GPUコンピューティング（CUDA）講習会

CUDAプログラムの最適化

東工大学術情報センター　丸山直也

はじめに

- シングルGPUプログラミングにおける最適化を紹介します
- NVIDIA提供の以下の資料に基づいています
  - Mark Harris, “Optimizing Parallel Reduction in CUDA”
  - http://developer.download.nvidia.com/compute/cuda/1_1/Website/Data-Parallel_Algorithms.html#reduction
  - CUDA SDKのreductionディレクトリに資料、プログラムあり

サンプルコード

- /work/nmaruyam/gpu-tutorial/reduction
  - CUDA SDKの一部を講習会用に改変したものです
- ホームディレクトリへコピーしてお使いください

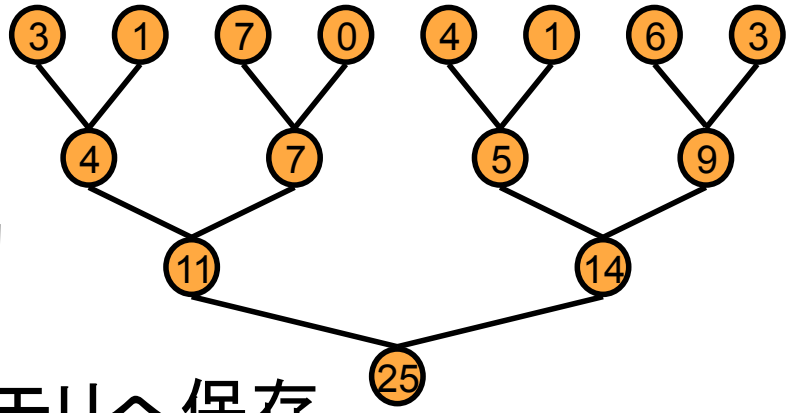
```
$ cp –r /work/nmaruyam/gpu-tutorial/reduction ~
$ cd ~/reduction
$ cd projects/reduction
$ make
$ cd ../..
$ ./bin/linux/release/reduction
```

例題： 配列のリダクション

- 配列の総和、などを求める計算
  - GPUメモリにある大規模な配列の総和を計算
- データタイプは32ビット整数と仮定
  - 他のデータタイプにも応用できるが、1要素のサイズが異なる場合は一部手法も異なる
- 計算量がメモリアクセス量に比べて少ない→プログラミングによる実行効率の違いが大きくなりがち
- メモリアクセスなどの効率化が重要

並列リダクション

- ２要素のリダクションを並列に処理
- log(n)ステップで完了
- CUDAにおける基本方針
  1. 要素数と同数のスレッドを起動
  2. 各スレッドが1要素をグローバルメモリから読み込み、共有メモリへ保存
  3. 半分のスレッドが2つの要素の和を計算
  4. さらに半分の要素が次のステップの和を計算
  5. １要素になるまで続ける



- 他のスレッドブロックでリダクションされた値を用いる場合は、同期のためにそこで一端カーネルを区切る

最適化指標

- アプリケーションの特性にあった指標
  - 計算量が多いプログラム　→ FLOPS
    - 行列積など
  - メモリアクセスが多いプログラム → バンド幅
    - 流体、**リダクション**、など
- 理論最大バンド幅に対する実行バンド幅を評価
  - TSUBAME Tesla GPU (S1070): 102 GB/s

