Complementi di CUDA
Riduzione Parallela

Sistemi Digitali, Modulo 2

A.A. 2024/2025

Fabio Tosi, Università di Bologna

Il Problema della Riduzione: Somma di un Array

Obiettivo: Calcolare la **somma di un array di N elementi** in modo efficiente sfruttando il parallelismo.

Soluzione Sequenziale

```
int sum = 0;
for (int i = 0; i < N; i++)
 sum += array[i];
```

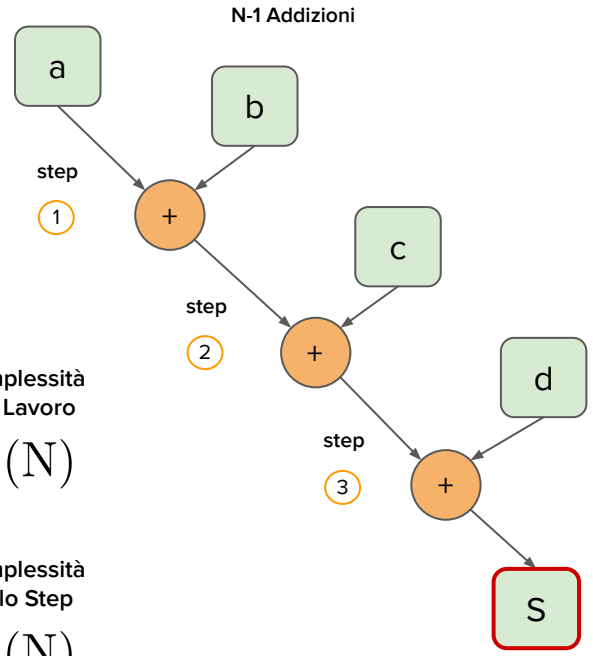
- La soluzione sequenziale ha complessità **O(N)**, inefficiente per array di grandi dimensioni.

Soluzione Parallela

- **Suddividere il problema** in sotto-problemi indipendenti eseguiti in parallelo da diversi thread.
- **Fondamento Matematico**: Proprietà *associativa* e *commutativa* dell'addizione. Gli elementi dell'array possono essere sommati in qualsiasi ordine.
- **Fasi della Riduzione Parallela:**
  - **Partizionamento (Decomposition)**: Dividere l'array in *chunk* più piccoli.
  - **Somma Parziale (Local Reduction)**: Ogni thread calcola la somma degli elementi del proprio chunk.
  - **Somma Finale (Global Reduction)**: Combinare le somme parziali per ottenere la somma totale.

Il Problema della Riduzione: Somma di un Array

Riduzione Sequenziale



Complessità del Lavoro

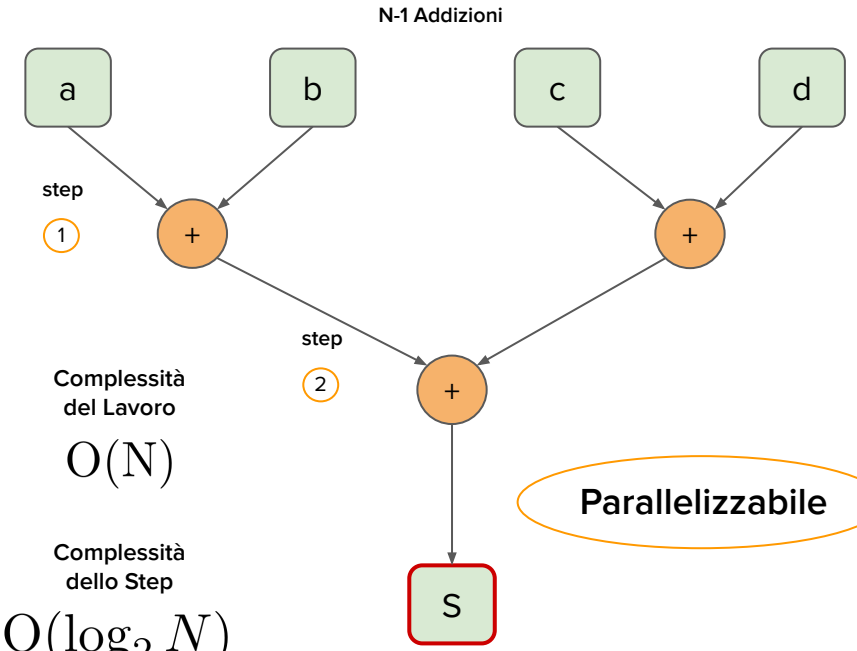
$$O(N)$$

Complessità dello Step

$$O(N)$$

$$\Big(((a+b)+c)+d\Big)$$

Riduzione Iterativa a Coppie



Complessità del Lavoro

$$O(N)$$

Parallelizzabile

Complessità dello Step

$$O(\log_2 N)$$

$$(a+b)+(c+d)$$

Implementazione Iterativa a Coppie per la Riduzione

Somma a Coppie:

- Ad ogni passo, l'algoritmo processa gli elementi **a coppie**.
- La **somma** di una coppia produce un **risultato parziale**.
- I **risultati parziali** vengono **memorizzati** ***in-place*** nell'array originale.
- Questi nuovi valori diventano **input** per l'iterazione successiva.

Dimezzamento Iterativo

- Ad ogni iterazione, il numero di valori di input **si dimezza**.
- Il processo continua finché non si ottiene un singolo valore finale (**somma finale)**.

Assunzioni Iniziali

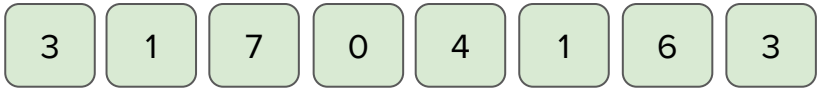
- Tratteremo array con dimensioni **potenze di due** per semplicità.
- Per N arbitrario? **Padding** con zeri fino alla potenza di 2, **scomposizione** in potenze di 2, etc.

Tipi di Implementazione a Coppie

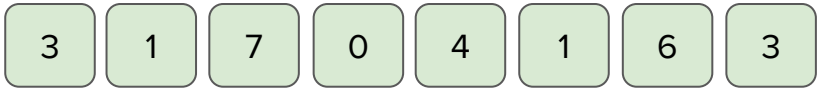
- A seconda di dove vengono memorizzati gli elementi di output *in-place* ad ogni iterazione, le implementazioni a coppie si classificano in:
    - **Neighbored Pair**: Gli elementi vengono accoppiati con il loro vicino immediato a distanza **stride**.
    - **Interleaved Pair**: Gli elementi accoppiati sono separati da un determinato passo (**stride**).

Implementazione Iterativa a Coppie per la Riduzione

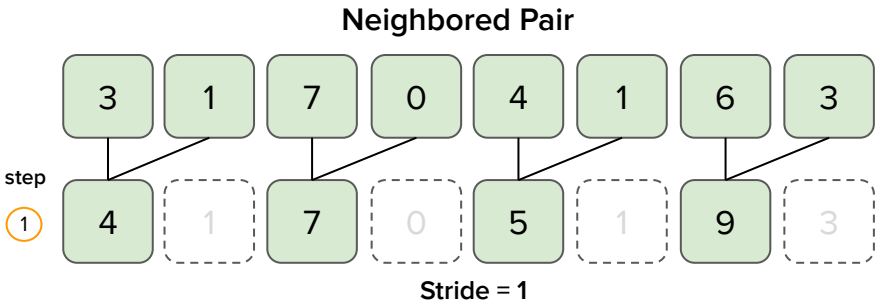
Neighbored Pair

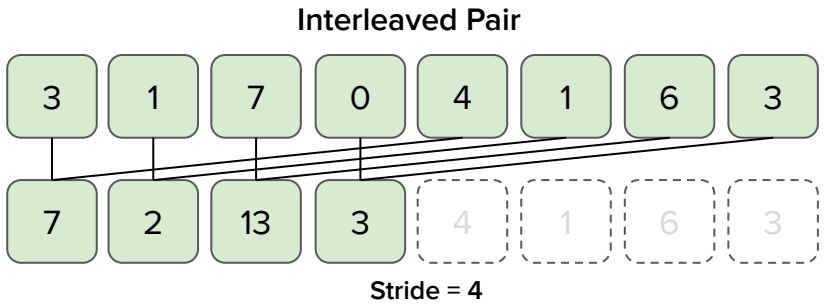


Interleaved Pair

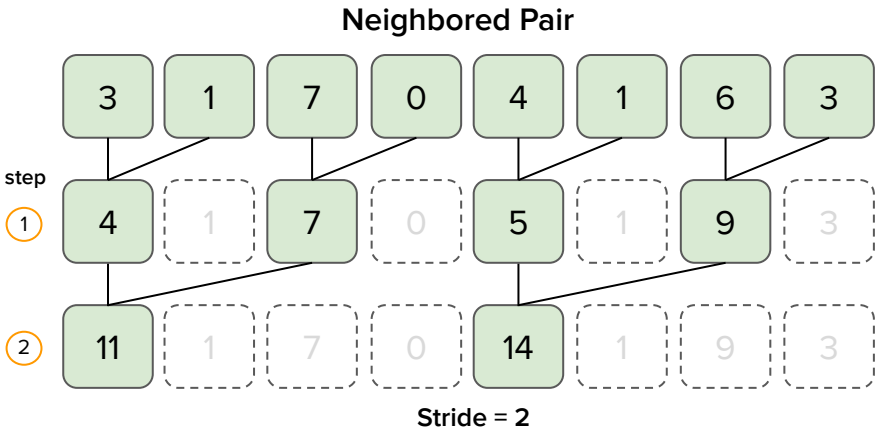


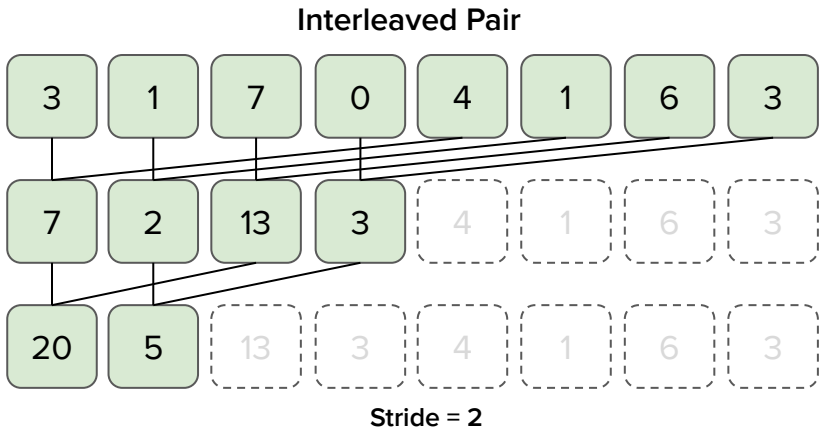
Implementazione Iterativa a Coppie per la Riduzione



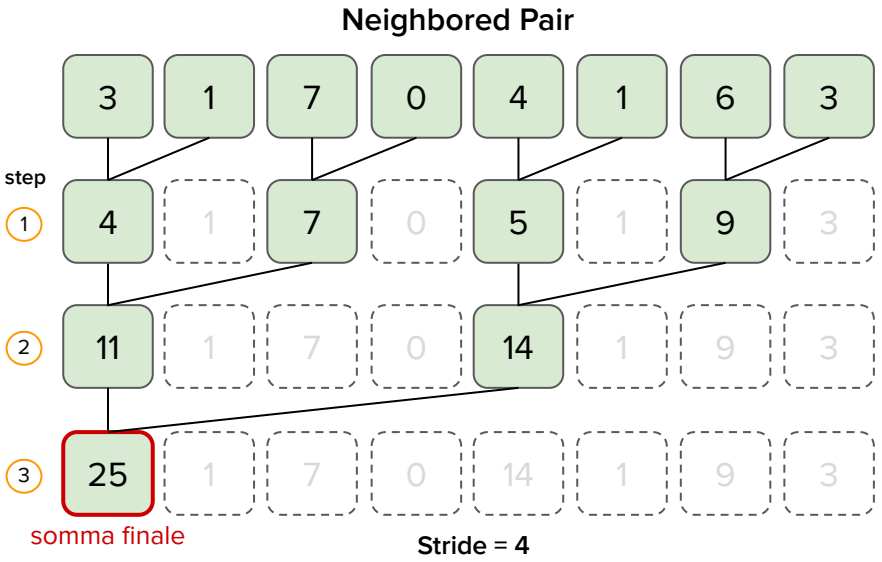


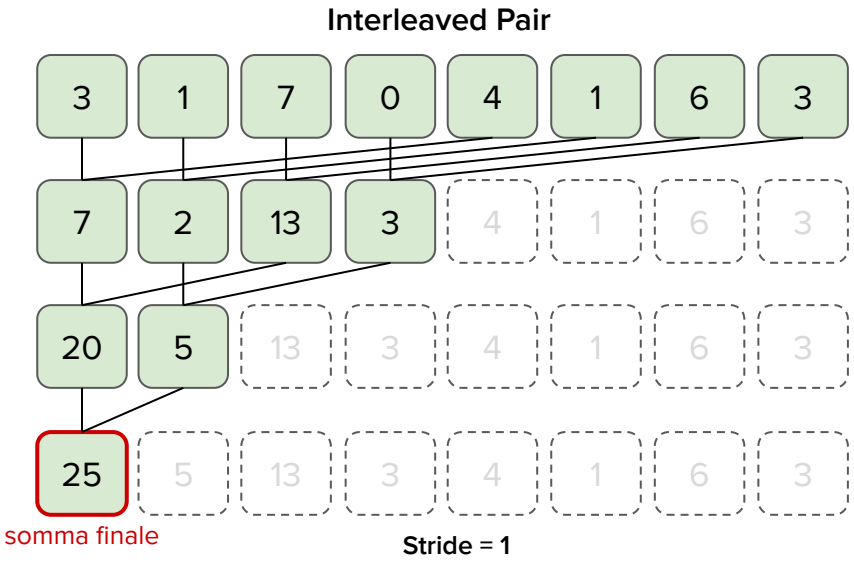
Implementazione Iterativa a Coppie per la Riduzione



