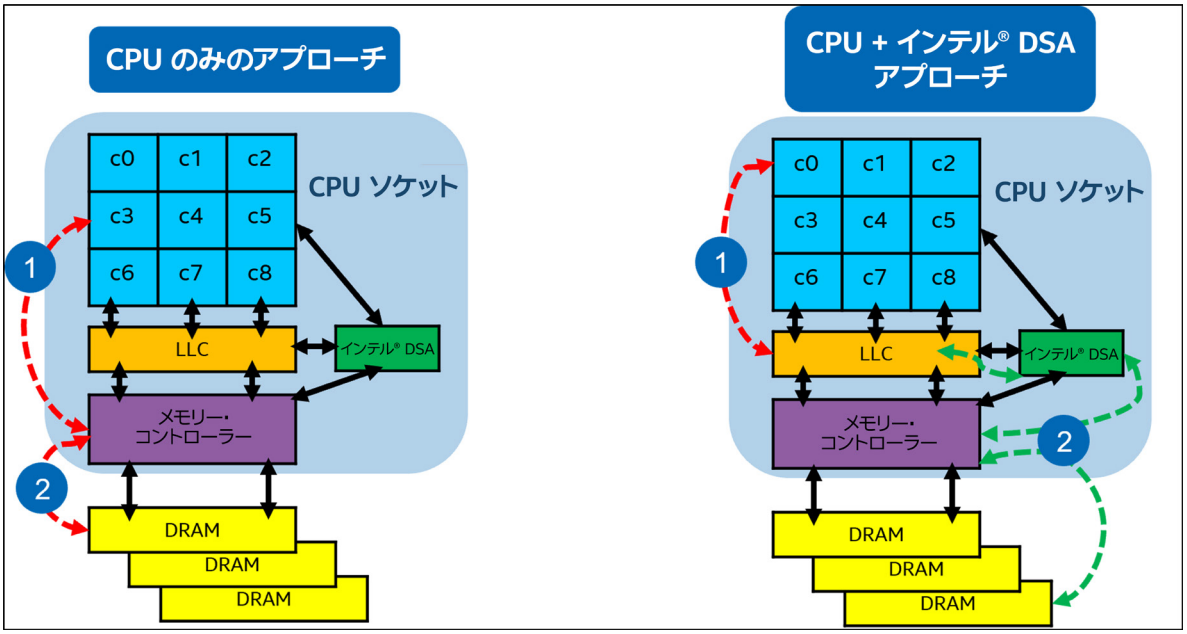

図 1. CPU のみのアプローチと CPU + インテル® DSA のハイブリッド実装の高レベルの違い

CPU のみのアプローチでは、ステップ 1 と 2 は、DRAM 上のデータに対する CPU コアからのロード / ストア要求を指し、メモリー・コントローラーに送られます。CPU + インテル® DSA アプローチのステップ 1 では、CPU は DRAM ではなく LLC からデータをロードします。ステップ 2 では、インテル® DSA は DRAM（メモリー・コントローラー経由）から LLC へのデータ転送を開始します。ステップ 1 と 2 はパイプライン化され、非同期的に実行されるため、CPU が T_x の時点で読み取る必要のあるデータは、$T_{(x-1)}$ でインテル® DSA によって LLC にコピー済みです。つまり、インテル® DSA が同時に次の反復のデータを DRAM からフェッチして LLC に書き込む一方で、CPU は現在の反復のデータを LLC から読み取って演算を実行します。ソフトウェア・パイプラインにより、すべてのインテル® DSA エンジンが効率良く使用されるようになります。

図 2 は、DRAM からデータ配列を読み取るなどの基本的な CPU 操作に対する CPU + インテル® DSA のハイブリッド・パイプライン・アプローチのワークフローを示しています。この例では、各 CPU スレッドに対して、キューの深さを 4 に設定し、各エントリーに入力バッファーの 1MB のデータを保持しています。つまり、CPU が LLC から 4MB を読み取っている間に、インテル® DSA は DRAM から次の 4MB のチャンクをフェッチします。