Reduction #1

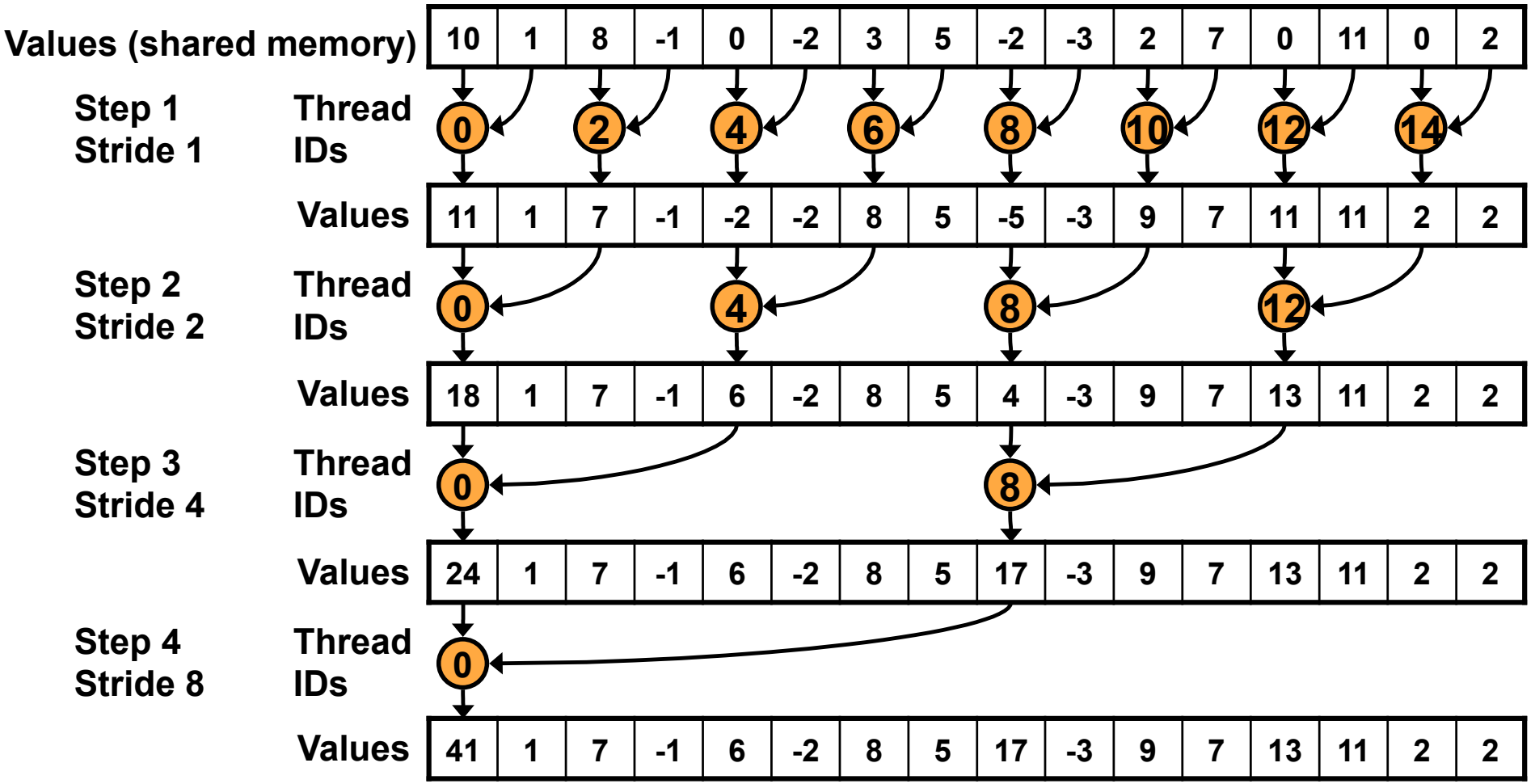
```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // 各スレッドが1要素をグローバルメモリより読み込み
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // 共有メモリを使い、半々にリダクション
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // 残り1要素になったら結果をグローバルメモリへ書き出して終了
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

スレッドブロックの動作



結果

	実行時間	バンド幅	理論最大値 に対する割合
Kernel 1	**3.51 ms**	**4.77 GB/s**	**4.6 %**

ブロックサイズ128、TSUBAME S1070の1GPUを利用

リダクション#1の性能低下の要因

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // 各スレッドが1要素をグローバルメモリより読み込み
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // 共有メモリを使い、半々にリダクション
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // 残り1要素になったら結果をグローバルメモリへ書き出して終了
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

ダイバージェント分岐多発
“%”演算は高コスト

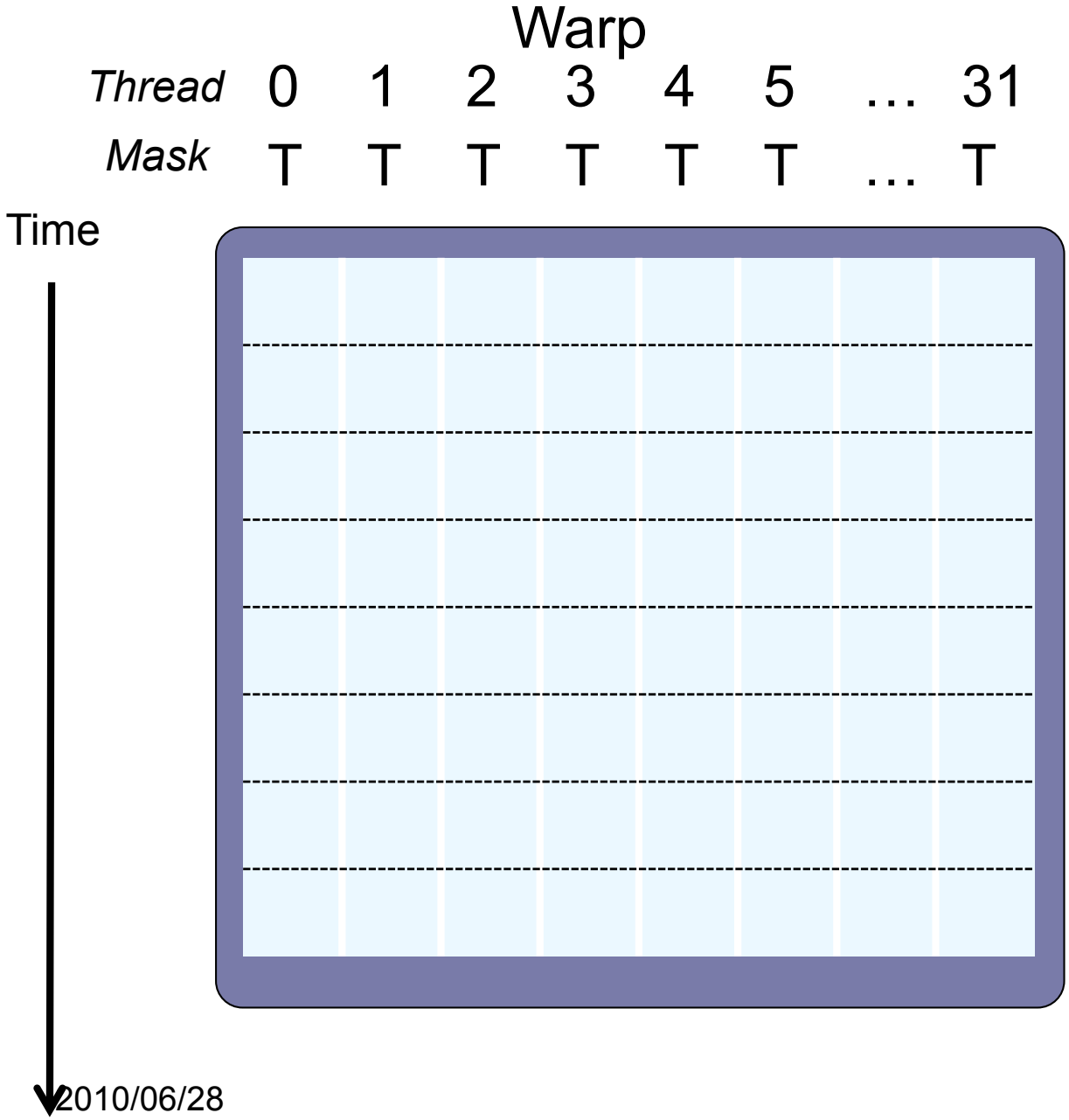
分岐

- CUDAでは通常のC/Fortranの制御文を利用可能
  - if, while, for, do-while
- しかし、GPUハードウェアによる実行の仕方はCPUとは異なる
  - 「ワープ」と呼ばれるスレッドのグループはすべて同じコードを実行
- ワープ
  - スレッドブロック内の32個の連続したスレッドのグループ（32という数字は将来変更の可能性有り）

ダイバージェント分岐

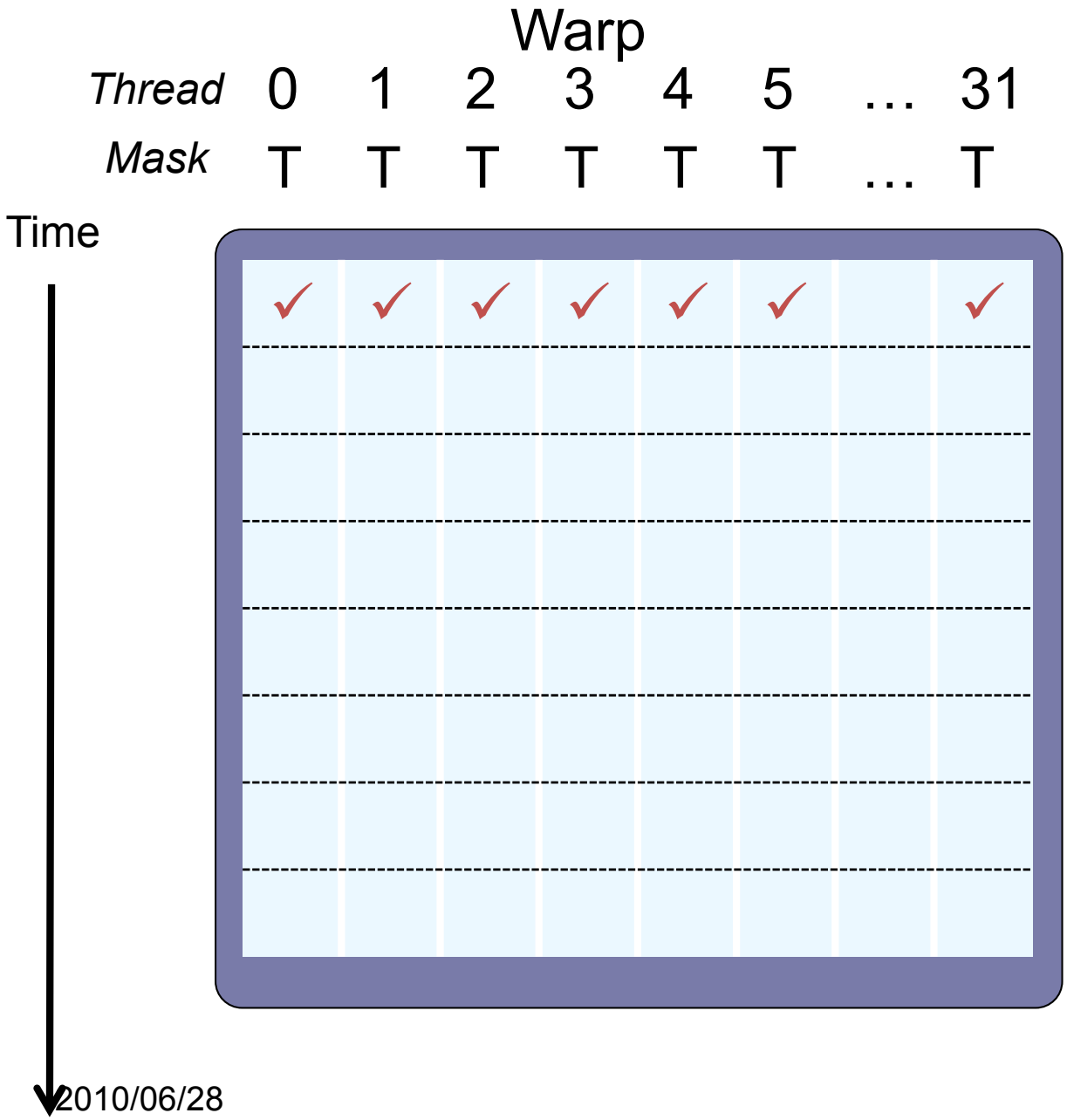
- ワープ内のスレッドが異なる分岐結果を実行する場合 → ワープ内スレッドすべてが両方の条件分岐コードを実行
- CUDAにおけるよくある性能低下の原因

分岐処理



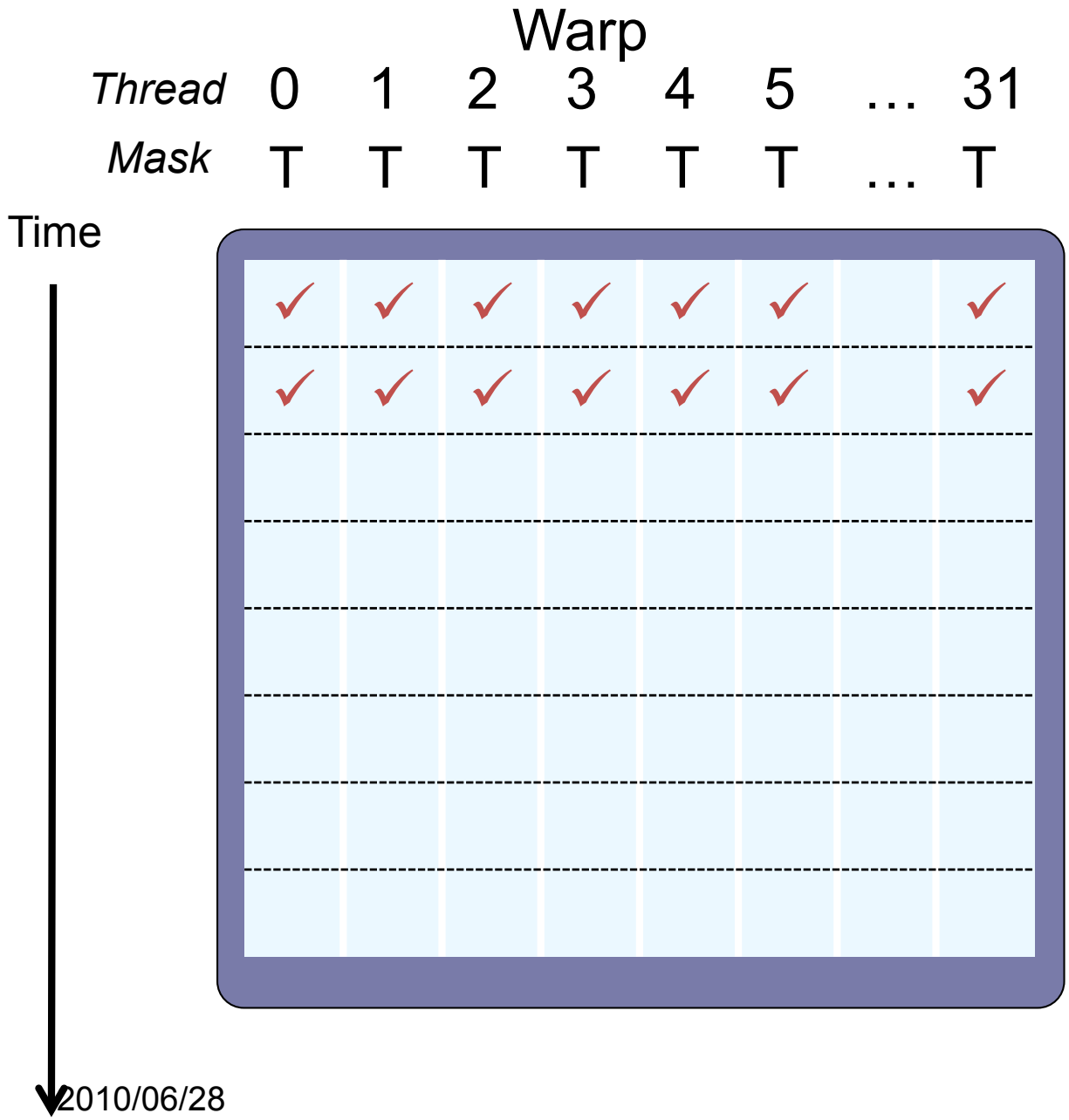

```
x = z - k;
if (x > 0) {
   t = y * x - a;
   s = y * y;
} else {
   t = 2 * x + a;
   s = x * y;
}
a[i] = t * s;
```

分岐処理



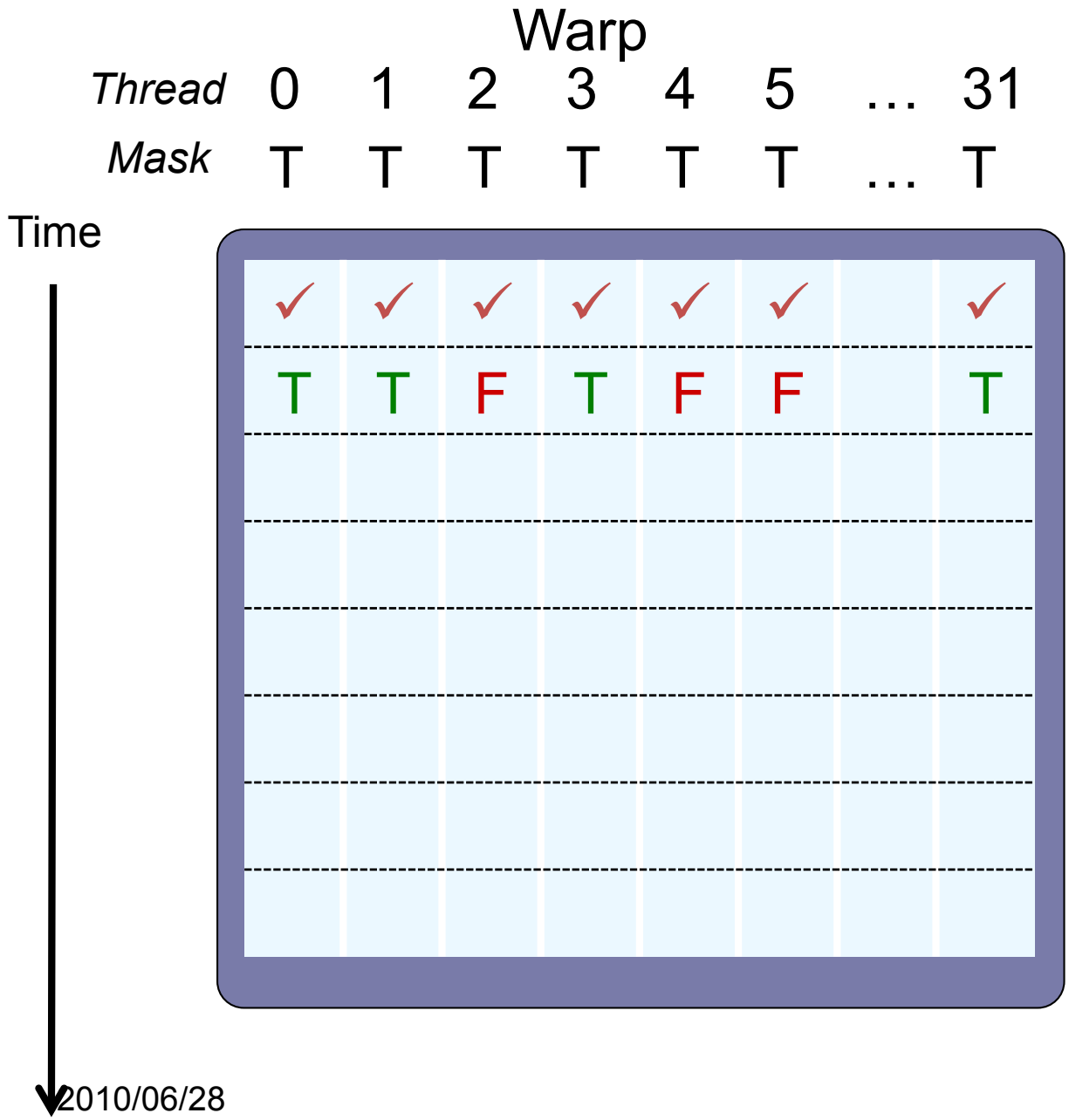