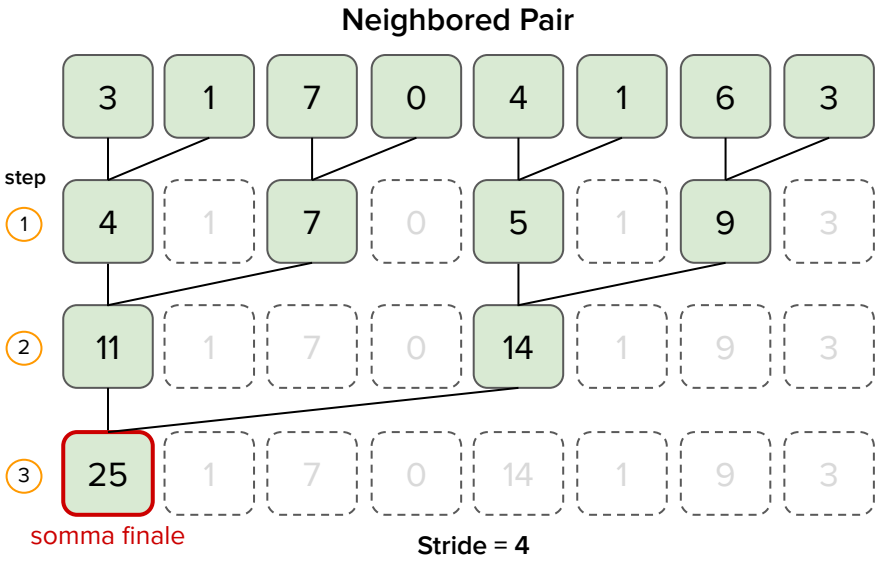


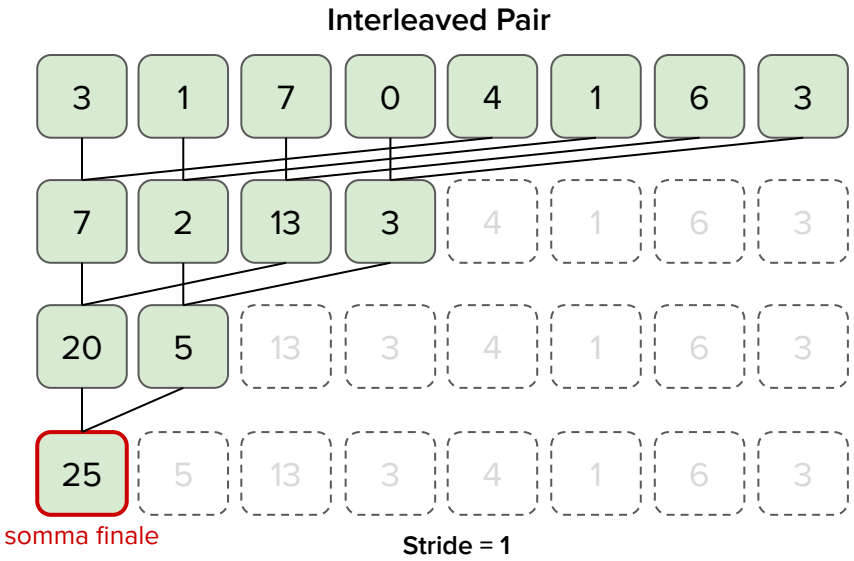
Implementazione Iterativa a Coppie per la Riduzione





Implementazione Iterativa a Coppie per la Riduzione





- **Numero di somme (N-1):** Indipendentemente dall'approccio, per sommare N elementi, sono necessarie N-1 operazioni di somma totali. Ad ogni step, $N/(2^{step})$ operazioni.
- **Numero di passi (log_2N):** Ad ogni passo, il numero di elementi viene dimezzato. Quindi, per un array di N elementi, servono $\mathbf{log_2N}$ passi per arrivare ad un singolo elemento.
- **Differenze**: La differenza principale tra i due approcci sta nel modo in cui gli elementi vengono accoppiati e nella distribuzione del lavoro tra i thread, non nel numero totale di operazioni o passi.

Implementazione Iterativa a Coppie per la Riduzione

Implementazione Ricorsiva dell'Approccio Neighbored Pair - Funzione C Standard

```c
int recursiveReduceNeighbored(int *data, int const size, int stride) {
  // Caso base: quando stride è maggiore o uguale a metà della dimensione
  if (stride >= size / 2) return data[0];

  // Riduzione in-place: somma ogni elemento con quello più vicino a distanza 'stride'
  for (int i = 0; i < size; i += 2 * stride) {
      data[i] += data[i + stride];
  }

  // Chiamata ricorsiva con il passo raddoppiato
  return recursiveReduceNeighbored(data, size, 2 * stride);
}
```

Caratteristiche

- **$O(\log_2 N)$** step di ricorsione, con **$N/2^{step}$** operazioni per livello. **Complessità del lavoro/temporale** pari a **O(N)**.
- **In-place**: Modifica l'array originale senza allocare memoria aggiuntiva.
- **Parallelizzabile**: Le somme parziali possono essere eseguite in parallelo.

Implementazione Iterativa a Coppie per la Riduzione

Implementazione Ricorsiva dell'Approccio Interleaved Pair - Funzione C Standard

```c
int recursiveReduceInterleaved(int *data, int const size) {
    // Caso base: se l'array ha un solo elemento, lo restituisce
    if (size == 1) return data[0];

    // Calcola la nuova "ampiezza" del passo (metà della dimensione attuale)
    int const stride = size / 2;

    // Riduzione in-place: somma gli elementi a coppie
    for (int i = 0; i < stride; i++) {
        data[i] += data[i + stride];  // Somma l'elemento corrente con quello a distanza 'stride'
    }

    // Chiamata ricorsiva con la metà degli elementi
    return recursiveReduceInterleaved(data, stride);
}
```

Caratteristiche

- **$O(\log_2 N)$** step di ricorsione, con **$N/2^{step}$** operazioni per livello. **Complessità del lavoro/temporale** pari a **O(N)**.
- **In-place**: Modifica l'array originale senza allocare memoria aggiuntiva.
- **Parallelizzabile**: Le somme parziali possono essere eseguite in parallelo.

Il Problema della Riduzione Parallela

Definizione

- Il problema generale di applicare un'operazione *associativa* e (non necessariamente) *commutativa* su un array di elementi è noto come **problema di riduzione**. La sua esecuzione parallela è chiamata **riduzione parallela**.
- Con P thread fisicamente in parallelo (P processori) e N elementi, la **complessità temporale** è **$O(N/P + \log_2 N)$**
  - **N/P**: rappresenta la divisione iniziale del lavoro tra i P processori disponibili.
  - **$\log_2 N$**: rappresenta il tempo necessario per la riduzione ad albero.
- In un thread block in CUDA, quando N=P (un thread per elemento), la complessità diventa **$O(1 + \log_2 N) = O(\log_2 N)$**

Oltre l'Addizione

- Il codice di riduzione parallela, pur implementando l'addizione, può essere generalizzato a qualsiasi operazione associativa e commutativa:
  - Trovare il valore **massimo** nell'array.
  - Trovare il valore **minimo** nell'array.
  - Calcolare la **media** degli elementi dell'array.
  - Calcolare il **prodotto** degli elementi dell'array.

Importanza

- La riduzione parallela è uno dei **pattern di parallelismo** più comuni e un'operazione chiave in molti algoritmi paralleli.

Il Problema della Riduzione Parallela in CUDA

Sfida e Approcci

- **Problema**: Riduzione di array molto grandi
  - Necessità di utilizzare **più blocchi di thread**.
  - Ogni blocco elabora una **porzione dell'array** con approccio ad albero.
  - Impossibilità di sincronizzazione globale tra blocchi CUDA (sarebbe troppo costoso). Come si sommano i risultati ottenuti da ciascun blocco?

Soluzioni Possibili

- **Approccio Multi-Kernel**:
  - Lancio di kernel successivi come punto di sincronizzazione.
  - Ogni kernel elabora i risultati intermedi del precedente.
  - Riduzione progressiva fino al risultato finale.
- **Approccio Ibrido GPU-CPU (Soluzione Semplificata)**:
  - GPU: Riduzione parallela per blocco
  - CPU: Somma finale dei risultati parziali

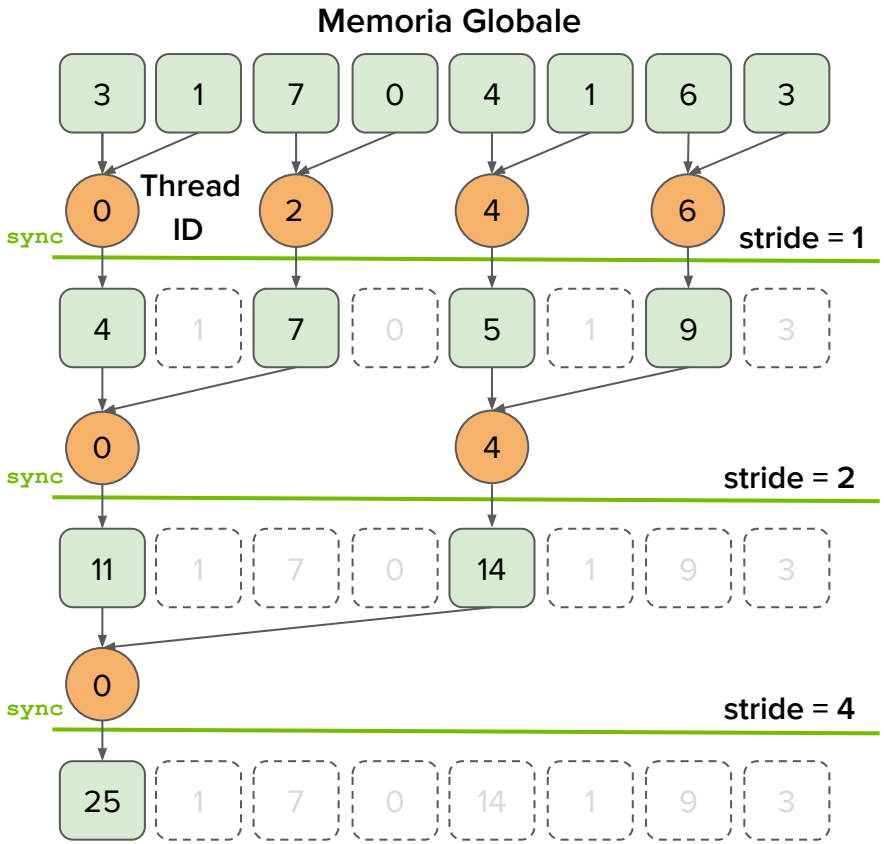
Obiettivo

- **Massime Prestazioni.** La riduzione ha una bassa intensità aritmetica, quindi è "**memory-bound**" (limitata dalla memoria). Il nostro obiettivo deve essere **massimizzare la bandwidth.**