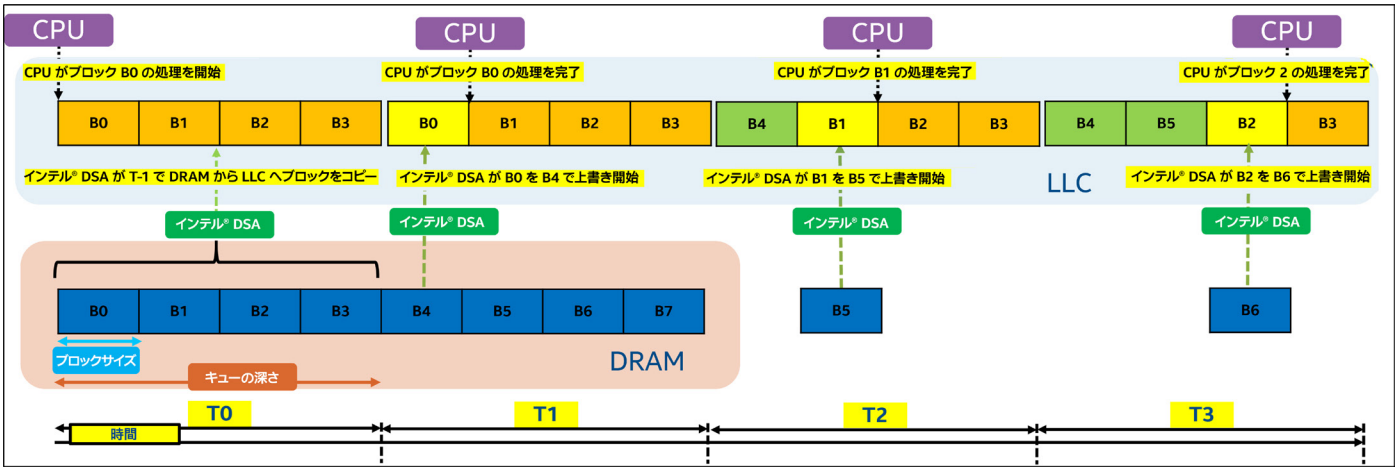

図 2. CPU + インテル® DSA のハイブリッド実装で 1 つの CPU コアで 1 つのバッファーにアクセスするデータ操作のパイプラインの例

インテル® Data Mover Library（インテル® DML）（英語）は、インテル® DSA を使用してデータ操作を実行する C/C++ API を提供するオープンソース・ライブラリーです。インテル® DML を使用して、インテル® DSA を初期化し、作業記述子を作成し、STREAM カーネルのジョブステータスを送信および照会します。**図 3** に例を示します。Triad 操作は OpenMP* スレッドを使用して並列化されます。各スレッドはデータバッファーを均等に分割し、キューのようなメカニズムを使用してスレッドチャンクをさらにブロックに分割します。各 CPU スレッドは、非同期コピー操作をインテル® DSA に送信し、完了するまで待機してからデータにアクセスして Triad 操作を計算します。ループの中で、非同期コピー操作は、将来の反復（`QUEUE_DEPTH`）でアクセスされるデータに対して送信されます。**図 4** は、インテル® DML を使用してデスティネーションを LLC（`DML_FLAG_PREFETCH_CACHE`）にする非同期コピー操作のコードを示しています。

```
void dsa_omp_triad(long long *p_n, double *p_src1, double *p_src2, double *p_dst,
                   double *p_tmp1, double *p_tmp2,
                   dml_job_t **p_dml_jobs_t1, dml_job_t **p_dml_jobs_t2)
{
  long long n = *p_n;

#pragma omp parallel
  {
    int nthrs = omp_get_num_threads();
    int ithr  = omp_get_thread_num();

    long long chunk = n/nthrs;
    long long tail  = n - (chunk*nthrs);
    long long start = ithr * chunk;
    if ((tail) && (ithr == nthrs-1)) {
      chunk += tail;
    }

    double *p_t_src1 = p_src1 + start;
    double *p_t_src2 = p_src2 + start;
    double *p_t_dst  = p_dst  + start;

    double *p_t_tmp1 = p_tmp1 + (ithr * ((BLK_SIZE_IN_BYTES*QUEUE_DEPTH)/sizeof(double)));
    double *p_t_tmp2 = p_tmp2 + (ithr * ((BLK_SIZE_IN_BYTES*QUEUE_DEPTH)/sizeof(double)));

    dml_job_t **p_t_dml_jobs_t1 = p_dml_jobs_t1 + (ithr*QUEUE_DEPTH);
    dml_job_t **p_t_dml_jobs_t2 = p_dml_jobs_t2 + (ithr*QUEUE_DEPTH);