```
  x = z - k;
  if (x > 0) {
     t = y * x - a;
     s = y * y;
} else {
     t = 2 * x + a;
     s = x * y;
  }
  a[i] = t * s;
```

分岐処理



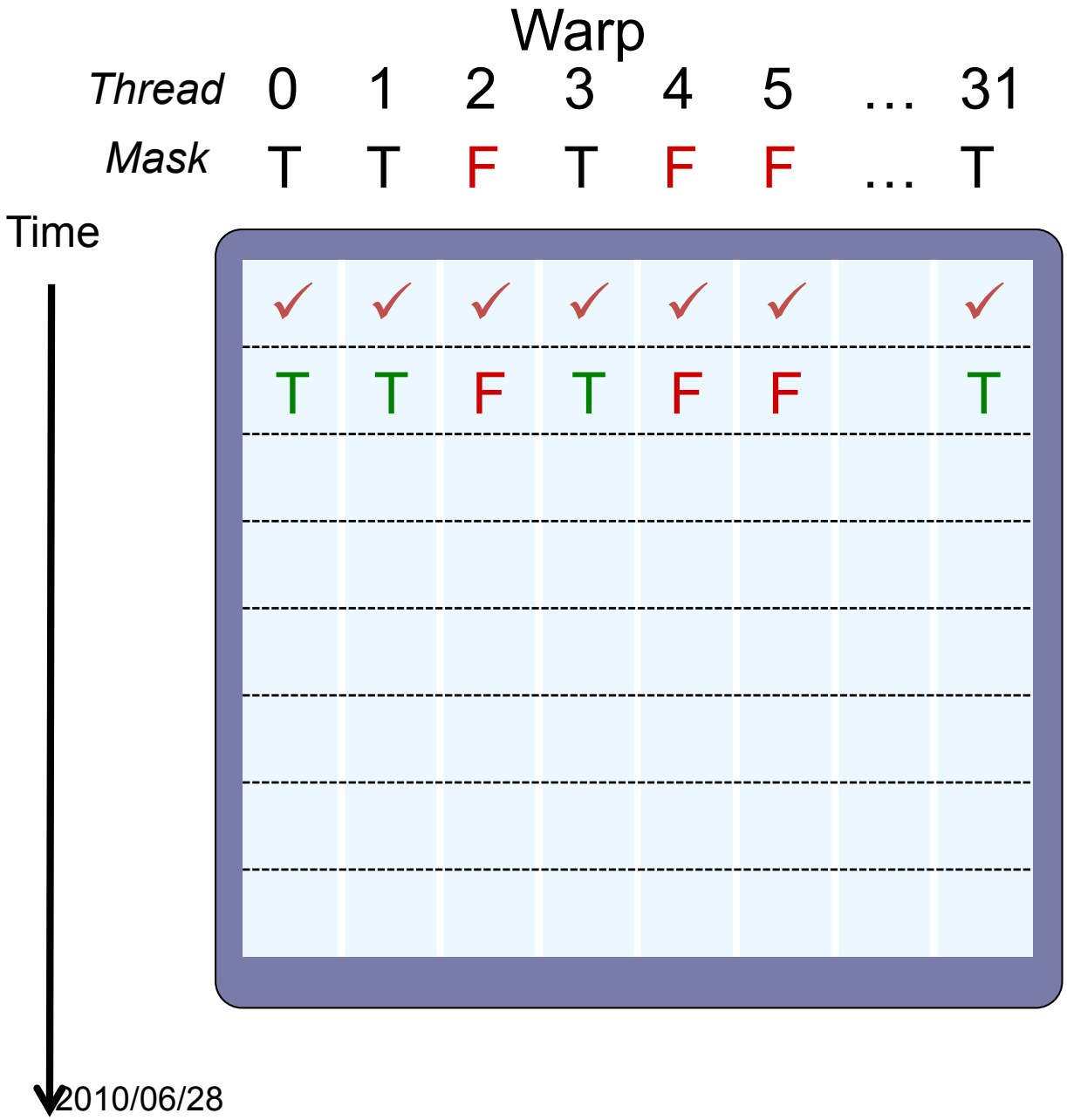

```
  x = z - k;
  if (x > 0) {
     t = y * x - a;
     s = y * y;
} else {
     t = 2 * x + a;
     s = x * y;
  }
  a[i] = t * s;
```

分岐処理



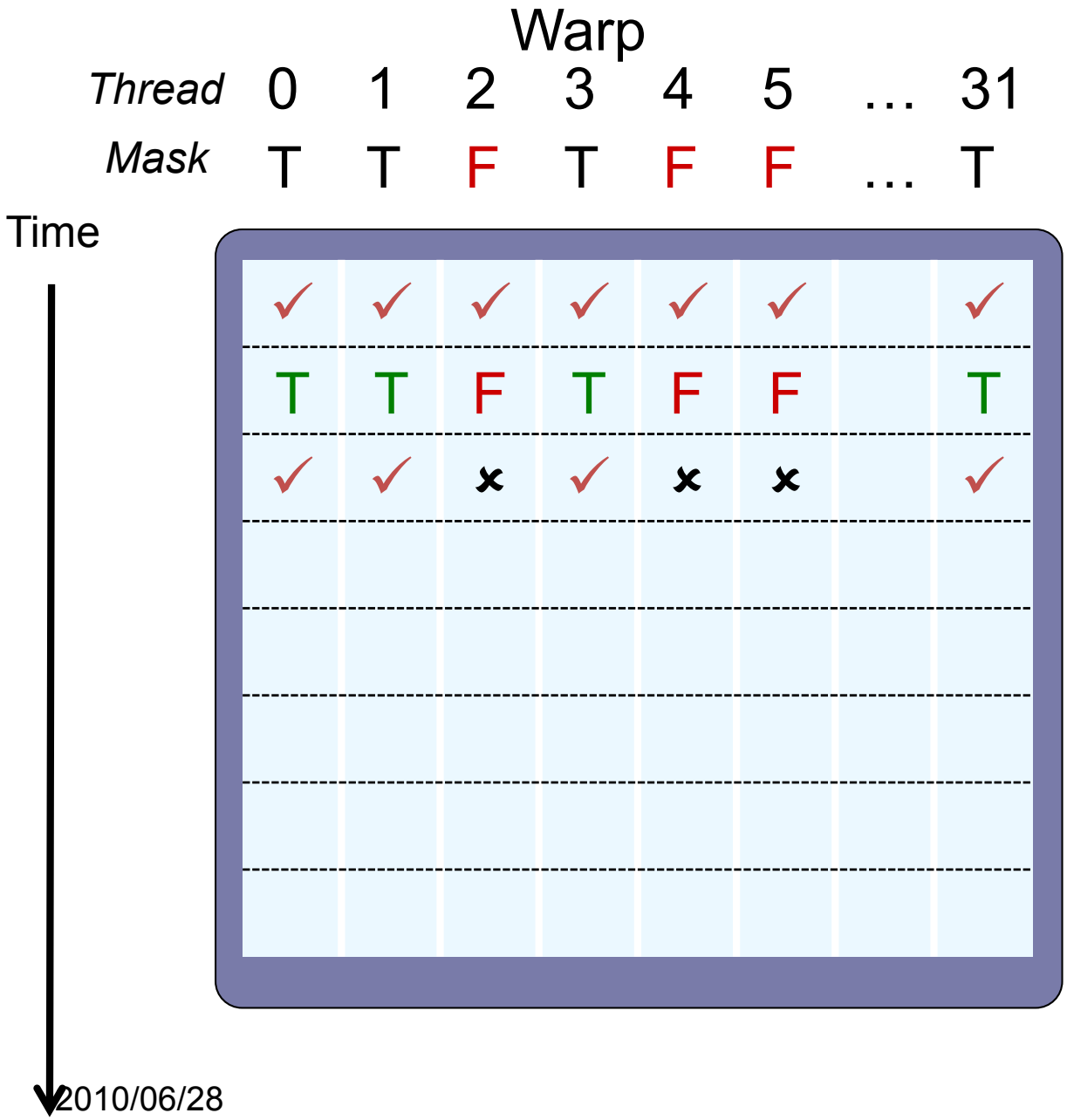

```
x = z - k;
if (x > 0) {
   t = y * x - a;
   s = y * y;
} else {
   t = 2 * x + a;
   s = x * y;
}
a[i] = t * s;
```

分岐処理



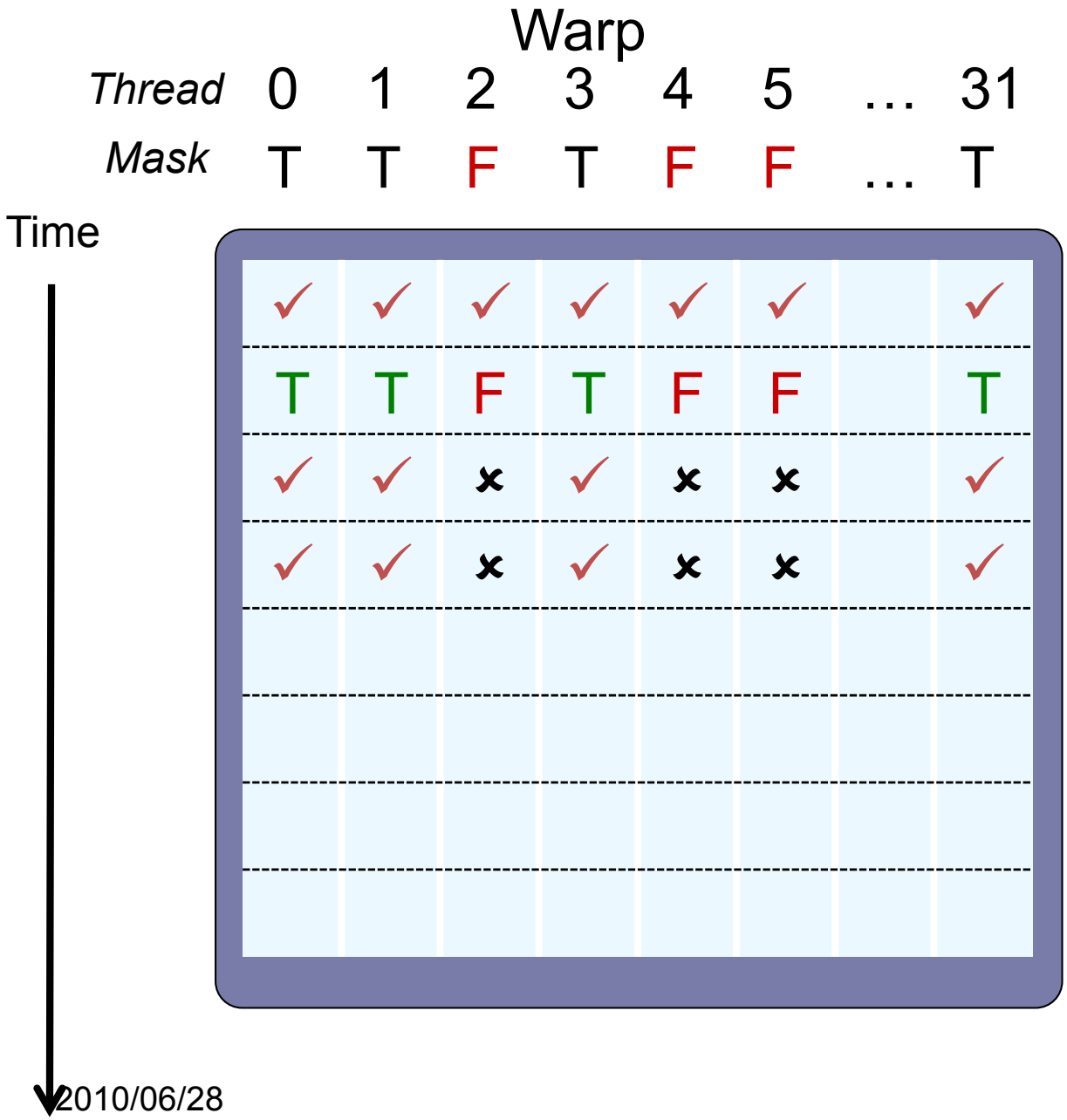