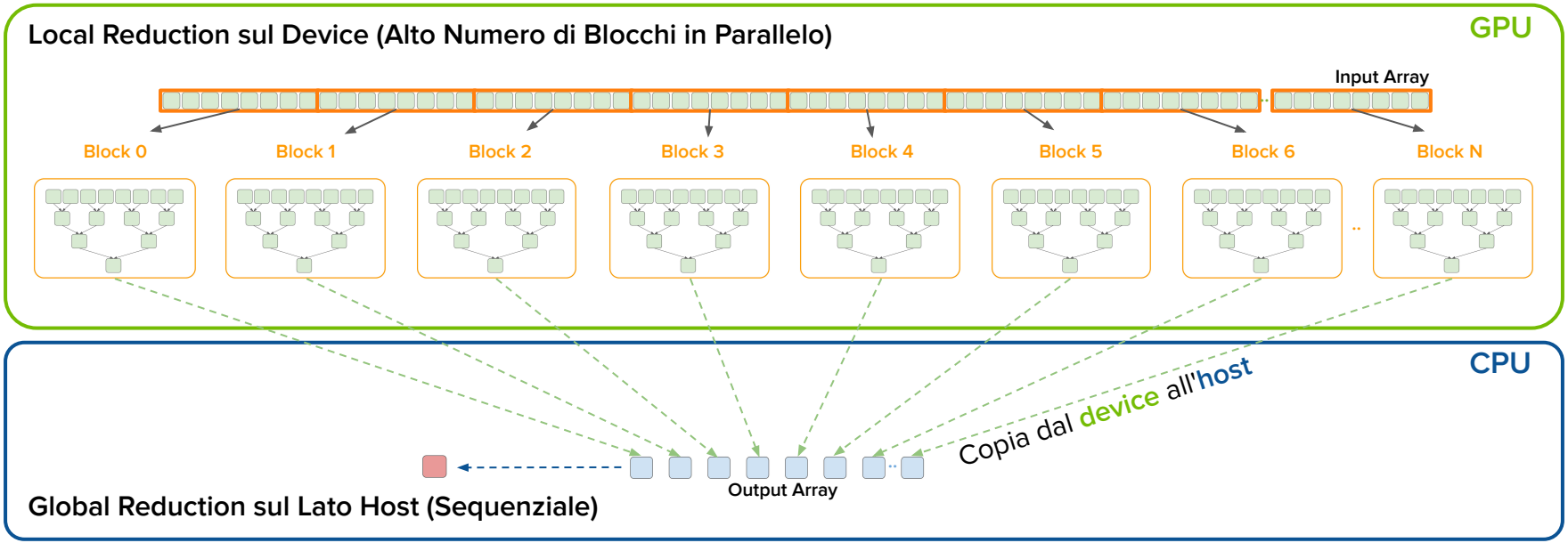
Riduzione Parallela: 1) Implementazione Neighbored Pair

Funzionamento

- Ogni thread somma due elementi adiacenti per produrre una **somma parziale**.
- La distanza tra gli elementi sommati (stride) **raddoppia** ad ogni iterazione.
- **Due array in memoria globale**: uno per l'array completo, uno per le somme parziali di ogni blocco.
- Ogni blocco opera **indipendentemente** su una porzione dell'array.
- **Riduzione in-place**: i valori in memoria globale vengono sostituiti dalle somme parziali ad ogni passo.
- `__syncthreads()`: Garantisce la sincronizzazione tra i thread di un blocco prima dell'iterazione successiva.
- Le somme parziali finali di ogni blocco vengono **accumulate** in un array separato e poi **sommate sequenzialmente** sull'host.



Riduzione Parallela: 1) Implementazione Neighbored Pair



Passi

- **Distanza iniziale** (stride): Impostata a 1.
- **Raddoppio dello stride**: Ad ogni ciclo di riduzione, la distanza stride viene moltiplicata per 2.
- **Assenza di sincronizzazione tra blocchi**: I blocchi di thread operano indipendentemente.
- **Somma finale**: Le somme parziali di ogni blocco vengono copiate sull'host e sommate sequenzialmente.

Riduzione Parallela: 1) Implementazione Neighbored Pair

Implementazione Neighbored Pair - Funzione CUDA C

```
// Grid e Block 1D
__global__ void reduceNeighbored(int *g_idata, int *g_odata, unsigned int n) {
 unsigned int tid = threadIdx.x; // ID del thread all'interno del blocco
 unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x; // Indice globale del thread

 int *idata = g_idata + blockIdx.x * blockDim.x; // Puntatore ai dati di input per questo blocco

 if (idx >= n) return; // Verifica se il thread è fuori dai limiti dei dati

 // Riduzione in-place nella memoria globale
 for (int stride = 1; stride < blockDim.x; stride *= 2) {
   // Riduzione in-place con passi raddoppiati
   if ((tid % (2 * stride)) == 0) {
     // Solo i thread con ID multiplo di 2*stride partecipano alla somma
     idata[tid] += idata[tid + stride]; // Somma il valore del vicino
   }
   __syncthreads(); // Sincronizzazione dei thread all'interno del blocco
 }

 if (tid == 0) g_odata[blockIdx.x] = idata[0]; // Il thread 0 scrive il risultato del blocco in g_odata
}
```

Riduzione Parallela: 1) Implementazione Neighbored Pair

Implementazione Neighbored Pair - Funzione CUDA C

```
// Grid e Block 1D
__global__ void reduceNeighbored(int *g_id
unsigned int tid = threadIdx.x; // ID del
unsigned int idx = blockIdx.x * blockDim.

int *idata = g_idata + blockIdx.x * blockD

if (idx >= n) return; // Verifica se il t

// Rid                    oria globa
for (int stride = 1; stride < blockDim.x;
  // Riduzione in-place con passi raddopp
  if ((tid % (2 * stride)) == 0) {
    // Solo i thread con ID multiplo di 2
    idata[tid] += idata[tid + stride]; //
  }
  __syncthreads(); // Sincronizzazione de
}

if (tid == 0) g_odata[blockIdx.x] = idata[0]; // Il thread 0 scrive il risultato del blocco in g_odata
}
```

Operatore % molto lento

Warp Divergence

- La condizione `if ((tid % (2 * stride)) == 0)` crea una situazione di **warp divergence** (Inefficiente!)
- La condizione `if` fa sì che solo alcuni thread all'interno di un warp eseguano l'addizione ad ogni iterazione.
  - **Prima Iterazione (stride = 1):** Solo i thread con ID pari sono attivi.
  - **Seconda Iterazione (stride = 2)**: Solo un thread ogni quattro è attivo (thread 0, 4, 8, etc.).
- **Soluzione**: È possibile ridurre la warp divergence riordinando gli indici dell'array in modo tale che i thread vicini tra loro processino elementi consecutivi durante l'operazione di addizione

Riduzione Parallela: Riduzione Globale (Host)

Global Reduction (Host)

```
int main(int argc, char **argv) {

// Configurazione del device
...

// Dimensione dell'array (potenza di 2)
int size = 1 << 26;

// Configurazione Griglia e Blocchi
int blocksize = 512;
dim3 block(blocksize, 1);
dim3 grid((size + block.x - 1) / block.x, 1);

// Allocazione ed Inizializzazione Memoria Host
size_t bytes = size * sizeof(int);
int *h_idata = (int *)malloc(bytes);
int *h_odata = (int *)malloc(grid.x * sizeof(int));

// Inizializzazione Random (max 10)
for (int i = 0; i < size; i++) h_idata[i] = (int)(rand() % 10);
```

Riduzione Parallela: Riduzione Globale (Host)

Implementazione Neighbored Pair - Global Reduction (Host)

```
// Allocazione Memoria Device
int *d_idata, *d_odata;
cudaMalloc((void **)&d_idata, bytes);
cudaMalloc((void **)&d_odata, grid.x * sizeof(int));

// Trasferimento Dati Host -> Device + Calcolo Parallelo
cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice);
reduceNeighbored<<<grid, block>>>(d_idata, d_odata, size);
cudaDeviceSynchronize(); // Sincronizzazione prima della Riduzione Globale

// Trasferimento Risultati Device -> Host + Somma Finale
gpu_sum = 0;
cudaMemcpy(h_odata, d_odata, grid.x * sizeof(int), cudaMemcpyDeviceToHost);
for (int i = 0; i < grid.x; i++) gpu_sum += h_odata[i];
printf("GPU Reduction Sum: %d\n", gpu_sum);
```

Riduzione Parallela - Confronto fra Kernel

NVIDIA Nsight Compute

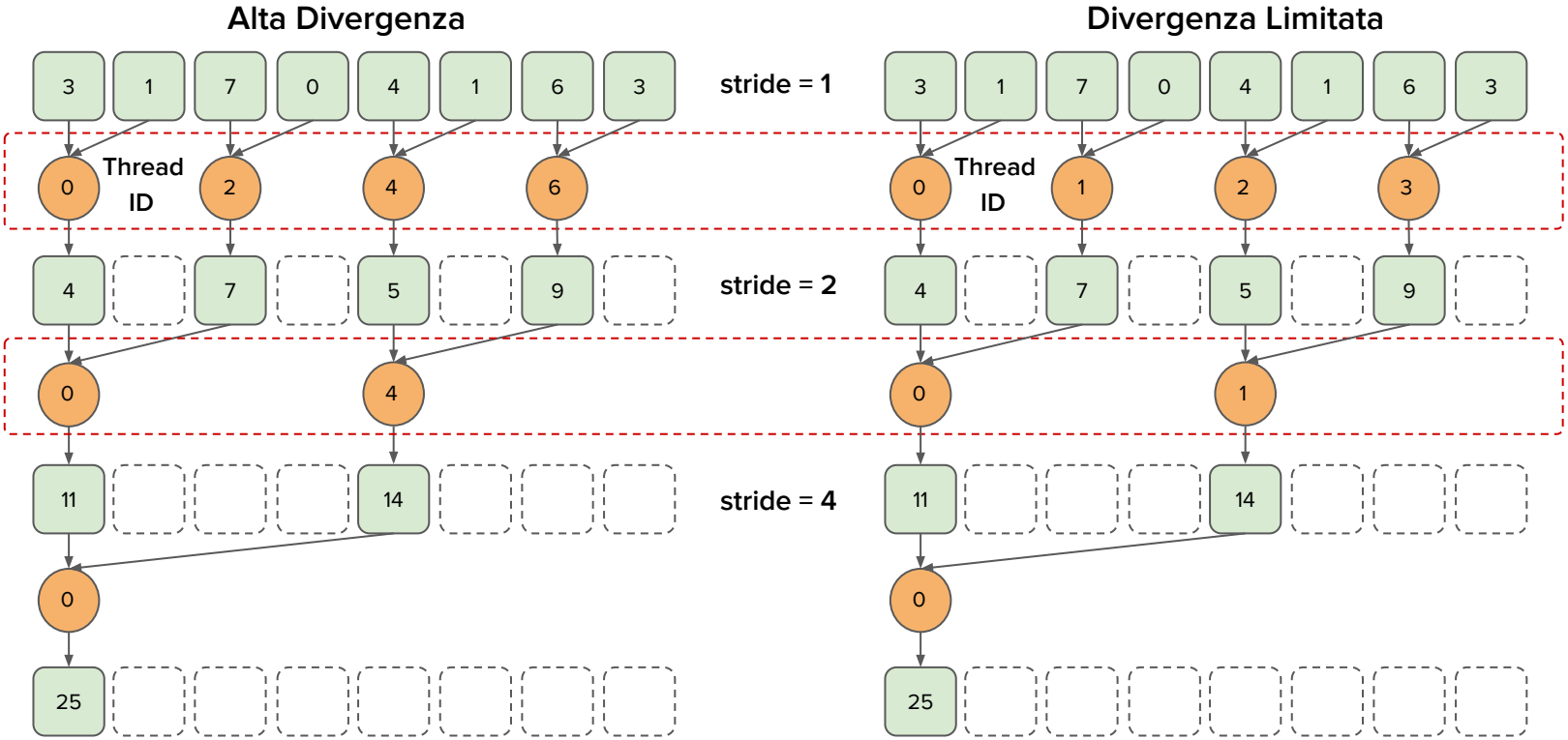
Dim. Array (2^{26}), Dim. Blocco (512)

GPU RTX 3090
CPU i9-10920X

Kernel	Dim. Griglia	Istruzioni Eseguite (M)	Load Memory Through. (GB/s)	Branch Efficiency (%)	Runtime (ms)	Cumulative Speedup
Recursive Interleaved (CPU)	–	–	–	–	130.888	–
Neighbored (Divergence)	(131072)	612,10	123,56	68,75	2,332*	56.13