    long long blk_elems = BLK_SIZE_IN_BYTES/sizeof(double);
    long long num_blks  = chunk/blk_elems;
    tail = chunk - (blk_elems*num_blks);

    for (int i=0; i<QUEUE_DEPTH; i++) {
      async_copy(&blk_elems, p_t_src1+((QUEUE_DEPTH+i)*blk_elems), p_t_tmp1+(i*blk_elems), p_t_dml_jobs_t1[i]);
      async_copy(&blk_elems, p_t_src2+((QUEUE_DEPTH+i)*blk_elems), p_t_tmp2+(i*blk_elems), p_t_dml_jobs_t2[i]);
    }

    long long tmp_elems = QUEUE_DEPTH * blk_elems;
    seq_triad(&tmp_elems, p_t_src1, p_t_src2, p_t_dst);

    int i;
    for (i=QUEUE_DEPTH; i<(num_blks-QUEUE_DEPTH); i++) {
      dml_wait_job(p_t_dml_jobs_t1[i%QUEUE_DEPTH]);
      dml_wait_job(p_t_dml_jobs_t2[i%QUEUE_DEPTH]);

      seq_triad(&blk_elems, p_t_tmp1+((i%QUEUE_DEPTH)*blk_elems), p_t_tmp2+((i%QUEUE_DEPTH)*blk_elems), p_t_dst+(i*blk_elems));

      async_copy(&blk_elems, p_t_src1+((QUEUE_DEPTH+i)*blk_elems), p_t_tmp1+((i%QUEUE_DEPTH)*blk_elems), p_t_dml_jobs_t1[i%QUEUE_DEPTH]);
      async_copy(&blk_elems, p_t_src2+((QUEUE_DEPTH+i)*blk_elems), p_t_tmp2+((i%QUEUE_DEPTH)*blk_elems), p_t_dml_jobs_t2[i%QUEUE_DEPTH]);
    }

    for (int j=i; j<(i+QUEUE_DEPTH); j++) {
      dml_wait_job(p_t_dml_jobs_t1[j%QUEUE_DEPTH]);
      dml_wait_job(p_t_dml_jobs_t2[j%QUEUE_DEPTH]);
      seq_triad(&blk_elems, p_t_tmp1+((j%QUEUE_DEPTH)*blk_elems), p_t_tmp2+((j%QUEUE_DEPTH)*blk_elems), p_t_dst+(j*blk_elems));
    }

    if (tail) {
      seq_triad(&tail, p_t_src1+(num_blks*blk_elems),  p_t_src2+(num_blks*blk_elems),  p_t_dst+(num_blks*blk_elems));
    }
  }
}
```