```
  x = z - k;
  if (x > 0) {
     t = y * x - a;
     s = y * y;
} else {
     t = 2 * x + a;
     s = x * y;
  }
  a[i] = t * s;
```

分岐処理



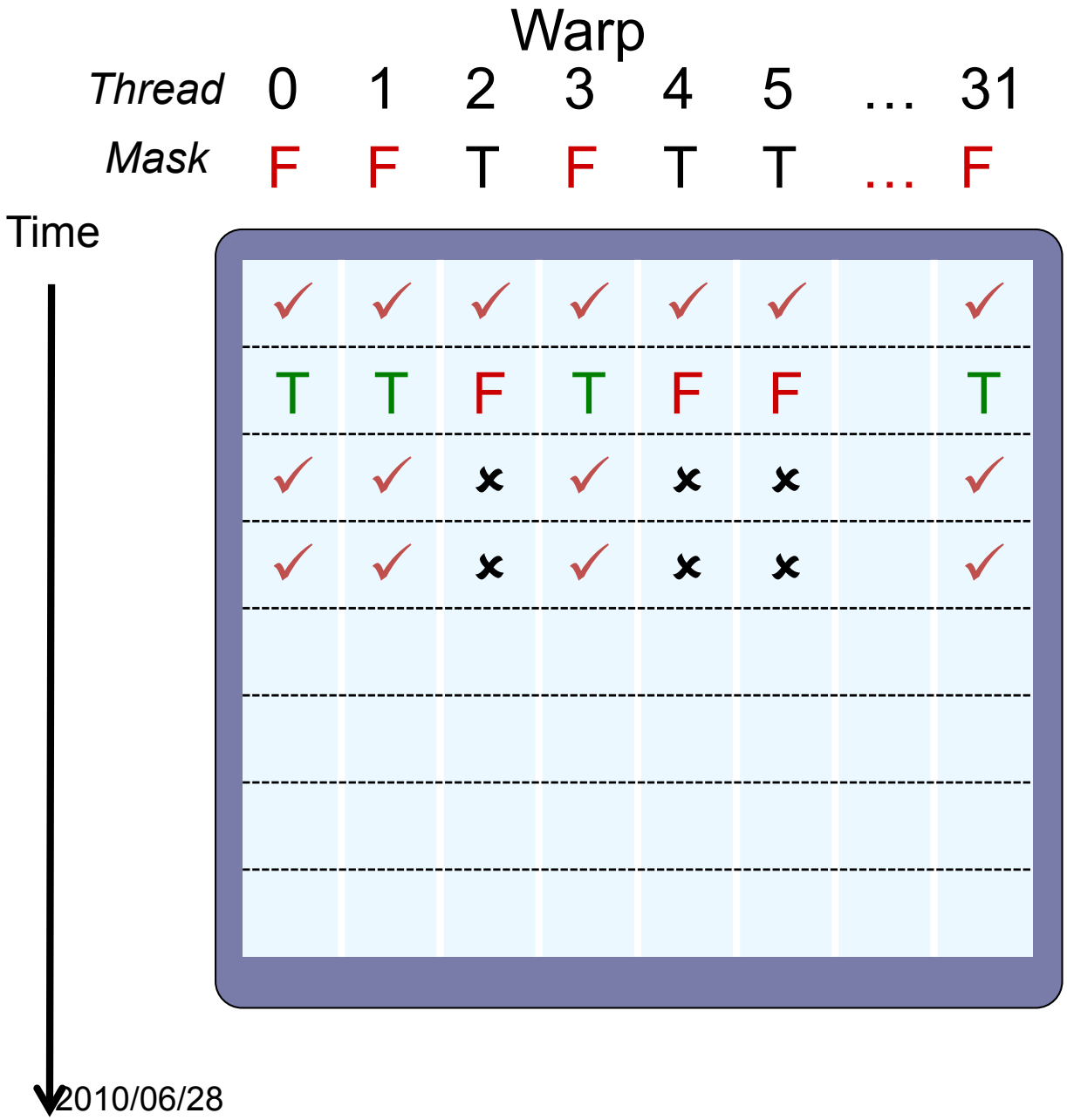

```
  x = z - k;
  if (x > 0) {
     t = y * x - a;
     s = y * y;
} else {
     t = 2 * x + a;
     s = x * y;
  }
  a[i] = t * s;
```

分岐処理



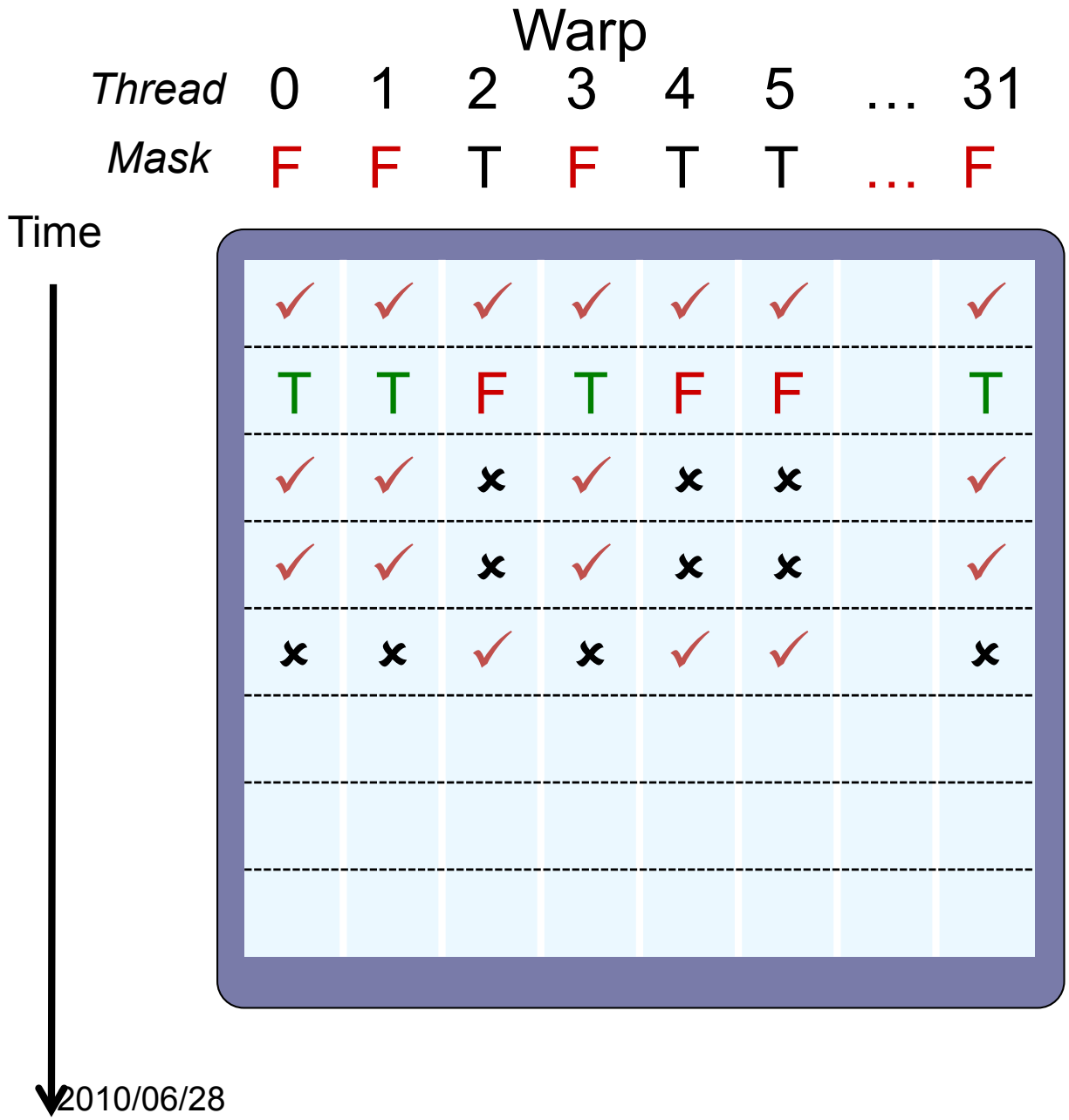

```
  x = z - k;
  if (x > 0) {
     t = y * x - a;
     s = y * y;
} else {
     t = 2 * x + a;
     s = x * y;
  }
  a[i] = t * s;
```

分岐処理



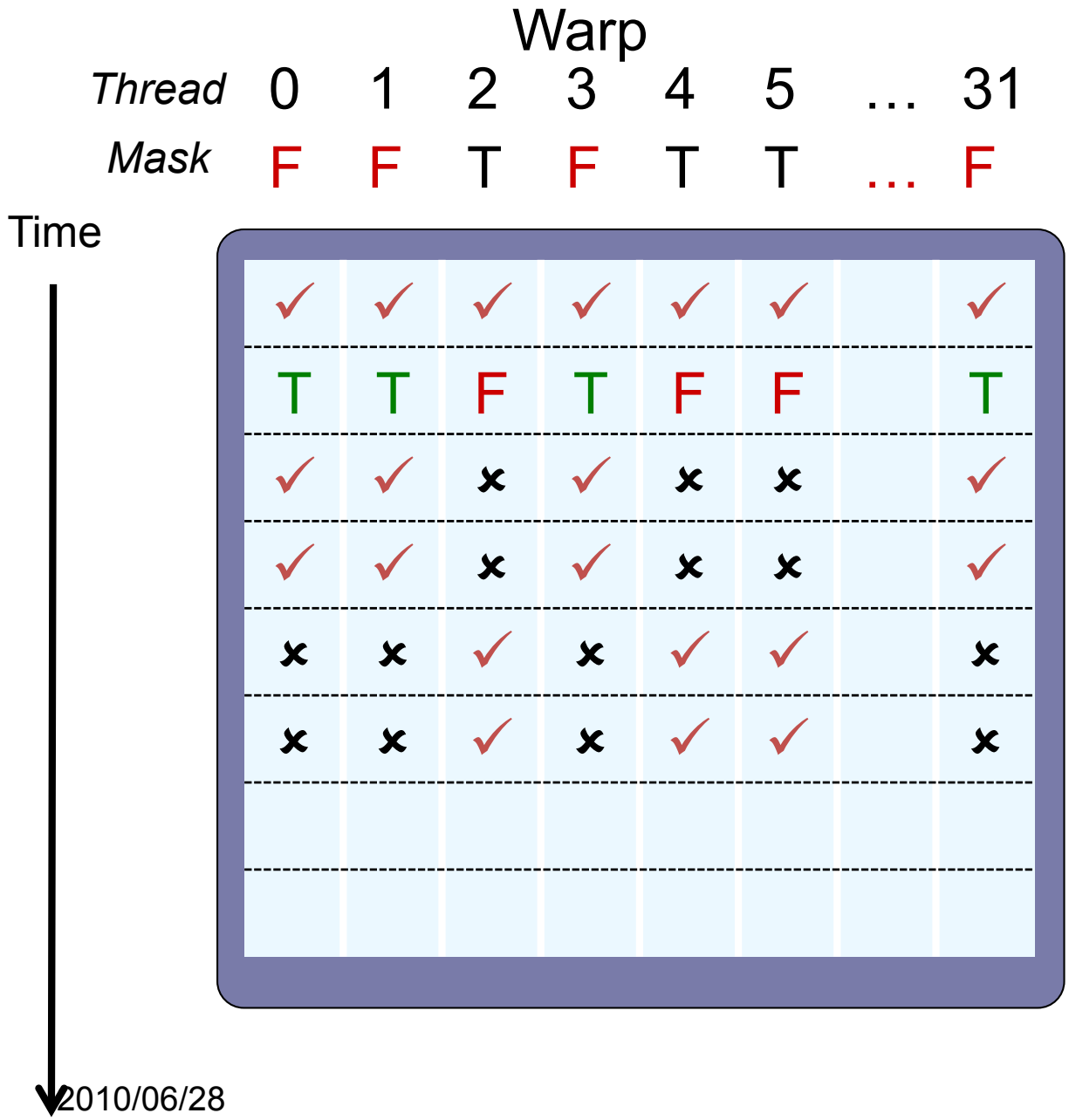