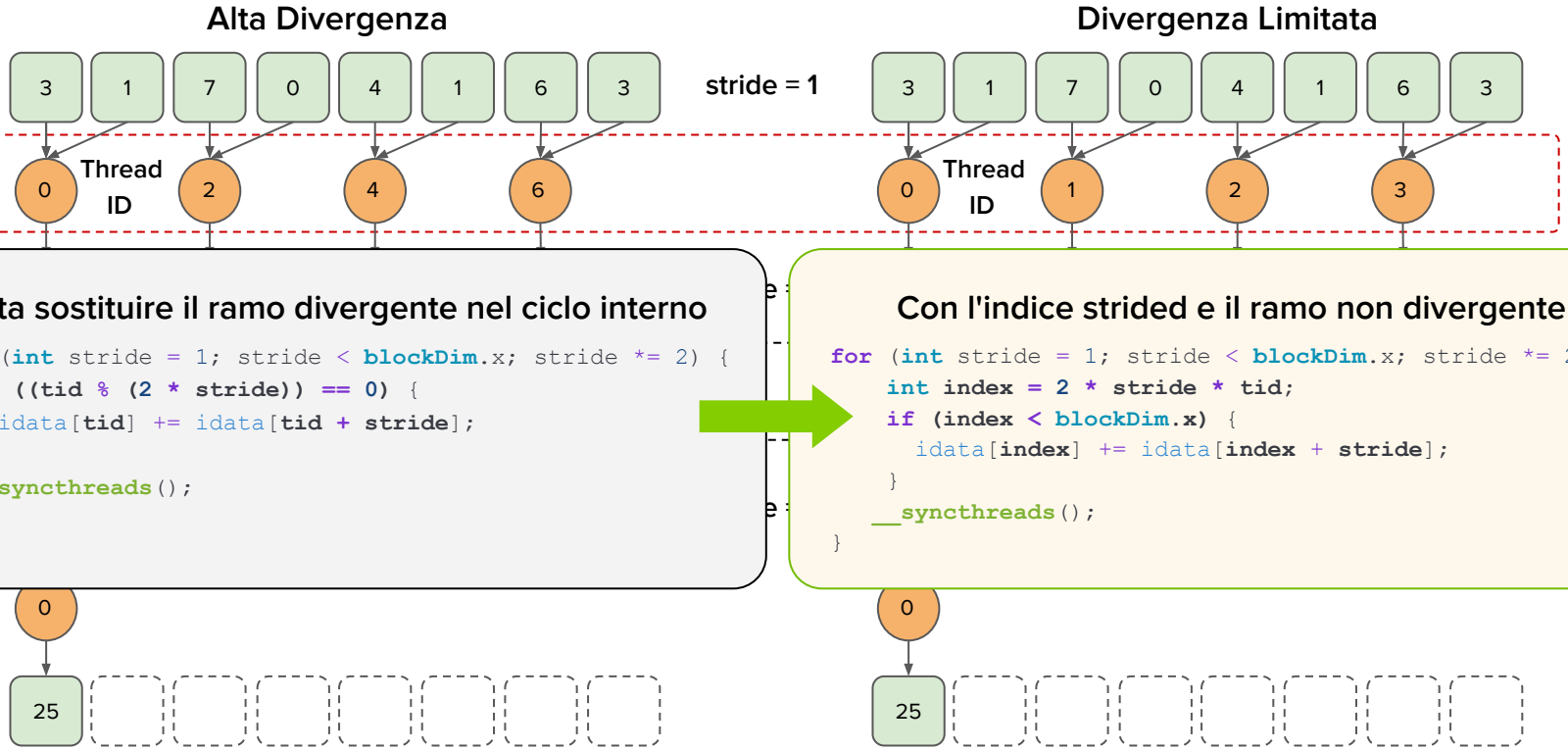
* Tempistiche rilevate con Timer CPU: (Calcolo Somme Parziali [Device] + Calcolo Somma Finale [Host])

Riduzione Parallela: Migliorare la Divergenza



- Confrontando con la figura precedente (**sinistra**), la posizione di memorizzazione delle somme parziali non è cambiata, diversamente dalla suddivisione del lavoro dei thread (**destra**).

Riduzione Parallela: Migliorare la Divergenza



Basta sostituire il ramo divergente nel ciclo interno

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

Con l'indice strided e il ramo non divergente

```
for (int stride = 1; stride < blockDim.x; stride *= 2) {
    int index = 2 * stride * tid;
    if (index < blockDim.x) {
      idata[index] += idata[index + stride];
    }
    __syncthreads();
}
```

- Confrontando con la figura precedente (**sinistra**), la posizione di memorizzazione delle somme parziali non è cambiata, diversamente dalla suddivisione del lavoro dei thread (**destra**).

Riduzione Parallela: Migliorare la Divergenza

Implementazione Neighbored Pair - Funzione CUDA C

```
__global__ void reduceNeighboredLess(int *g_idata, int *g_odata, unsigned int n) {
 unsigned int tid = threadIdx.x; // ID del thread all'interno del blocco
 unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x; // Indice globale del thread

 int *idata = g_idata + blockIdx.x * blockDim.x; // Puntatore ai dati di input per questo blocco

 if(idx >= n) return; // Verifica se il thread è fuori dai limiti dei dati

 // Riduzione in-place nella memoria globale
 for (int stride = 1; stride < blockDim.x; stride *= 2) { // Raddoppia lo stride ad ogni iterazione
   // Converte tid in un indice locale dell'array
   int index = 2 * stride * tid; // Calcola l'indice dell'elemento da sommare

   if (index < blockDim.x) { // Verifica se l'indice è all'interno del blocco
     idata[index] += idata[index + stride]; // Somma gli elementi a distanza stride
   }

   __syncthreads(); // Assicura che tutti i thread abbiano completato la somma prima di proseguire
 }

 if (tid == 0) g_odata[blockIdx.x] = idata[0]; // Il thread 0 scrive il risultato del blocco in g_odata
}
```

Riduzione Parallela: Migliorare la Divergenza

Implementazione Neighbored Pair - Funzione CUDA C

```
__global__ void reduceNeighboredLess(int *g_idata, int *g_odata, unsigned int n) {
 unsigned int tid = threadIdx.x; // ID del thread all'interno del blocco
 unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x; // Indice globale del thread

 int *idata = g_idata + blockIdx.x * block

 if(idx >= n) return; // Verifica se il th

 // Riduzione in-place nella memoria globa
 for (int stride = 1; stride < blockDim.x;
   // Converte tid in un indice locale de
   int index = 2 * stride * tid; // Calco

   if (index < blockDim.x) { // Verifica
     idata[index] += idata[index + stride]
   }

   __syncthreads(); // Assicura che tutti
 }

 if (tid == 0) g_odata[blockIdx.x] = idata[0]; // Il thread 0 scrive il risultato del blocco in g_odata
}
```

Branch Non Divergente

- Si ottiene un indice che assegna a **thread adiacenti** elementi che sono a distanza **`stride`** l'uno dall'altro.
- Nel codice precedente, l'accesso agli elementi era sequenziale e causava **divergenza di warp**, perché solo alcuni thread eseguivano l'addizione in ogni iterazione.
- In questo modo, i thread di un warp eseguono la stessa istruzione (l'addizione) contemporaneamente, **riducendo** la divergenza.

Riduzione Parallela: Migliorare la Divergenza

Implementazione Neighbored Pair - Funzione CUDA C

```
__globa
 unsig
 unsig

 int *

 if(id

 // Ri
 for (
   //
   int

   if
     i
   }

   __sy
 }

 if (tid == 0) g_odata[blockIdx.x] = idata[0]; // Il thread 0 scrive il risultato del blocco in g_odata
}
```

Riduzione con Blocchi di 512 Thread

- **Ipotesi**: Dimensione del blocco pari a 512 thread (16 warp)
- **Fasi di Riduzione**
  - **Prime 4 iterazioni (no divergenza):**
    - *Iter 1*: 8 warp attivi, 8 inattivi
    - *Iter 2*: 4 warp attivi, 12 inattivi
    - *Iter 3*: 2 warp attivi, 14 inattivi
    - *Iter 4*: 1 warp attivo, 15 inattivi
    - Non c'è divergenza.
  - **Ultime 5 iterazioni (con divergenza):**
    - Thread attivi < 32 (dimensione warp)
    - Thread attivi e inattivi mescolati nello stesso warp
    - C'è divergenza

Riduzione Parallela - Confronto fra Kernel

NVIDIA Nsight Compute

Dim. Array (2^{26}), Dim. Blocco (512)

GPU RTX 3090
CPU i9-10920X

Kernel	Dim. Griglia	Istruzioni Eseguite (M)	Load Memory Through. (GB/s)	Branch Efficiency (%)	Runtime (ms)	Cumulative Speedup
Recursive Interleaved (CPU)	–	–	–	–	130.888	–
Neighbored (Divergence)	(131072)	612,10	123,56	68,75	2,332*	56.13
Neighbored (No Divergence)	**(131072)**	**241,96**	**230,84**	**98,36**	**1,325***	**98.81**

* Tempistiche rilevate con Timer CPU: (Calcolo Somme Parziali [Device] + Calcolo Somma Finale [Host])

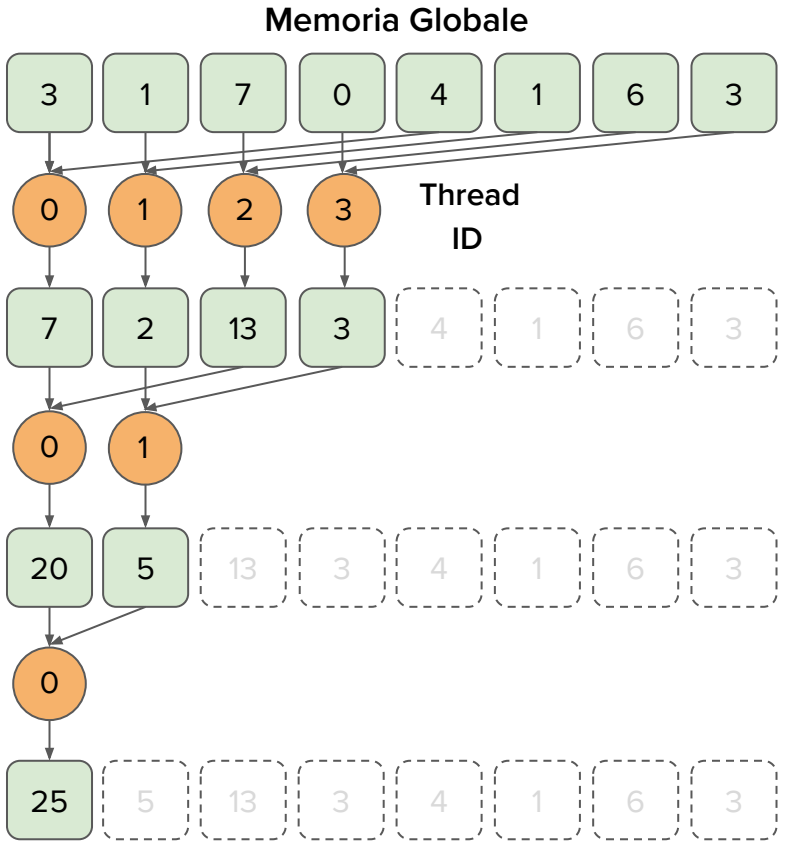
Riduzione Parallela: 2) Implementazione Interleaved Pair

Funzionamento

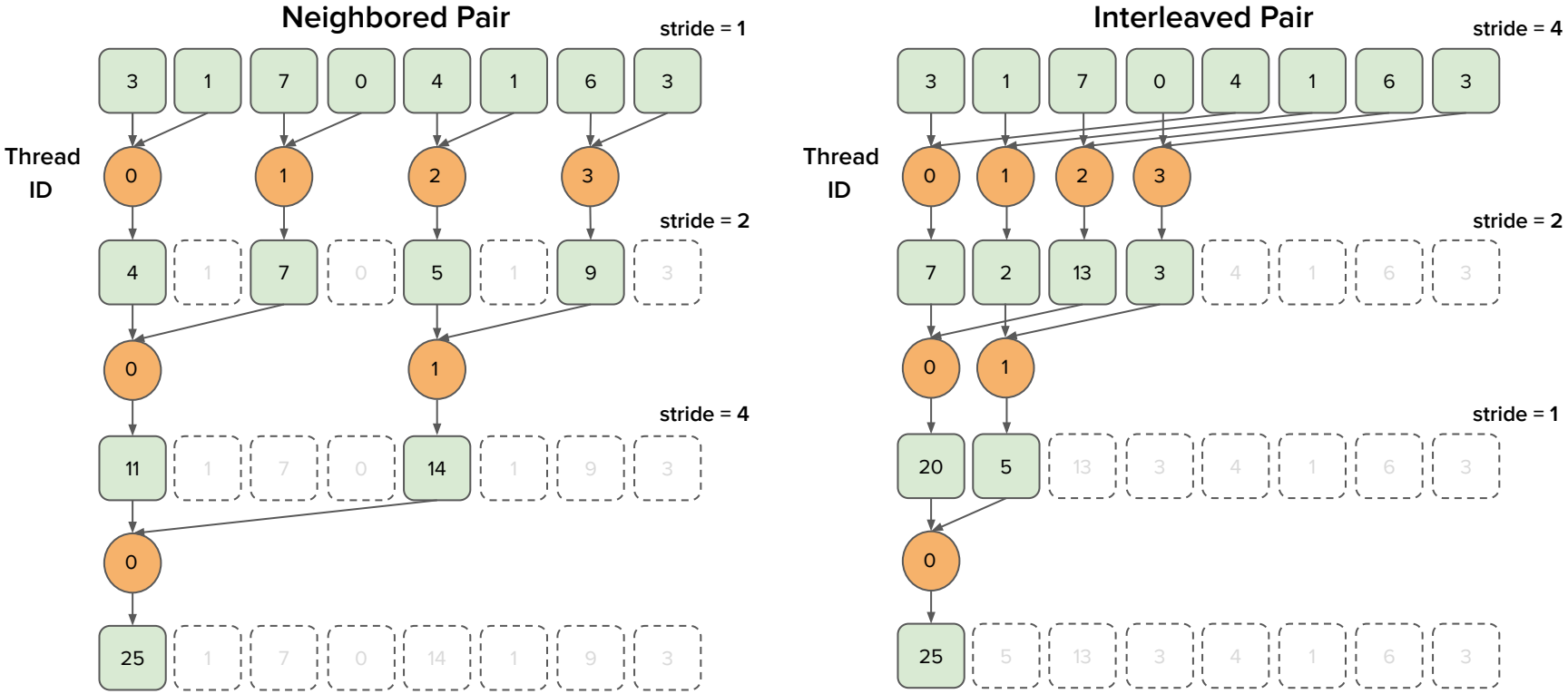
- Lo stride inizia dalla **metà** della dimensione del blocco e viene **dimezzato** ad ogni iterazione.
- Ogni thread somma due elementi separati dallo stride corrente per produrre la somma parziale.
- La riduzione avviene **in-place** nella memoria globale.
- Il numero di thread attivi si riduce **progressivamente** ad ogni iterazione.