図 3. OpenMP* とインテル® DML を使用した STREAM Triad カーネルの CPU + インテル® DSA のハイブリッド実装

```
void async_copy(long long *p_n, double *p_src, double *p_dst, dml_job_t *p_dml_job)
{
  dml_status_t status;

  p_dml_job->operation             = DML_OP_MEM_MOVE;
  p_dml_job->flags                 = DML_FLAG_COPY_ONLY|DML_FLAG_PREFETCH_CACHE;
  p_dml_job->source_first_ptr      = (void *)p_src;
  p_dml_job->destination_first_ptr = (void *)p_dst;
  p_dml_job->source_length         = (*p_n)*sizeof(double);
  p_dml_job->destination_length    = (*p_n)*sizeof(double);

  status = dml_submit_job(p_dml_job);

  if (status) {
    printf ("\tdml_submit_job status failed, status = %u\n", status); fflush(0);
  }
}
```

図 4. インテル® DML を使用した LLC への非同期コピー

標準の STREAM カーネルに加えて、多くのアプリケーション・ドメインでよく使用されるベクトルドット積演算（`res += a[i] × b[i];` 読み取りのみ、書き込みなし）のベンチマークも行いました。カーネルのベンチマークは次のように行いました。

- 各入力バッファーのサイズは 1GB、Scale と Dot のメモリー・フットプリントは 2GB、Triad は 3GB にしました。各カーネルを 100 回実行して、最高のパフォーマンスを結果としました。
- CPU のみの実装では、CPU コアで OpenMP* により並列化された操作を実行しました。キャッシュをバイパスするストア / 非テンポラルなストア（`vmovntpd`）を使用しました。
- インテル® DSA + CPU 実装では、インテル® DSA を使用して DRAM から LLC に入力バッファーをフェッチしました。CPU による非テンポラルなストアを使用して、メモリー内のデスティネーション・バッファーを直接更新しました。

図 5 は、第 4 世代インテル® Xeon® スケーラブル・プロセッサー（56 コア、8 チャネル DDR5@4800MT/s、理論上のピーク帯域幅は 8 チャネル x 8 バイト x 4.8GT/s = 307GB/s）のさまざまなコア数でのパフォーマンスを示しています。CPU のみの実装と比較したインテル® DSA + CPU のスピードアップは、ヒートマップで表示しています。

CPU コア数	Triad、リード 2:ライト 1 (GB/s)			ドット積、100% リード (GB/s)			Scale、リード 1:ライト 1 (GB/s)		
	CPU のみ	インテル® DSA + CPU	スピードアップ	CPU のみ	インテル® DSA + CPU	スピードアップ	CPU のみ	インテル® DSA + CPU	スピードアップ
1	20.09	37.23	1.85	15.78	32.55	2.06	21.13	33.51	1.59
2	39.8	71.97	1.81	31.49	65.48	2.08	41.71	66.37	1.59
3	57.83	98.46	1.70	46.23	82.31	1.78	60.62	96.89	1.60
4	74.3	115.77	1.56	60.55	99.8	1.65	77.84	121.48	1.56
5	91.94	133.41	1.45	74.72	120.74	1.62	96.02	137.47	1.43
6	108.56	154.55	1.42	89.92	129.4	1.44	112.36	160.99	1.43
7	124.21	173.53	1.40	104.82	141.94	1.35	127.46	180.40	1.42
8	138.96	189	1.36	118.11	156.85	1.33	140.70	198.91	1.41
9	153.02	189.66	1.24	131.33	169.62	1.29	151.79	214.96	1.42
10	167.11	191.17	1.14	144.75	187.08	1.29	162.44	223.37	1.38
11	176.66	201.54	1.14	157.29	198.32	1.26	172.00	231.38	1.35
12	186.78	208.3	1.12	170.52	208.94	1.23	179.69	228.60	1.27
13	196.31	212.49	1.08	178.94	217.53	1.22	186.15	229.21	1.23
14	204.55	217.95	1.07	190.3	225.45	1.18	192.15	227.13	1.18
15	211.51	221.73	1.05	203.22	234.85	1.16	197.61	228.88	1.16
16	218.38	223.77	1.02	215.32	242.56	1.13	201.79	228.96	1.13
17	223.51	226.84	1.01	224.49	242.84	1.08	205.78	228.72	1.11
18	228.37	226.48	0.99	229.66	242.89	1.06	209.17	227.17	1.09
19	232.59	227.97	0.98	235.9	241.71	1.02	212.55	225.18	1.06
20	234.98	229.38	0.98	245.3	243.05	0.99	215.23	225.72	1.05