```
  x = z - k;
  if (x > 0) {
    t = y * x - a;
    s = y * y;
} else {
    t = 2 * x + a;
    s = x * y;
  }
  a[i] = t * s;
```

分岐処理



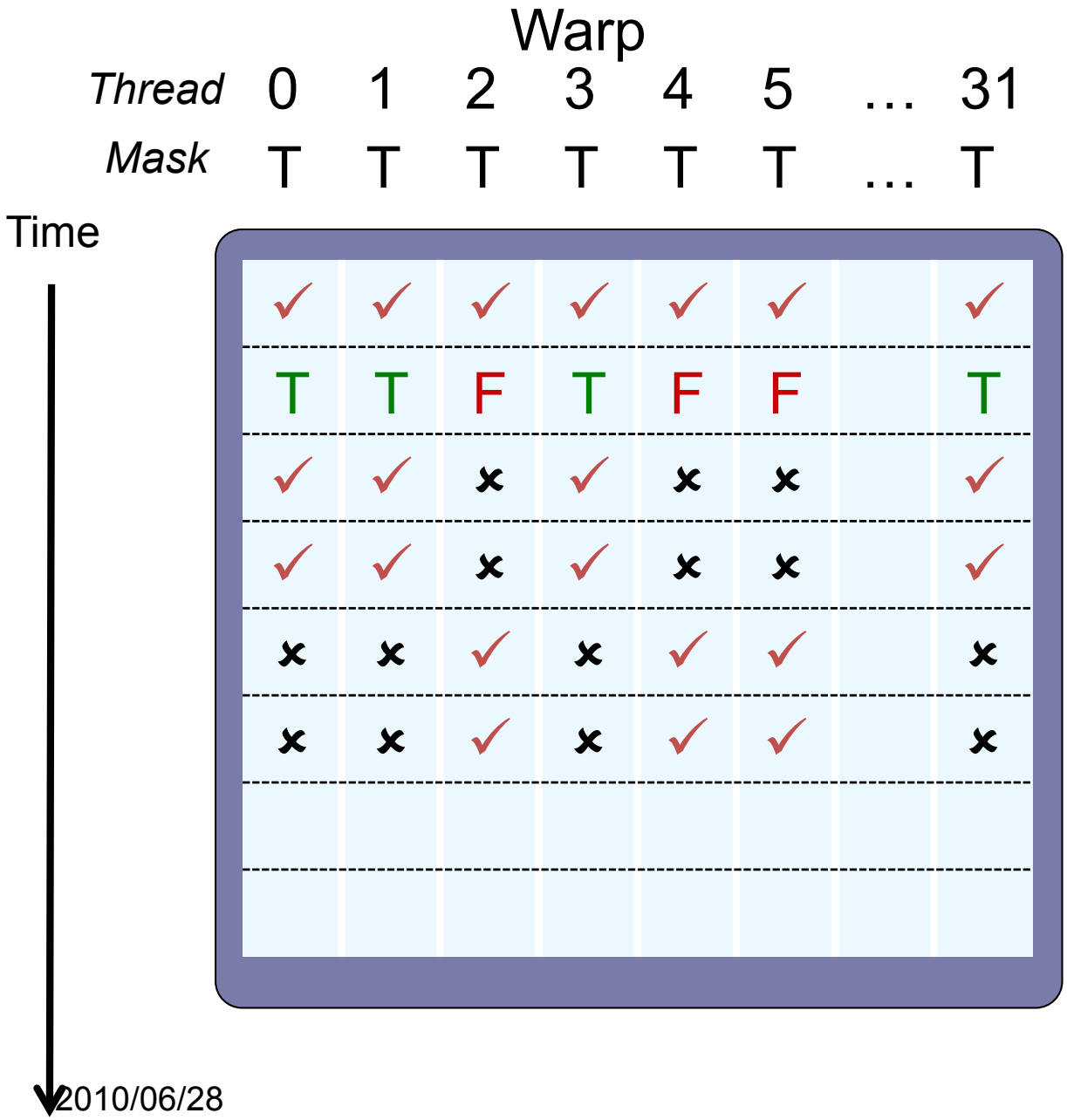

```
  x = z - k;
  if (x > 0) {
     t = y * x - a;
     s = y * y;
} else {
     t = 2 * x + a;
     s = x * y;
  }
  a[i] = t * s;
```

分岐処理



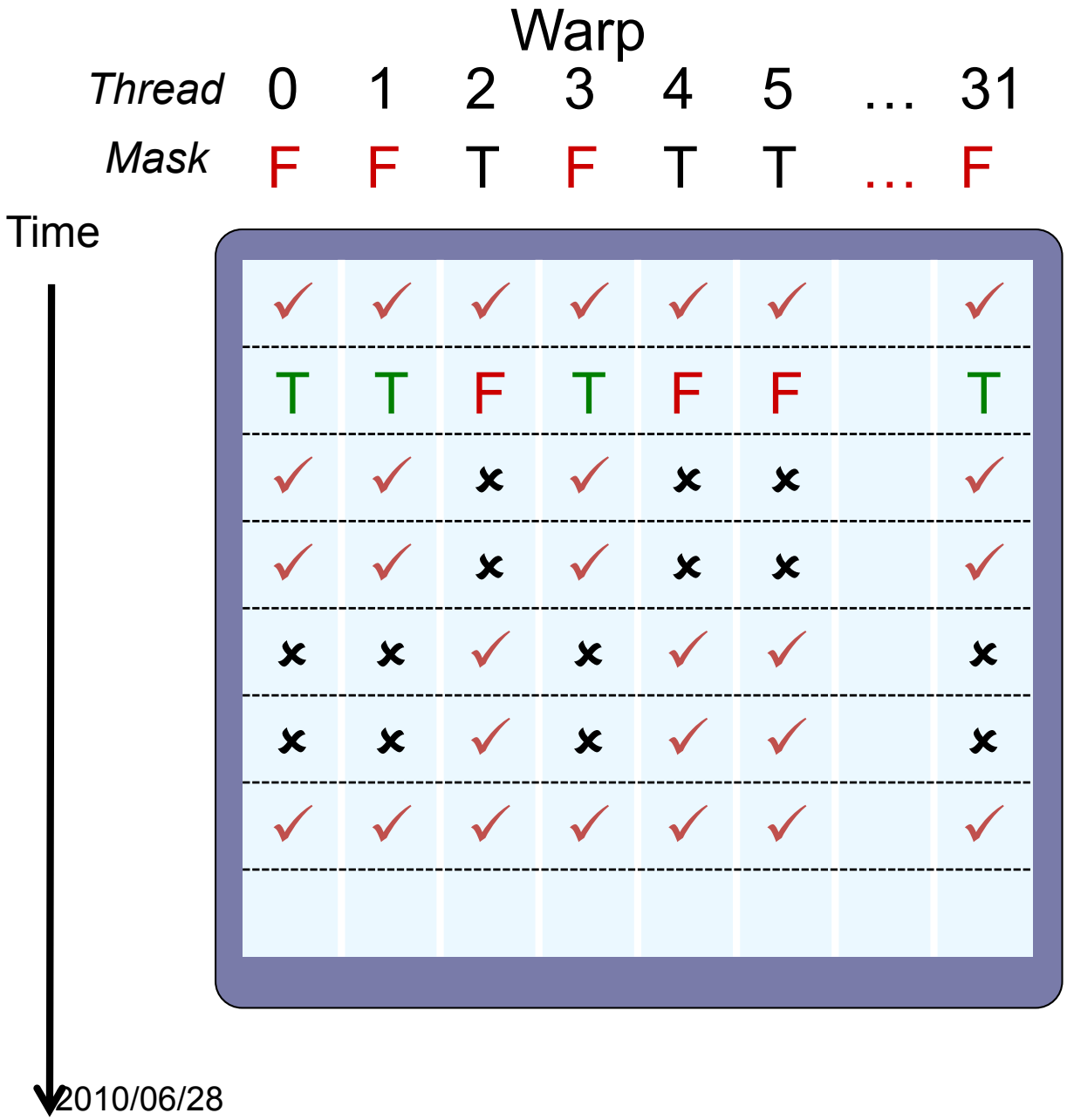

```
  x = z - k;
  if (x > 0) {
     t = y * x - a;
     s = y * y;
} else {
     t = 2 * x + a;
     s = x * y;
  }
  a[i] = t * s;
```

分岐処理




```
  x = z - k;
  if (x > 0) {
     t = y * x - a;
     s = y * y;
} else {
     t = 2 * x + a;
     s = x * y;
  }
  a[i] = t * s;
```

分岐処理