Differenze rispetto a Neighbored Pair

- I thread attivi nella riduzione interleaved non vengono cambiati.
- **Direzione dello stride**: Parte grande e si riduce, invece di partire piccolo e aumentare.
- Le **locazioni di lettura e scrittura in memoria** globale per ciascun thread sono differenti.

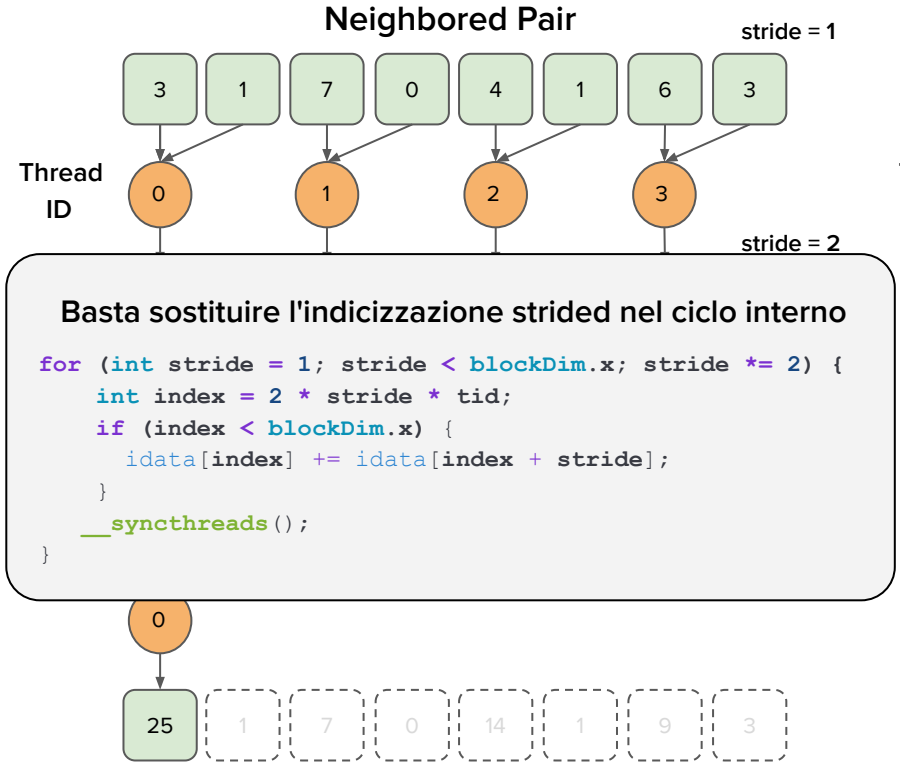


Riduzione Parallela: 2) Implementazione Interleaved Pair



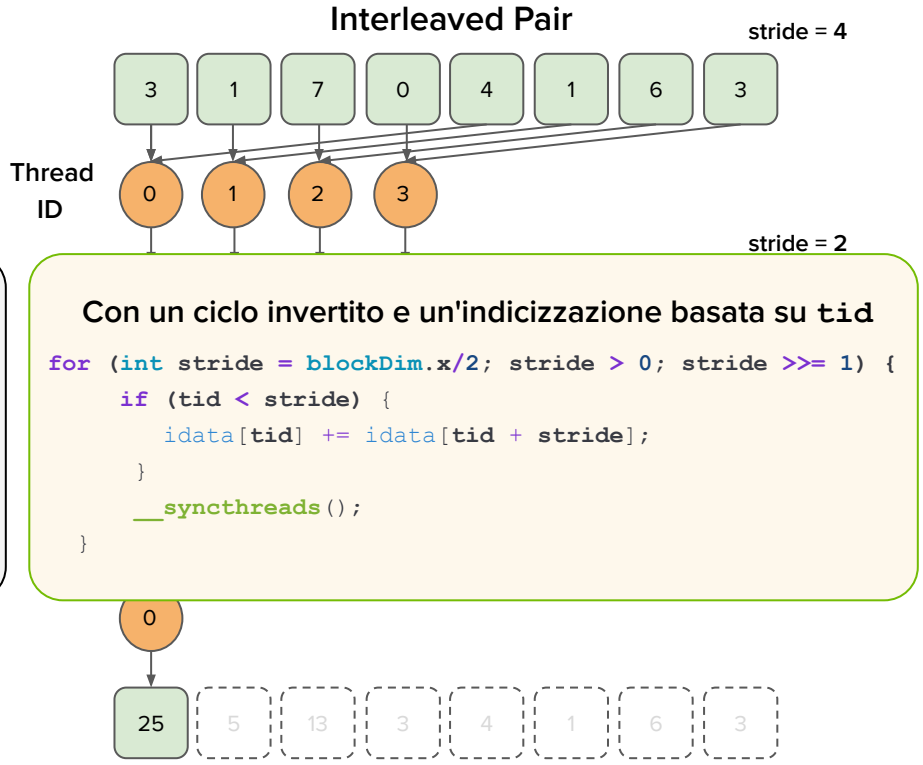
- Nel Neighbored Pair (**sinistra**) lo stride tra thread attivi raddoppia ad ogni passo, mentre nell'Interleaved Pair (**destra**) lo stride iniziale viene dimezzato ad ogni iterazione.

Riduzione Parallela: 2) Implementazione Interleaved Pair



Basta sostituire l'indicizzazione strided nel ciclo interno

```
for (int stride = 1; stride < blockDim.x; stride *= 2) {
    int index = 2 * stride * tid;
    if (index < blockDim.x) {
      idata[index] += idata[index + stride];
    }
   __syncthreads();
}
```



Con un ciclo invertito e un'indicizzazione basata su `tid`

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

- Nel Neighbored Pair (**sinistra**) lo stride tra thread attivi raddoppia ad ogni passo, mentre nell'Interleaved Pair (**destra**) lo stride iniziale viene dimezzato ad ogni iterazione.

Riduzione Parallela: 2) Implementazione Interleaved Pair

Implementazione Interleaved Pair - Funzione CUDA C

```
__global__ void reduceInterleaved(int *g_idata, int *g_odata, unsigned int n) {
    unsigned int tid = threadIdx.x; // ID del thread all'interno del blocco
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x; // Indice globale del thread

    int *idata = g_idata + blockIdx.x * blockDim.x; // Puntatore ai dati di input per questo blocco

    if(idx >= n) return; // Verifica se il thread è fuori dai limiti dei dati
    // Riduzione in-place nella memoria globale
    // Inizia con uno stride pari alla metà della dimensione del blocco
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        // Ogni thread attivo somma due elementi separati dallo stride corrente
        if (tid < stride) {
            // Il thread somma l'elemento alla sua posizione con quello a distanza 'stride'
            idata[tid] += idata[tid + stride];
        }
        __syncthreads();
    }

    if (tid == 0) g_odata[blockIdx.x] = idata[0]; // Il thread 0 scrive il risultato del blocco in g_odata
}
```

Riduzione Parallela: 2) Implementazione Interleaved Pair

Implementazione Interleaved Pair - Funzione CUDA C

```
__global__ void reduceInterleaved(int *g_idata, int *g_odata, unsigned int n) {
    unsigned int tid = threadIdx.x; // Ind
    unsigned int idx = blockIdx.x * bloc

    int *idata = g_idata + blockIdx.x * b

    if(idx >= n) return; // Verifica se i
    // Riduzione in-place nella memoria globale
    // Inizia con uno stride pari alla metà della dimensione del blocco
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        // Ogni thread attivo somma due elementi separati dallo stride corrente
        if (tid < stride) {
            // Il thread somma l'elemento alla sua posizione con quello a distanza 'stride'
            idata[tid] += idata[tid + stride];
        }
        __syncthreads();
    }

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

- Lo stride tra due elementi è inizializzato a **metà della dimensione del blocco di thread** e poi viene ridotto della metà in ogni round.

- Forza la prima metà del blocco di thread a eseguire l'addizione alla prima iterazione, il primo quarto di un blocco di thread alla seconda iterazione e così via.
- Metà dei thread sono **inattivi** alla prima iterazione del ciclo! (**Spreco**)

Riduzione Parallela - Confronto fra Kernel

NVIDIA Nsight Compute

Dim. Array (2^{26}), Dim. Blocco (512)

GPU RTX 3090
CPU i9-10920X

Kernel	Dim. Griglia	Istruzioni Eseguite (M)	Load Memory Through. (GB/s)	Branch Efficiency (%)	Runtime (ms)	Cumulative Speedup
Recursive Interleaved (CPU)	–	–	–	–	130.888	–
Neighbored (Divergence)	(131072)	612,10	123,56	68,75	2,332*	56.13
Neighbored (No Divergence)	(131072)	241,96	230,84	98,36	1,325*	98.81
Interleaved	**(131072)**	**205,78**	**266,01**	**98,36**	**1,175***	**111.38**

* Tempistiche rilevate con Timer CPU: (Calcolo Somme Parziali [Device] + Calcolo Somma Finale [Host])

Loop Unrolling

Cos'è?

- **Loop Unrolling** è una tecnica di ottimizzazione che **riduce le istruzioni di controllo nei cicli**, scrivendo il corpo del ciclo più volte nel codice.

Qual è il problema?

- **Sovraccarico nei Cicli Tradizionali:** Controlli ripetuti (salti e aggiornamenti dell'indice) riducono le prestazioni, specialmente con cicli lunghi.

Perché viene introdotto?

- **Migliorare le Prestazioni**: Diminuisce il sovraccarico delle istruzioni di salto.
- **Sfruttare meglio CPU/GPU**: Aumenta l'efficienza delle operazioni parallele perché le istruzioni nel ciclo srotolato (unrolled) sono più indipendenti, consentendo alla CPU/GPU (e thread CUDA) di eseguirle simultaneamente.

Come funziona?

- **Fattore di Unrolling**: Numero di repliche del corpo del ciclo, riducendo le iterazioni.
- **Efficace per Cicli Sequenziali**: Ideale per il processamento di array quando il numero di iterazioni è noto.

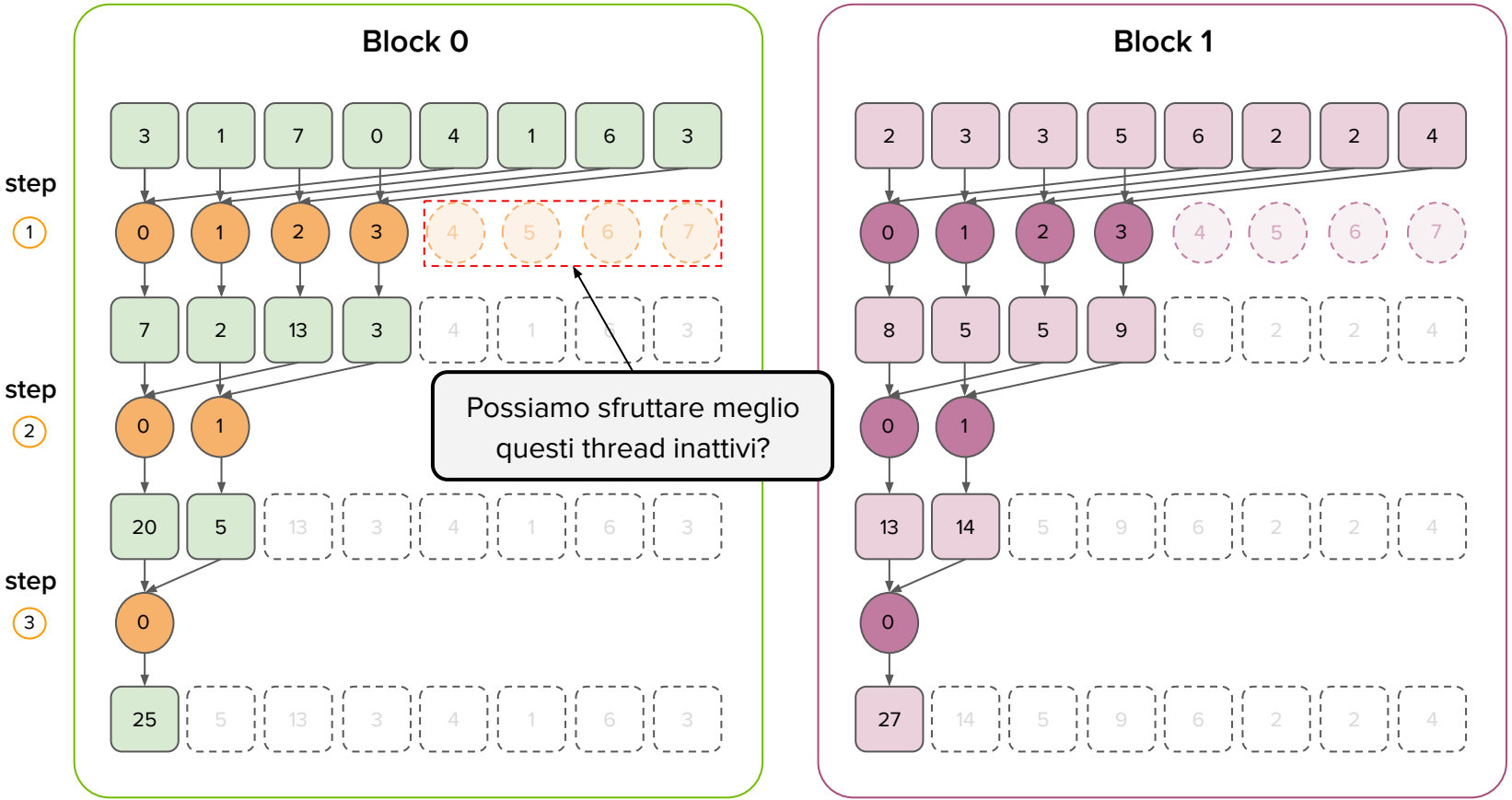
Ciclo Tradizionale