図 5. CPU のみの実装と CPU + インテル® DSA のハイブリッド実装のパフォーマンス比較

前述のように、1 コアの CPU メモリー・パフォーマンスはアーキテクチャーのロード / ストアバッファーのサイズによって決まりますが、インテル® DSA にはこの制限はありません。したがって、CPU + インテル® DSA ハイブリッド方式では、インテル® DSA がパイプライン方式でデータを DRAM から LLC にコピーするため、CPU によって読み取られるデータはすべて LLC でヒットします。LLC ヒット・レイテンシーは DRAM アクセス・レイテンシーよりも短いため、メモリー要求がロード / ストアキューに留まる時間が短縮され、その結果、1 コアの帯域幅は、カーネルタイプに応じて 1.6 倍～ 2 倍に増加します。

コア数が 3 以下の場合、CPU が LLC からデータを読み出しても、LLC から 1 ブロックのデータを処理する時間は、インテル® DSA が DRAM から 1 ブロックをフェッチする時間よりも長くなり、DRAM レイテンシーが完全に隠蔽されるため、少ないコア数で高いゲインが得られます。ワークフローに CPU コアを追加すると、LLC からの CPU リード帯域幅はインテル® DSA よりも速くなり、CPU はインテル® DSA が LLC へのコピーを完了するのを待機することになります。この問題を軽減するため、使用するコア数に応じて 2 つの異なる実装を使用します。コア数が多い場合、インテル® DSA のキューで未処理の要求が多くなると大きな競合が発生します。そこで、CPU がインテル® DSA とともにメモリー要求も処理するような別の実装に切り替えます。Triad と Dot では、CPU も使用して入力バッファーの 1 つ（`src1`）をフェッチし、インテル® DSA がもう 1 つの入力バッファー（`src2`）をフェッチします。この実装は、コア数が多い場合に次のような利点があります。第 1 に、インテル® DSA が DRAM から LLC へのコピーを完了するのを待機する代わりに、CPU を使用できます。第 2 に、インテル® DSA は DRAM から LLC にバッファーを 1 つのみフェッチすればよくなるため、未処理のインテル® DSA 要求の数が半分になり、インテル® DSA キューの競合が減少します。全体として、CPU + インテル® DSA ハイブリッド方式のパフォーマンス上の利点は、コア数が増えるにつれて小さくなります。

結論として、STREAM ベンチマークは、インテル® DSA が少ない CPU コアを使用してパフォーマンスを向上できることを示しています。例えば、Scale カーネルでは、インテル® DSA を使用して 9 コアで、20 コアの CPU のみの実装と同じパフォーマンスを達成できます。Triad カーネルでは、インテル® DSA を使用して 8 コアで、12 コアの CPU のみの実装と同じパフォーマンスを達成できます。Dot カーネルでは、インテル® DSA を使用して 5 コアで、8 コアの CPU のみの実装と同じパフォーマンスを達成します。インテル® DSA と組み合わせる CPU の数が少ないほど、メモリー・パフォーマンスが向上するため、これはヘテロジニアス・アプリケーションの高速化に役立ちます。計算依存とメモリー依存のカーネルが混在するヘテロジニアス・アプリケーションでは、計算カーネルにより多くの CPU を割り当てて、全体的なパフォーマンスを向上できます。シングルスレッドのメモリー帯域幅依存のアプリケーションも高速化できます。単一の CPU では帯域幅を完全に飽和させることはできないため、アムダールの法則の影響を軽減するという観点から、インテル® DSA はワークロードのシーケンシャルな部分の時間を短縮するのに使用できます。