```
  x = z - k;
  if (x > 0) {
     t = y * x - a;
     s = y * y;
} else {
     t = 2 * x + a;
     s = x * y;
  }
  a[i] = t * s;
```

リダクションカーネル #2

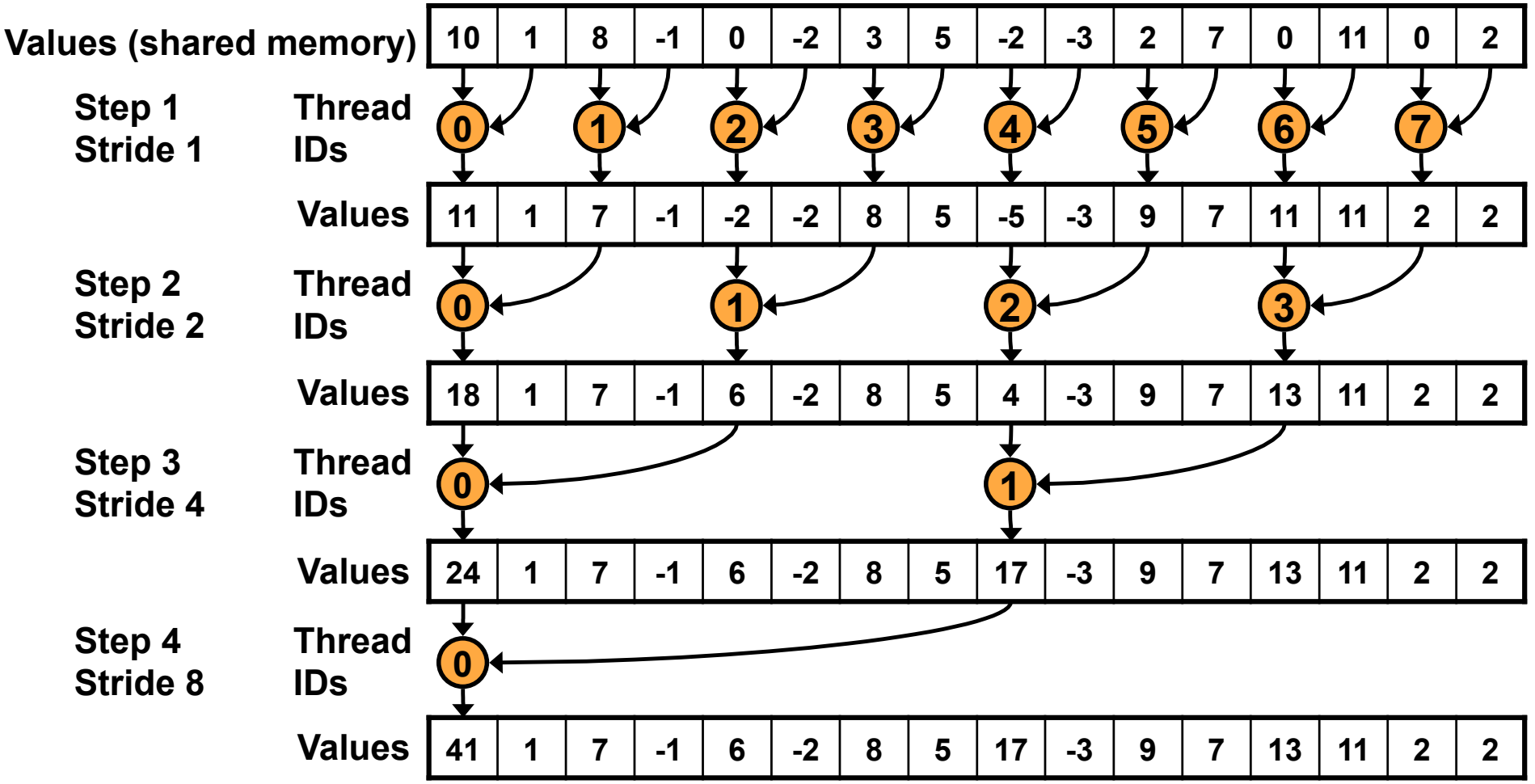
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

ダイバージェント分岐をストライドアクセスによるダイバージェント分岐の削減

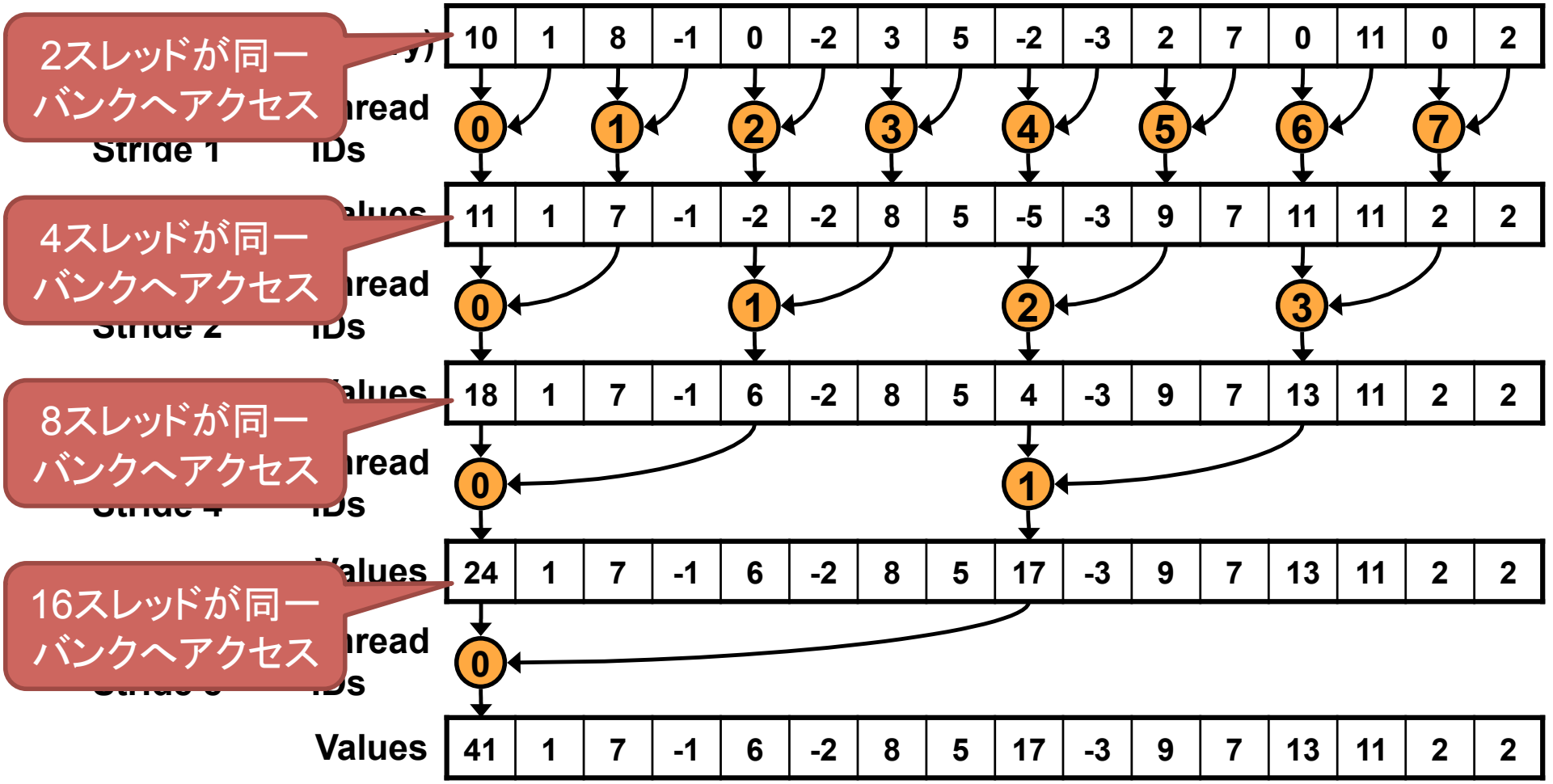
リダクション#2の動作


Values (shared memory)
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1
Stride 1
Thread IDs
0 1 2 3 4 5 6 7
Values
11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2
Step 2
Stride 2
Thread IDs
0 1 2 3
Values
18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2
Step 3
Stride 4
Thread IDs
0 1
Values
24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
Step 4
Stride 8
Thread IDs
0
Values
41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

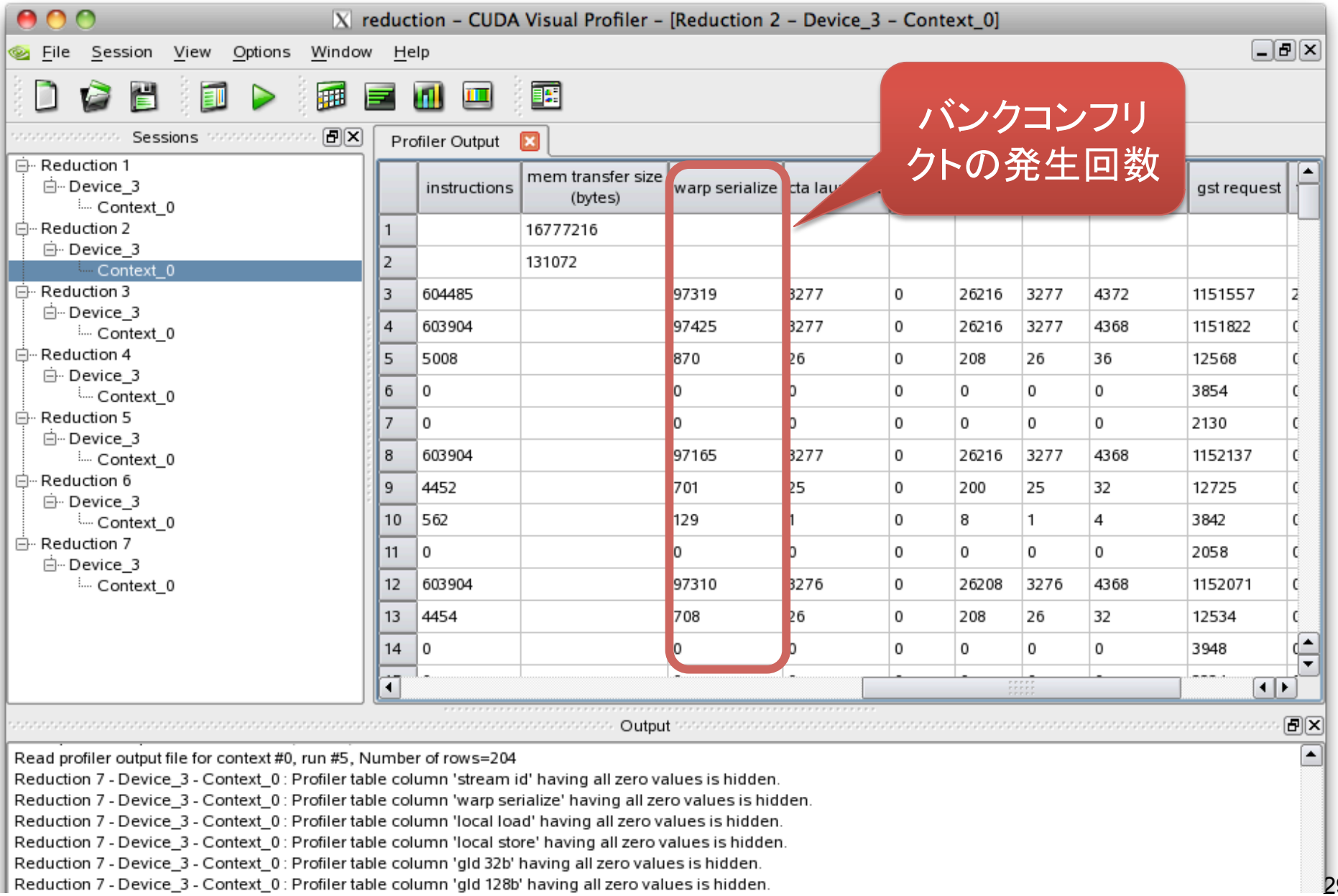
最適化結果

	実行時間	バンド幅	% MAX	効果	効果（全体）
Kernel 1	**3.51 ms**	**4.77 GB/s**	**4.6 %**		
Kernel 2	**1.62 ms**	**10.4 GB/s**	**10.1 %**	**2.2x**	**2.2x**

リダクション#2の性能低下要因



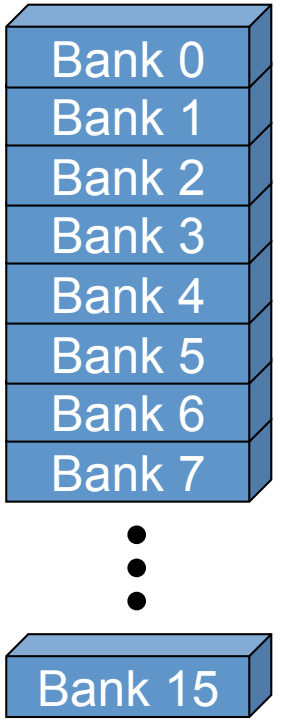
プロファイラによる確認



	instructions	mem transfer size (bytes)	warp serialize	cta lau				gst request	
1		16777216							
2		131072							
3	604485		97319	3277	0	26216	3277	4372	1151557
4	603904		97425	3277	0	26216	3277	4368	1151822
5	5008		870	26	0	208	26	36	12568
6	0		0	0	0	0	0	0	3854
7	0		0	0	0	0	0	0	2130
8	603904		97165	3277	0	26216	3277	4368	1152137
9	4452		701	25	0	200	25	32	12725
10	562		129	1	0	8	1	4	3842