```
// 100 verifiche della condizione
for (int i = 0; i < 100; i ++) {
   a[i] = b[i] + c[i];}
```

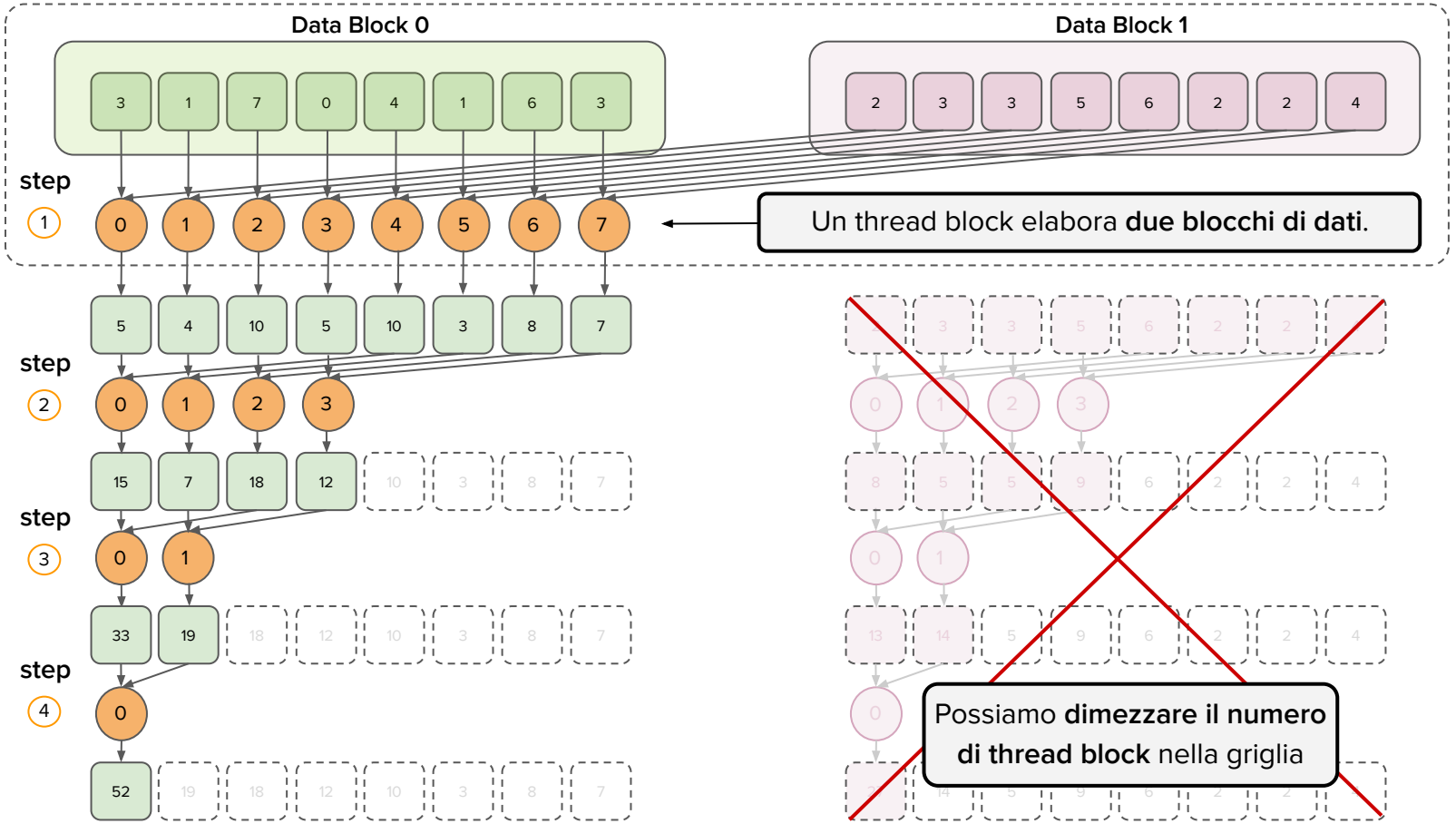
Ciclo Unrolled (fattore 2)

```
// Solo 50 verifiche
for (int i = 0; i < 100; i += 2) {
   a[i] = b[i] + c[i];
   a[i + 1] = b[i + 1] + c[i + 1];}
```

Riduzione Parallela: 2) Implementazione Interleaved Pair



Riduzione Parallela: 3) Interleaved Pair con Unrolling



Riduzione Parallela: 3) Interleaved Pair con Unrolling

Implementazione Interleaved Pair - Funzione CUDA C

```
__global__ void reduceInterleaved(int *g_idata, int *g_odata, unsigned int n) {
```

Dimezzare il numero di blocchi (lato host) e recuperare una iterazione del loop di riduzione

```
unsigned int tid = threadIdx.x;
unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

int *idata = g_idata + blockIdx.x * blockDim.x;
if(idx >= n) return;
```

Ogni thread aggiunge un elemento dal blocco dati vicino (tutti i thread attivi)

```
unsigned int tid = threadIdx.x;
unsigned int idx = blockIdx.x * blockDim.x * 2 + threadIdx.x;

int *idata = g_idata + blockIdx.x * blockDim.x * 2;

if (idx + blockDim.x < n) g_idata[idx] += g_idata[idx + blockDim.x];

__syncthreads();
```

Riduzione Parallela: 3) Interleaved Pair con Unrolling

Implementazione Riduzione con Unrolling - Fattore 2

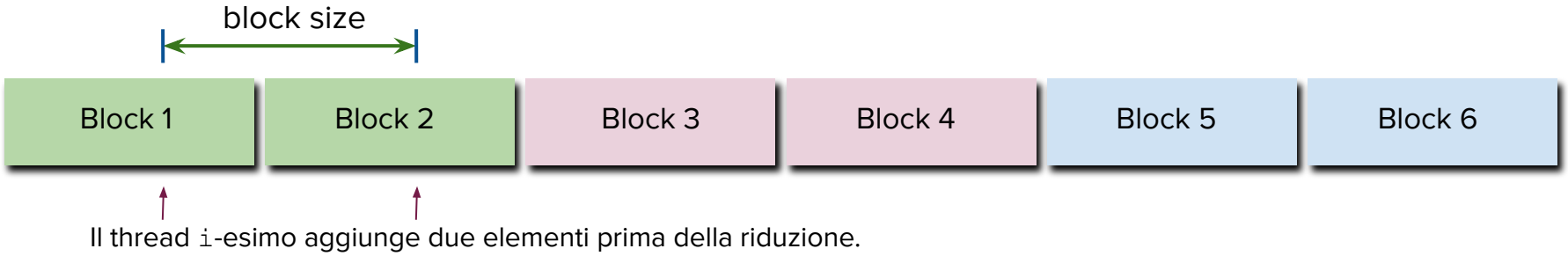
```
__global__ void reduceUnrolling2 (int *g_idata, int *g_odata, unsigned int n) {
 // Calcola l'ID del thread e l'indice globale dei dati
 unsigned int tid = threadIdx.x;
 unsigned int idx = blockIdx.x * blockDim.x * 2 + threadIdx.x;

 // Ottiene il puntatore ai dati di input per questo blocco (srotolamento di 2)
 int *idata = g_idata + blockIdx.x * blockDim.x * 2;

 // Srotolamento del ciclo di riduzione: ogni thread somma due elementi di due blocchi differenti
 if (idx + blockDim.x < n) g_idata[idx] += g_idata[idx + blockDim.x];

 __syncthreads(); // Sincronizza i thread del blocco

 // Riduzione in-place nella memoria globale
 for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
   if (tid < stride) {
     // Somma gli elementi a distanza 'stride'
     idata[tid] += idata[tid + stride];
   }
   __syncthreads(); // Sincronizza i thread del blocco
 }

 // Il thread 0 scrive il risultato del blocco in g_odata
 if (tid == 0)g_odata[blockIdx.x] = idata[0];
}
```

Riduzione Parallela: 3) Interleaved Pair con Unrolling

- L'indice globale dell'array viene adattato come mostrato, dato che **servono metà dei blocchi di thread** per elaborare gli stessi dati.

```
unsigned int idx = blockIdx.x * blockDim.x * 2 + threadIdx.x;
int *idata = g_idata + blockIdx.x * blockDim.x * 2;
```

- Ogni blocco elabora una porzione di dati **doppia** rispetto al caso baseline (metà dei blocchi necessari).
- **Iterazione "Implicita"**: Si anticipa una somma che avverrebbe comunque nelle fasi successive.
- Si **riduce il parallelismo** disponibile (warp e blocchi) a parità di dimensione dei dati.
- **Effetto**: Accesso alla memoria più efficiente con una leggera riduzione del parallelismo.



Modifiche per Richiamare il Kernel con Unrolling

Chiamata al Kernel per Unrolling di 2

```
// Riduce la dimensione della grid di un fattore di 2
reduceUnrolling2<<<grid.x / 2, block>>>(d_idata, d_odata, size);
```

- Ogni thread block gestisce 2 blocchi di dati, quindi si riduce la grid di un fattore di 2.

Estensione a Unrolling di 4 e 8

- Modifica la logica nel kernel per sommare 4 o 8 elementi alla volta.

```
// Riduce la dimensione della grid di un fattore di 4
reduceUnrolling4<<<grid.x / 4, block>>>(d_idata, d_odata, size);
```

```
// Riduce la dimensione della grid di un fattore di 8
reduceUnrolling8<<<grid.x / 8, block>>>(d_idata, d_odata, size);
```

- Il codice del kernel **va modificato** per gestire l'unrolling di 4 o 8 elementi. **Come?**
- La dimensione della grid **si riduce in base al fattore di unrolling** per mantenere costante il numero di elementi gestiti.

Riduzione Parallela - Confronto fra Kernel

NVIDIA Nsight Compute

Dim. Array (2^{26}), Dim. Blocco (512)

GPU RTX 3090
CPU i9-10920X

Kernel	Dim. Griglia	Istruzioni Eseguite (M)	Load Memory Through. (GB/s)	Branch Efficiency (%)	Runtime (ms)	Cumulative Speedup
Recursive Interleaved (CPU)	–	–	–	–	130.888	–
Neighbored (Divergence)	(131072)	612,10	123,56	68,75	2,332*	56.13
Neighbored (No Divergence)	(131072)	241,96	230,84	98,36	1,325*	98.81
Interleaved	(131072)	205,78	266,01	98,36	1,175*	111.38
Unroll 2 Blocks	**(65536)**	**111,28**	**471,18**	**98,36**	**0,683***	**191.55**
Unroll 4 Blocks	**(32768)**	**58,79**	**694,13**	**98,36**	**0,484***	**270.30**
Unroll 8 Blocks	**(16384)**	**34,37**	**768,26**	**98,44**	**0,423***	**309.29**

Più operazioni di memoria indipendenti per thread migliorano le prestazioni nascondendo meglio la latenza.

* Tempistiche rilevate con Timer CPU: (Calcolo Somme Parziali [Device] + Calcolo Somma Finale [Host])

Riduzione Parallela: 3) Interleaved Pair con Unrolling

Implementazione Riduzione con Unrolling - Fattore 4

```
__global__ void reduceUnrolling4(int *g_idata, int *g_odata, unsigned int n) {
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x * 4 + threadIdx.x;

    int *idata = g_idata + blockIdx.x * blockDim.x * 4;