11	0		0	0	0	0	0	0	2058
12	603904		97310	3276	0	26208	3276	4368	1152071
13	4454		708	26	0	208	26	32	12534
14	0		0	0	0	0	0	0	3948

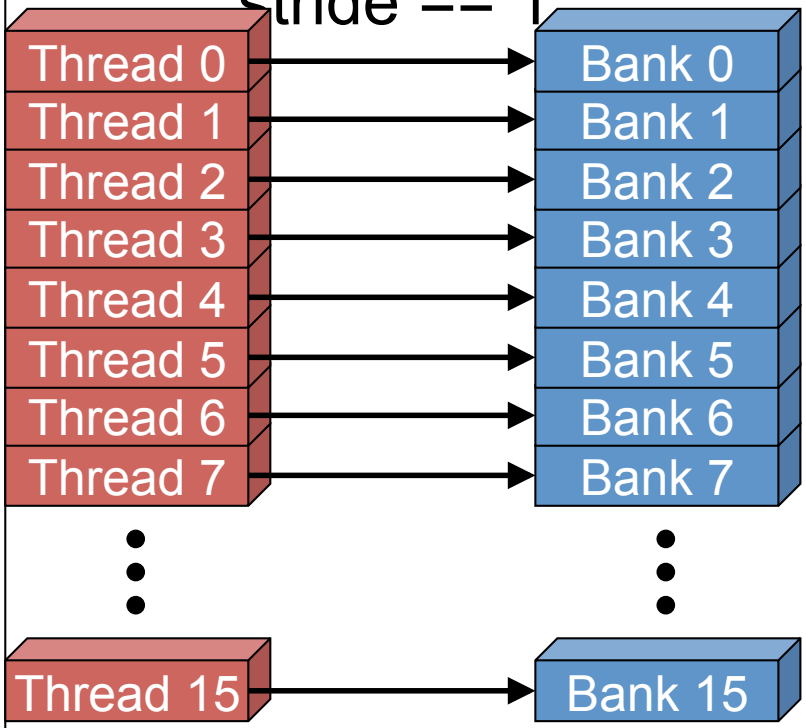
メモリバンクとバンクコンフリクト

- GPUのようなマルチスレッドアーキテクチャでは複数スレッドが同時にメモリにアクセス
  - メモリが一度に1アクセスしか処理できない場合、逐次処理に→ボトルネックになりがち
- CUDA共有メモリではメモリを16バンクに分割
  - 各バンクは連続したアドレスに対応
  - 16スレッドが別個のアドレスにアクセス→16バンクすべてを使うことにより並列処理
  - 複数スレッドが同一アドレスにアクセス→アクセス先バンクの衝突（バンクコンフリクト）

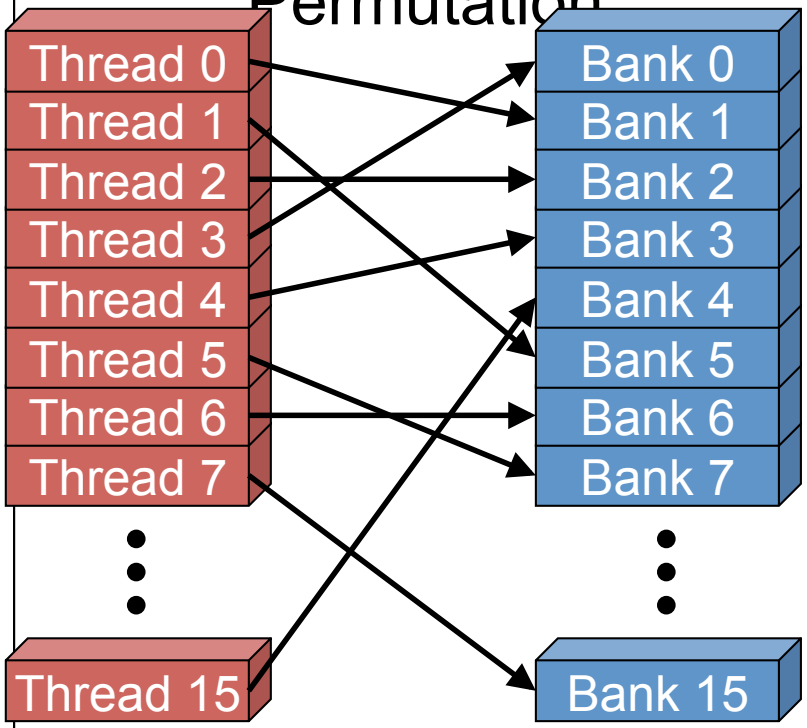


バンクコンフリクトが起きない例

- No Bank Conflicts
  - Linear addressing stride == 1

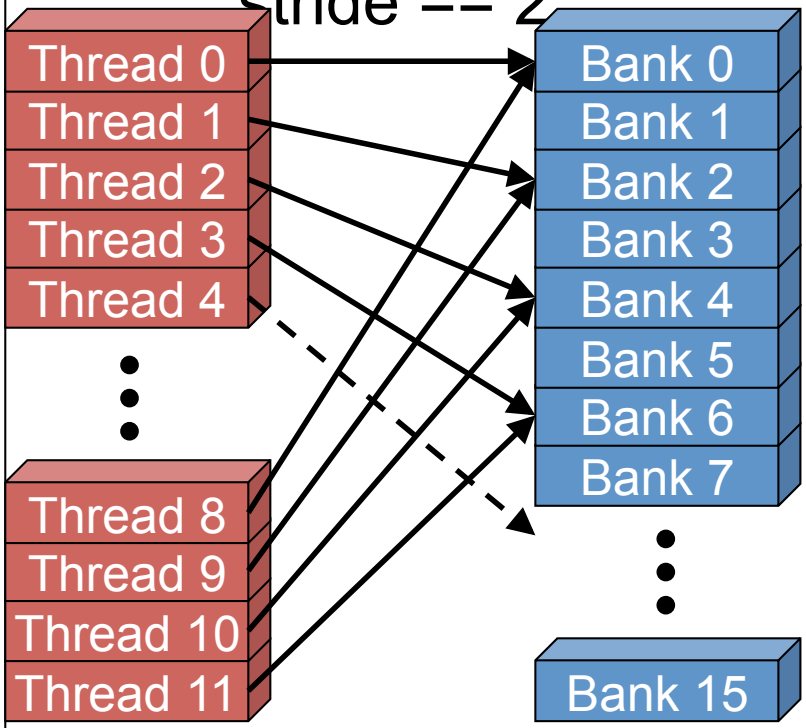


- No Bank Conflicts
  - Random 1:1 Permutation

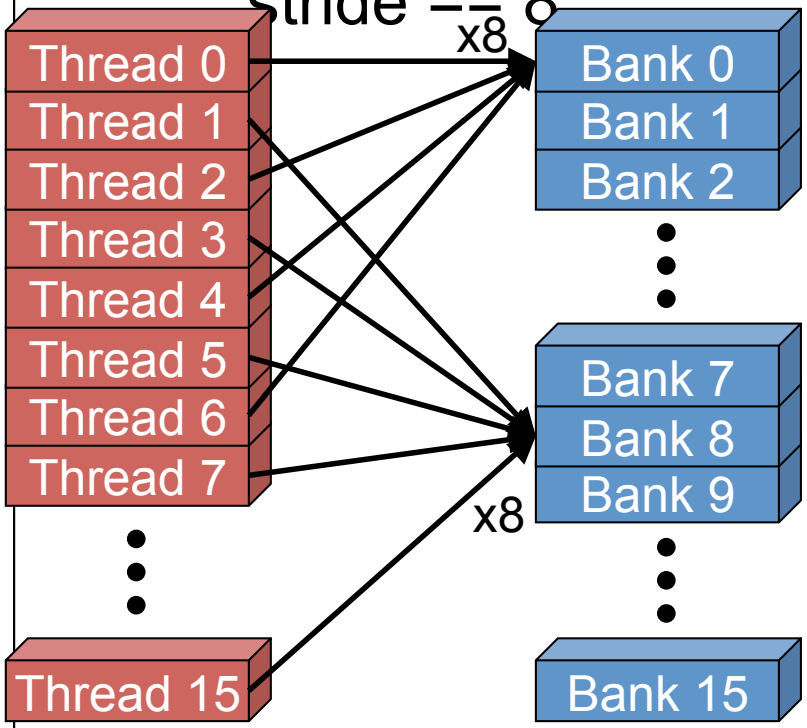


バンクコンフリクトが起きる例

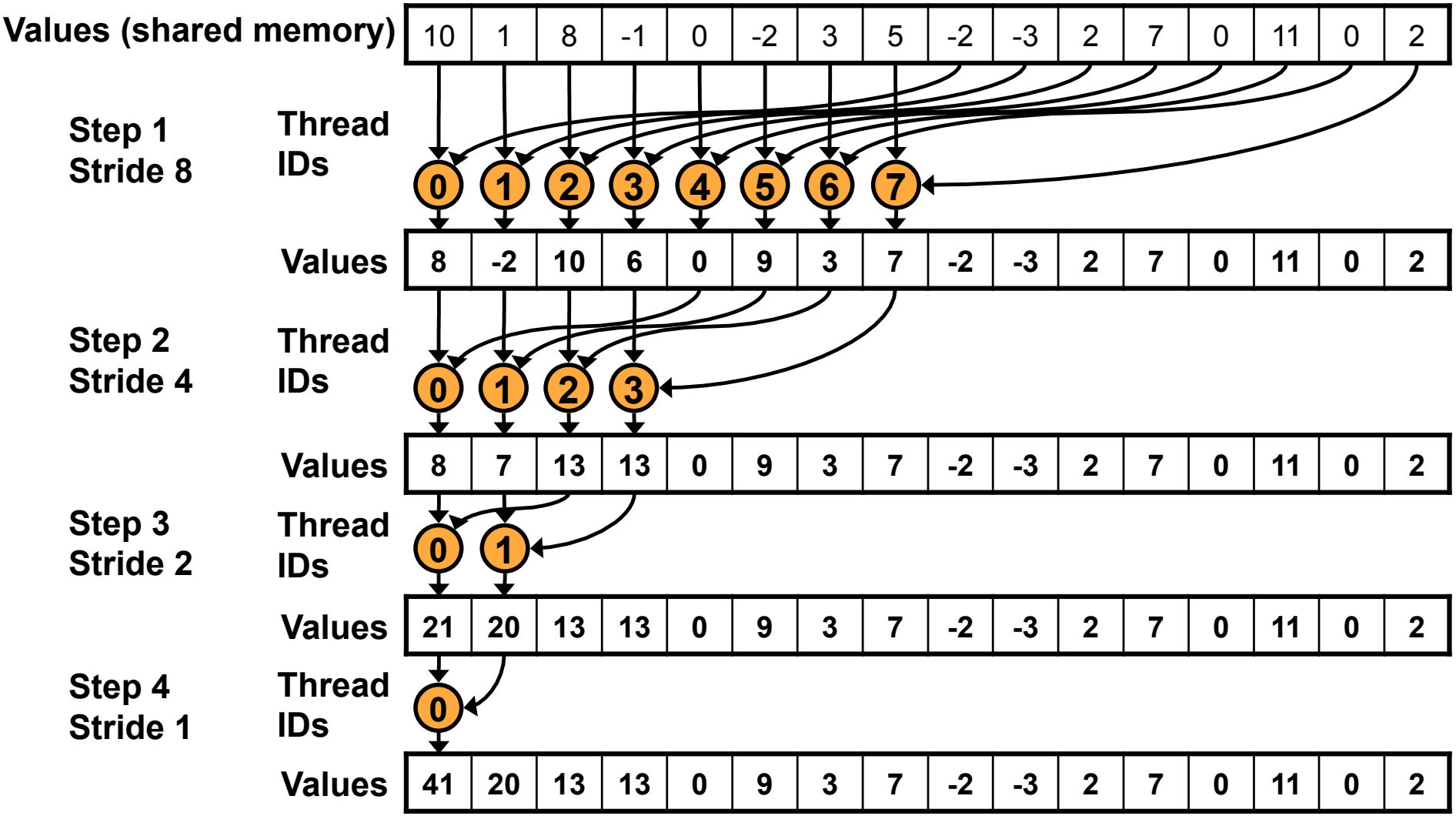
- 2-way Bank Conflicts
  - Linear addressing stride == 2



- 8-way Bank Conflicts
  - Linear addressing stride == 8



バンクコンフリクトの除去



連続したスレッドが連続したバンクへアクセス

リダクション #3

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

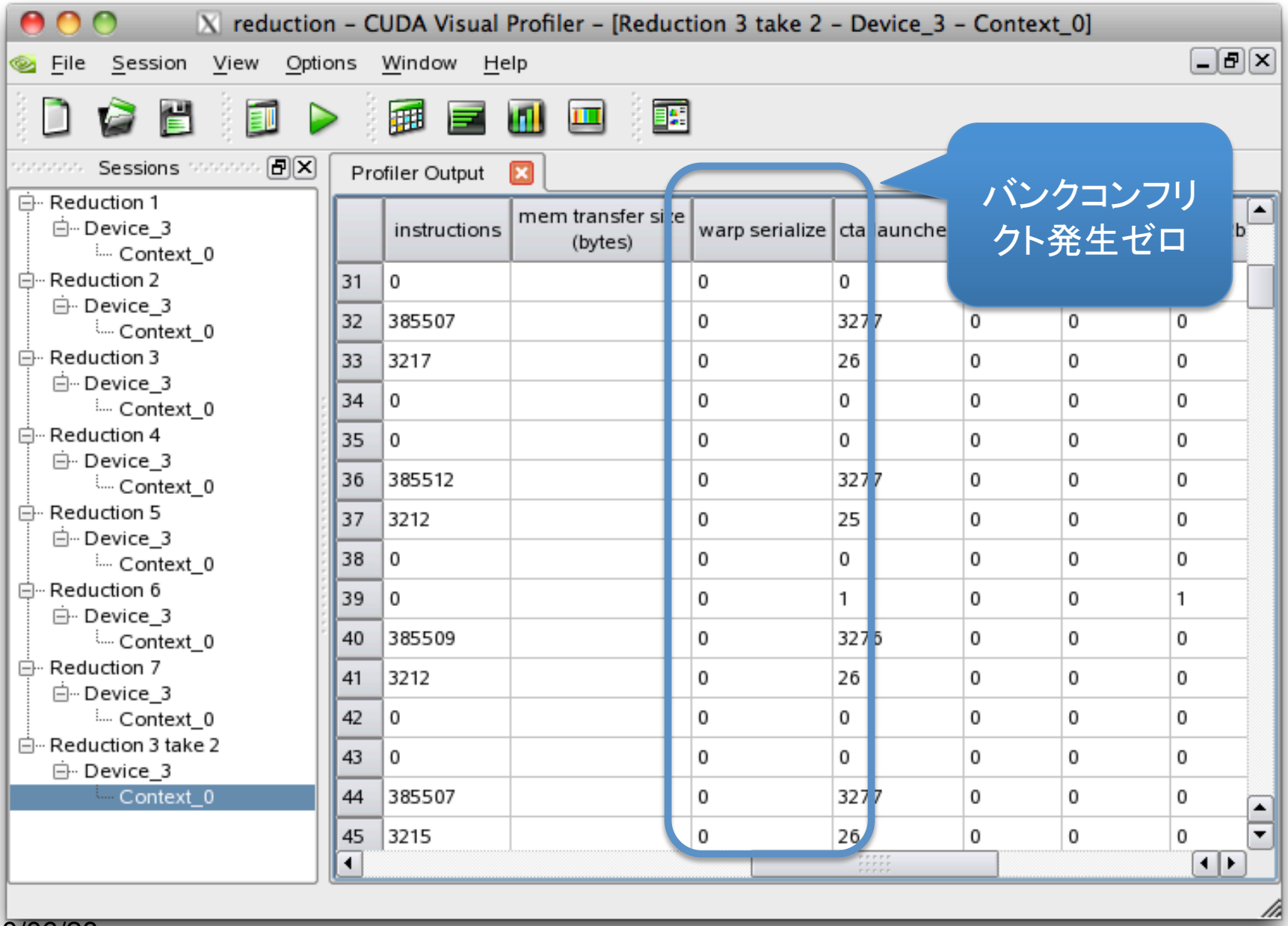
後半半分を前半半分に足し込む処理へ変更

→ バンクコンフリクト発生なし

最適化結果

	実行時間	バンド幅	% MAX	効果	効果（全体）
Kernel 1	**3.51 ms**	**4.77 GB/s**	**4.6 %**		
Kernel 2	**1.62 ms**	**10.4 GB/s**	**10.1 %**	**2.2x**	**2.2x**
Kernel 3	**0.81 ms**	**20.7 GB/s**	**20.2 %**	**2.0x**	**4.3x**

プロファイラによる確認



リダクション#3の性能低下の要因

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

ループ実行1回目の際、スレッドブロック内後半のスレッドはなにもしていない

後半スレッドはグローバルメモリから担当要素を共有メモリへロードするだけ

前半スレッドが後半スレッドの分も読めば、後半スレッドを起動しなくてOK

リダクション#4： 1スレッド