    // Unrolling del ciclo di riduzione: ogni thread somma quattro elementi alla volta
    if (idx + 3 * blockDim.x < n) {
        int a1 = g_idata[idx];
        int a2 = g_idata[idx + blockDim.x];
        int a3 = g_idata[idx + 2 * blockDim.x];
        int a4 = g_idata[idx + 3 * blockDim.x];
        g_idata[idx] = a1 + a2 + a3 + a4;
    }

    __syncthreads(); // Sincronizza i thread del blocco

    // Riduzione in-place nella memoria globale
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (tid < stride) {
            idata[tid] += idata[tid + stride];
        }
        __syncthreads(); // Sincronizza i thread del blocco
    }

    // Il thread 0 scrive il risultato del blocco in g_odata
    if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```

Algorithm Cascading (Ibrido)

- Somma sequenziale di elementi multipli per thread.
- Riduzione parallela ad albero.
- Porta a significativi incrementi di velocità nella pratica

Riduzione Parallela: 3) Interleaved Pair con Unrolling

Implementazione Riduzione con Unrolling - Fattore 8

```
__global__ void reduceUnrolling8(int *g_idata, int *g_odata, unsigned int n) {
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x * 8 + threadIdx.x;
    int *idata = g_idata + blockIdx.x * blockDim.x * 8;
    // Unrolling del ciclo di riduzione: ogni thread somma otto elementi alla volta
    if (idx + 7 * blockDim.x < n) {
        int a1 = g_idata[idx];
        int a2 = g_idata[idx + blockDim.x];
        int a3 = g_idata[idx + 2 * blockDim.x];
        int a4 = g_idata[idx + 3 * blockDim.x];
        int b1 = g_idata[idx + 4 * blockDim.x];
        int b2 = g_idata[idx + 5 * blockDim.x];
        int b3 = g_idata[idx + 6 * blockDim.x];
        int b4 = g_idata[idx + 7 * blockDim.x];
        g_idata[idx] = a1 + a2 + a3 + a4 + b1 + b2 + b3 + b4;
    }
    __syncthreads(); // Sincronizza i thread del blocco
    // Riduzione in-place nella memoria globale
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (tid < stride) {
            idata[tid] += idata[tid + stride];
        }
        __syncthreads(); // Sincronizza i thread del blocco
    }
    // Il thread 0 scrive il risultato del blocco in g_odata
    if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```

Algorithm Cascading (Ibrido)

- Somma sequenziale di elementi multipli per thread.
- Riduzione parallela ad albero.
- Porta a significativi incrementi di velocità nella pratica

Ottimizzazione Avanzata: 4) Unrolling a Livello di Warp

Obiettivo

- **Eliminare l'overhead di sincronizzazione e controllo del loop** nelle fasi finali della riduzione parallela, sfruttando il parallelismo intrinseco dei warp.

Contesto

- Man mano che la riduzione procede, diminuiscono i thread attivi. Quando lo stride ≤ 32, rimane attivo un solo warp.
- L'unrolling tradizionale riduce gli accessi alla memoria, ma `__syncthreads()` introduce ancora un overhead, specialmente quando il numero di thread attivi si riduce a un singolo warp (32 thread).

Soluzione

- Unrolling del warp, basato su due principi chiave:
  - **Sincronizzazione Implicita:** L'esecuzione di un warp è di tipo SIMD, quindi le istruzioni all'interno di un warp sono implicitamente sincronizzate. Non è necessaria la sincronizzazione esplicita `__syncthreads()`.
  - **Memoria Volatile** (`volatile`): Forzando l'accesso diretto alla memoria globale ad ogni operazione, si evita che le ottimizzazioni del compilatore (come l'utilizzo della cache) portino a risultati incoerenti.

Considerazioni

- L'efficacia dell'unrolling del warp **dipende dall'architettura GPU** e **dal problema specifico**. Misurare le prestazioni tramite benchmark per valutare l'effettivo guadagno.

Ottimizzazione Avanzata: 4) Unrolling a Livello di Warp

Implementazione Riduzione con Unrolling- Fattore 8

```
// Unrolling 8 (versione precedente)
for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
 if (tid < stride) idata[tid] += idata[tid + stride];
 __syncthreads();
}
```

Implementazione Riduzione con Unrolling a Livello di Warp - Fattore 8

```
// Unrolling Warp (versione ottimizzata)
for (int stride = blockDim.x / 2; stride > 32; stride >>= 1) { // Modifica 1
 if (tid < stride) idata[tid] += idata[tid + stride];
 __syncthreads(); // Sincronizzazione necessaria fino a quando stride > 32
}
if (tid < 32) { // Modifica 2 (unroll delle ultime 6 iterazioni del ciclo interno - No Loop)
 volatile int *vmem = idata;  // Forza l'accesso diretto alla memoria evitando la cache
 vmem[tid] += vmem[tid + 32]; // Sommatoria a livello di warp senza __syncthreads()
 vmem[tid] += vmem[tid + 16];
 vmem[tid] += vmem[tid + 8];
 vmem[tid] += vmem[tid + 4];
 vmem[tid] += vmem[tid + 2];
 vmem[tid] += vmem[tid + 1];
}
```

Nota: Questo evita lavoro inutile in tutti i warp, non solo nell'ultimo. Senza unrolling, tutti i warp eseguono ogni iterazione del ciclo `for` e dell'`if`

Riduzione Parallela - Confronto fra Kernel

NVIDIA Nsight Compute

Dim. Array (2^{26}), Dim. Blocco (512)

GPU RTX 3090
CPU i9-10920X

Kernel	Dim. Griglia	Istruzioni Eseguite (M)	Load Memory Through. (GB/s)	Branch Efficiency (%)	Runtime (ms)	Cumulative Speedup
Recursive Interleaved (CPU)	–	–	–	–	130.888	–
Neighbored (Divergence)	(131072)	612,10	123,56	68,75	2,332*	56.13
Neighbored (No Divergence)	(131072)	241,96	230,84	98,36	1,325*	98.81
Interleaved	(131072)	205,78	266,01	98,36	1,175*	111.38
Unroll 2 Blocks	(65536)	111,28	471,18	98,36	0,683*	191.55
Unroll 4 Blocks	(32768)	58,79	694,13	98,36	0,484*	270.30
Unroll 8 Blocks	(16384)	34,37	768,26	98,44	0,423*	309.29
Unroll 8 Blocks + Last Warp	**(16384)**	**22,30**	**767,86**	**100**	**0,414***	**316.42**

Stall Barrier (media cicli per warp)
9,29 vs 5,47

* Tempistiche rilevate con Timer CPU: (Calcolo Somme Parziali [Device] + Calcolo Somma Finale [Host])

Ottimizzazione Avanzata: 5) Unrolling Completo

Implementazione Riduzione con Unrolling a Livello di Warp - Fattore 8

```
// Unrolling Warp (versione ottimizzata)
for (int stride = blockDim.x / 2; stride > 32; stride >>= 1) {
  if (tid < stride) idata[tid] += idata[tid + stride];
  __syncthreads();
}
```

Unroll

- Ancora un **loop**. Possiamo provare a fare anche qui unroll.
- **`blockDim`** potrebbe non essere noto a compile time.
- Tuttavia, sappiamo che un limite (attuale) al numero dei thread di un blocco delle attuali GPU NVIDIA è pari a **1024**.

```
// Unrolling completo della riduzione
if (blockDim.x >= 1024 && tid < 512) idata[tid] += idata[tid + 512];
__syncthreads();
if (blockDim.x >= 512 && tid < 256) idata[tid] += idata[tid + 256];
__syncthreads();
if (blockDim.x >= 256 && tid < 128) idata[tid] += idata[tid + 128];
__syncthreads();
if (blockDim.x >= 128 && tid < 64) idata[tid] += idata[tid + 64];
__syncthreads();
```

Vale assumendo **potenze di 2**!

Ottimizzazione Avanzata: 5) Unrolling Completo

Implementazione Riduzione con Unrolling Completo - Fattore 8

```
__global__ void reduceCompleteUnrollWarps8(int *g_idata, int *g_odata, unsigned int n) {
    // Calcola l'ID del thread e l'indice globale dei dati
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x * 8 + threadIdx.x;

    // Converte il puntatore ai dati globali nel puntatore ai dati locali di questo blocco
    int *idata = g_idata + blockIdx.x * blockDim.x * 8;

    // Srotolamento del ciclo di riduzione
    if (idx + 7*blockDim.x < n) {
        int a1 = g_idata[idx];
        int a2 = g_idata[idx + blockDim.x];
        int a3 = g_idata[idx + 2 * blockDim.x];
        int a4 = g_idata[idx + 3 * blockDim.x];
        int b1 = g_idata[idx + 4 * blockDim.x];
        int b2 = g_idata[idx + 5 * blockDim.x];
        int b3 = g_idata[idx + 6 * blockDim.x];
        int b4 = g_idata[idx + 7 * blockDim.x];
        g_idata[idx] = a1 + a2 + a3 + a4 + b1 + b2 + b3 + b4;
    }
    __syncthreads();
```

Ottimizzazione Avanzata: 5) Unrolling Completo

Implementazione Riduzione con Unrolling Completo + Template - Fattore 8

```
    // Unrolling completo della riduzione
    if (blockDim.x >= 1024 && tid < 512) idata[tid] += idata[tid + 512];
    __syncthreads();
    if (blockDim.x >= 512 && tid < 256) idata[tid] += idata[tid + 256];
    __syncthreads();
    if (blockDim.x >= 256 && tid < 128) idata[tid] += idata[tid + 128];
    __syncthreads();
    if (blockDim.x >= 128 && tid < 64) idata[tid] += idata[tid + 64];
    __syncthreads();

    // Unrolling Warp (nessuna sync necessaria)
    if (tid < 32) {
        volatile int *vsmem = idata;
        vsmem[tid] += vsmem[tid + 32];
        vsmem[tid] += vsmem[tid + 16];
        vsmem[tid] += vsmem[tid + 8];
        vsmem[tid] += vsmem[tid + 4];
        vsmem[tid] += vsmem[tid + 2];
        vsmem[tid] += vsmem[tid + 1];
    }

    // Scrive il risultato di questo blocco nella memoria globale
    if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```

Branch Overhead dovuto alle diverse istruzioni condizionali. Si possono eliminare?

Riduzione Parallela - Confronto fra Kernel

NVIDIA Nsight Compute

Dim. Array (2^{26}), Dim. Blocco (512)

GPU RTX 3090
CPU i9-10920X

Kernel	Dim. Griglia	Istruzioni Eseguite (M)	Load Memory Through. (GB/s)	Branch Efficiency (%)	Runtime (ms)	Cumulative Speedup
Recursive Interleaved (CPU)	–	–	–	–	130.888	–
Neighbored (Divergence)	(131072)	612,10	123,56	68,75	2,332*	56.13
Neighbored (No Divergence)	(131072)	241,96	230,84	98,36	1,325*	98.81
Interleaved	(131072)	205,78	266,01	98,36	1,175*	111.38
Unroll 2 Blocks	(65536)	111,28	471,18	98,36	0,683*	191.55
Unroll 4 Blocks	(32768)	58,79	694,13	98,36	0,484*	270.30
Unroll 8 Blocks	(16384)	34,37	768,26	98,44	0,423*	309.29
Unroll 8 Blocks + Last Warp	(16384)	22,30	767,86	100	0,414*	316.42
Unroll 8 Blocks + Loop + Last Warp	**(16384)**	**21,04**	**767,49**	**100**	**0,412***	**317.52**

* Tempistiche rilevate con Timer CPU: (Calcolo Somme Parziali [Device] + Calcolo Somma Finale [Host])

Ottimizzazione Avanzata: 6) Unrolling con Template

Implementazione Riduzione con Unrolling Completo + Template - Fattore 8

```
template <unsigned int iBlockSize>
__global__ void reduceCompleteUnroll(int *g_idata, int *g_odata, unsigned int n) {
    // ... (codice precedente)
    if (iBlockSize >= 1024 && tid < 512) {
        idata[tid] += idata[tid + 512];
    }
    __syncthreads();

    if (iBlockSize >= 512 && tid < 256) {
        idata[tid] += idata[tid + 256];
    }
    __syncthreads();

    if (iBlockSize >= 256 && tid < 128) {
        idata[tid] += idata[tid + 128];
    }
    __syncthreads();

    if (iBlockSize >= 128 && tid < 64) {
        idata[tid] += idata[tid + 64];
    }
    __syncthreads();

    // ... (continua)
```

Specificare la dimensione del blocco come **parametro template**.

- Gli `if` statement (codice in **rosso**) vengono valutati durante la compilazione (**compile time**).
- Il codice non necessario viene **eliminato** automaticamente dal compilatore.
- Riduzione dell'overhead dei branch.
- Risultato: loop interno **più efficiente**.

Ottimizzazione Avanzata: 6) Unrolling con Template

Invocazione dei Kernel con Template

- Il kernel deve essere chiamato utilizzando una struttura **switch-case**.
- Ciò consente al compilatore di ottimizzare automaticamente il codice per dimensioni specifiche dei blocchi.
- Tuttavia, significa che `reduceCompleteUnroll` può essere lanciato solo con **dimensioni dei blocchi predefinite**.