```
// 各スレッドが1要素をグローバルメモリより読み込み
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

```
// グローバルメモリより2要素読み込み、スレッド内でリダクション
// その結果を共有メモリへ書き込み
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

最適化結果

	実行時間	バンド幅	% MAX	効果	効果（全体）
Kernel 1	**3.51 ms**	**4.77 GB/s**	**4.6 %**		
Kernel 2	**1.62 ms**	**10.4 GB/s**	**10.1 %**	**2.2x**	**2.2x**
Kernel 3	**0.81 ms**	**20.7 GB/s**	**20.2 %**	**2.0x**	**4.3x**
Kernel 4	**0.47 ms**	**36.0 GB/s**	**35.2 %**	**1.7x**	**7.5x**

まだまだ35%しかでてない

最適化：ループアンローリング

```
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
 }

if (tid < 32) {
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid +  8];
        sdata[tid] += sdata[tid +  4];
        sdata[tid] += sdata[tid +  2];
        sdata[tid] += sdata[tid +  1];
}
```

最後6回の繰り返しを展開

- スレッドブロック内の最初のワープのみ動作
- ワープ内スレッドはすべて順々に動作

→ __syncthreads()の省略可

最適化結果

	実行時間	バンド幅	% MAX	効果	効果（全体）
Kernel 1	**3.51 ms**	**4.77 GB/s**	**4.6 %**		
Kernel 2	**1.62 ms**	**10.4 GB/s**	**10.1 %**	**2.2x**	**2.2x**
Kernel 3	**0.81 ms**	**20.7 GB/s**	**20.2 %**	**2.0x**	**4.3x**
Kernel 4	**0.47 ms**	**36.0 GB/s**	**35.2 %**	**1.7x**	**7.5x**
Kernel 5	**0.28 ms**	**58.1 GB/s**	**57.0 %**	**1.7x**	**12.5x**

さらなるループ最適化

```
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
 }

if (tid < 32) {
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid +  8];
        sdata[tid] += sdata[tid +  4];
        sdata[tid] += sdata[tid +  2];
        sdata[tid] += sdata[tid +  1];
}
```

ここもアンロールできる？

ブロックサイズは最大でも512

s は必ず64, 128, 256のいずれか

リダクション#6

```
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
}
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
}
if (blockSize >= 128) {
    if (tid <  64)  { sdata[tid] += sdata[tid +   64]; } __syncthreads();
}

if (tid < 32) {
    if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
    if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
    if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=   8)  sdata[tid] += sdata[tid +  4];
    if (blockSize >=   4)  sdata[tid] += sdata[tid +  2];
    if (blockSize >=   2)  sdata[tid] += sdata[tid +  1];
}
```

さらに、ブロックサイズはコンパイル時に定数にすることで、赤字の分岐をコンパイル時に処理可能

最適化結果

	実行時間	バンド幅	% MAX	効果	効果（全体）
Kernel 1	**3.51 ms**	**4.77 GB/s**	**4.6 %**		
Kernel 2	**1.62 ms**	**10.4 GB/s**	**10.1 %**	**2.2x**	**2.2x**
Kernel 3	**0.81 ms**	**20.7 GB/s**	**20.2 %**	**2.0x**	**4.3x**
Kernel 4	**0.47 ms**	**36.0 GB/s**	**35.2 %**	**1.7x**	**7.5x**
Kernel 5	**0.28 ms**	**58.1 GB/s**	**57.0 %**	**1.7x**	**12.5x**
Kernel 6	**0.25 ms**	**66.2 GB/s**	**65.0 %**	**1.13x**	**14.0x**

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
    extern __shared__ int sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <   64) { sdata[tid] += sdata[tid +   64]; } __syncthreads(); }

    if (tid < 32) {
        if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
        if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
        if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
        if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
        if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
        if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
    }

    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

1スレッドあたり複数の要素をリダクションさせることで、さらなる最適化

最適化結果

	実行時間	バンド幅	% MAX	効果	効果（全体）
Kernel 1	**3.51 ms**	**4.77 GB/s**	**4.6 %**		
Kernel 2	**1.62 ms**	**10.4 GB/s**	**10.1 %**	**2.2x**	**2.2x**
Kernel 3	**0.81 ms**	**20.7 GB/s**	**20.2 %**	**2.0x**	**4.3x**
Kernel 4	**0.47 ms**	**36.0 GB/s**	**35.2 %**	**1.7x**	**7.5x**
Kernel 5	**0.28 ms**	**58.1 GB/s**	**57.0 %**	**1.7x**	**12.5x**
Kernel 6	**0.25 ms**	**66.2 GB/s**	**65.0 %**	**1.13x**	**14.0x**
Kernel 7	**0.22 ms**	**76.1 GB/s**	**74.6 %**	**1.14x**	**16.0x**

CUDAプロファイラ

- CUDAツールキットに付属
- CUIとGUIあり
- CUIの使い方
  - 環境変数 CUDA_PROFILE を1にセット
  - 詳細はツールキット付属ドキュメント CUDA_Profiler.txtを参照してください
- GUIの使い方
  - sshログインの際に-Yオプションを追加

```
$ export LD_LIBRARY_PATH=/opt/cuda2.3/cudaprof/bin:$LD_LIBRARY_PATH
$ export PATH=/opt/cuda2.3/cudaprof/bin:$PATH
$ cudaprof
```