Invocazione Lato Host

```
switch (blocksize) {
    case 1024:
        reduceCompleteUnroll<1024><<<grid.x/8, block>>>(d_idata, d_odata, size);
        break;
    case 512:
        reduceCompleteUnroll<512><<<grid.x/8, block>>>(d_idata, d_odata, size);
        break;
    case 256:
        reduceCompleteUnroll<256><<<grid.x/8, block>>>(d_idata, d_odata, size);
        break;
    case 128:
        reduceCompleteUnroll<128><<<grid.x/8, block>>>(d_idata, d_odata, size);
        break;
    case 64:
        reduceCompleteUnroll<64><<<grid.x/8, block>>>(d_idata, d_odata, size);
        break;
}
```

Riduzione Parallela - Confronto fra Kernel

NVIDIA Nsight Compute

Dim. Array (2^{26}), Dim. Blocco (512)

GPU RTX 3090
CPU i9-10920X

Kernel	Dim. Griglia	Istruzioni Eseguite (M)	Load Memory Through. (GB/s)	Branch Efficiency (%)	Runtime (ms)	Cumulative Speedup
Recursive Interleaved (CPU)	–	–	–	–	130.888	–
Neighbored (Divergence)	(131072)	612,10	123,56	68,75	2,332*	56.13
Neighbored (No Divergence)	(131072)	241,96	230,84	98,36	1,325*	98.81
Interleaved	(131072)	205,78	266,01	98,36	1,175*	111.38
Unroll 2 Blocks	(65536)	111,28	471,18	98,36	0,683*	191.55
Unroll 4 Blocks	(32768)	58,79	694,13	98,36	0,484*	270.30
Unroll 8 Blocks	(16384)	34,37	768,26	98,44	0,423*	309.29
Unroll 8 Blocks + Last Warp	(16384)	22,30	767,86	100	0,414*	316.42
Unroll 8 Blocks + Loop + Last Warp	(16384)	21,04	767,49	100	0,412*	317.52
Templetized Kernel	**(16384)**	**18,93**	**767,56**	**100**	**0,411***	**318.07**

* Tempistiche rilevate con Timer CPU: (Calcolo Somme Parziali [Device] + Calcolo Somma Finale [Host])

*Come possiamo migliorare le prestazioni **riducendo l'accesso alla memoria globale**?*

Ottimizzazione Avanzata: 7) Unrolling con Templates + SMEM

Utilizzo della Shared Memory

- L'ottimizzazione si ottiene minimizzando gli accessi alla memoria globale attraverso l'utilizzo della **shared memory** (SMEM) on-chip, riducendo così la latenza e aumentando il throughput delle operazioni di riduzione.

Implementazione Riduzione con Unrolling Completo + Template + SMEM - Fattore 8

```
template <unsigned int iBlockSize>
__global__ void reduceCompleteUnrollShared(int *g_idata, int *g_odata, unsigned int n) {
    // Shared memory per il blocco
    __shared__ int smem[iBlockSize];

    // Set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x * 8 + threadIdx.x;

    // Inizializzazione variabile temporanea
    int tmp = 0;
    // continua
```

Ottimizzazione Avanzata: 7) Unrolling con Templates + SMEM

Implementazione Riduzione con Unrolling Completo + Template + SMEM - Fattore 8

```
// Carica i dati dalla Global Memory alla Shared Memory
// Unrolling del ciclo di riduzione
if (idx + 7*blockDim.x < n) {
    int a1 = g_idata[idx];
    int a2 = g_idata[idx + blockDim.x];
    int a3 = g_idata[idx + 2 * blockDim.x];
    int a4 = g_idata[idx + 3 * blockDim.x];
    int b1 = g_idata[idx + 4 * blockDim.x];
    int b2 = g_idata[idx + 5 * blockDim.x];
    int b3 = g_idata[idx + 6 * blockDim.x];
    int b4 = g_idata[idx + 7 * blockDim.x];
    tmp = a1 + a2 + a3 + a4 + b1 + b2 + b3 + b4;
}
smem[tid] = tmp;
__syncthreads();
// continua
```

Ottimizzazione Avanzata: 7) Unrolling con Templates + SMEM

Implementazione Riduzione con Unrolling Completo + Template + SMEM - Fattore 8

```
// Riduzione In-place nella Shared Memory
if (iBlockSize >= 1024 && tid < 512) {
    smem[tid] += smem[tid + 512];
}
__syncthreads();

if (iBlockSize >= 512 && tid < 256) {
    smem[tid] += smem[tid + 256];
}
__syncthreads();

if (iBlockSize >= 256 && tid < 128) {
    smem[tid] += smem[tid + 128];
}
__syncthreads();

if (iBlockSize >= 128 && tid < 64) {
    smem[tid] += smem[tid + 64];
}
__syncthreads();

// continua
```

Ottimizzazione Avanzata: 7) Unrolling con Templates + SMEM

Implementazione Riduzione con Unrolling Completo + Template + SMEM - Fattore 8

```
    // Unrolling Warp (nessuna sync necessaria)
    if (tid < 32) {
        volatile int *vsmem = smem;
        vsmem[tid] += vsmem[tid + 32];
        vsmem[tid] += vsmem[tid + 16];
        vsmem[tid] += vsmem[tid + 8];
        vsmem[tid] += vsmem[tid + 4];
        vsmem[tid] += vsmem[tid + 2];
        vsmem[tid] += vsmem[tid + 1];
    }

    // Scrittura del Risultato nella Memoria Globale
    if (tid == 0) {
        g_odata[blockIdx.x] = smem[0];
    }
}
```

Riduzione Parallela - Confronto fra Kernel

NVIDIA Nsight Compute

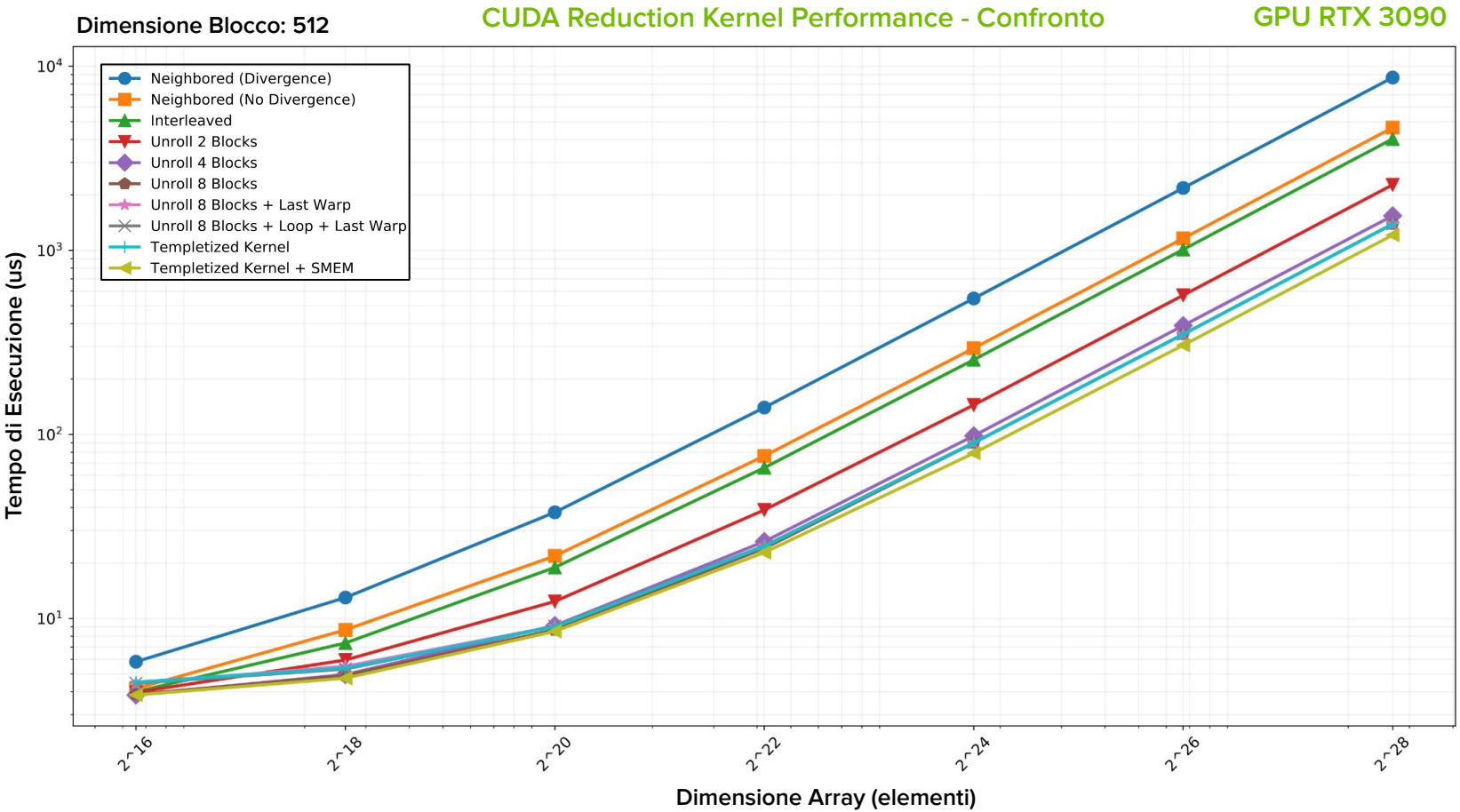
Dim. Array (2^{26}), Dim. Blocco (512)

GPU RTX 3090
CPU i9-10920X

Kernel	Dim. Griglia	Istruzioni Eseguite (M)	Load Memory Through. (GB/s)	Branch Efficiency (%)	Runtime (ms)	Cumulative Speedup
Recursive Interleaved (CPU)	–	–	–	–	130.888	–
Neighbored (Divergence)	(131072)	612,10	123,56	68,75	2,332*	56.13
Neighbored (No Divergence)	(131072)	241,96	230,84	98,36	1,325*	98.81
Interleaved	(131072)	205,78	266,01	98,36	1,175*	111.38
Unroll 2 Blocks	(65536)	111,28	471,18	98,36	0,683*	191.55
Unroll 4 Blocks	(32768)	58,79	694,13	98,36	0,484*	270.30
Unroll 8 Blocks	(16384)	34,37	768,26	98,44	0,423*	309.29
Unroll 8 Blocks + Last Warp	(16384)	22,30	767,86	100	0,414*	316.42
Unroll 8 Blocks + Loop + Last Warp	(16384)	21,04	767,49	100	0,412*	317.52
Templetized Kernel	(16384)	18,93	767,56	100	0,411*	318.07
Templetized Kernel + SMEM	**(16384)**	**18,04**	**879,96**	**100**	**0,360***	**363.58**

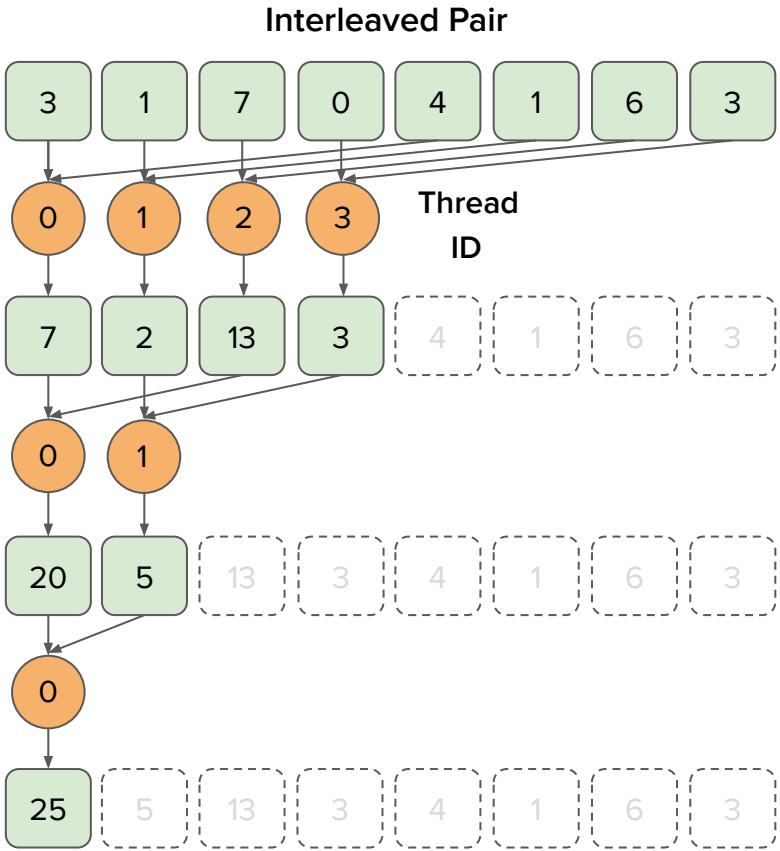
* Tempistiche rilevate con Timer CPU: (Calcolo Somme Parziali [Device] + Calcolo Somma Finale [Host])

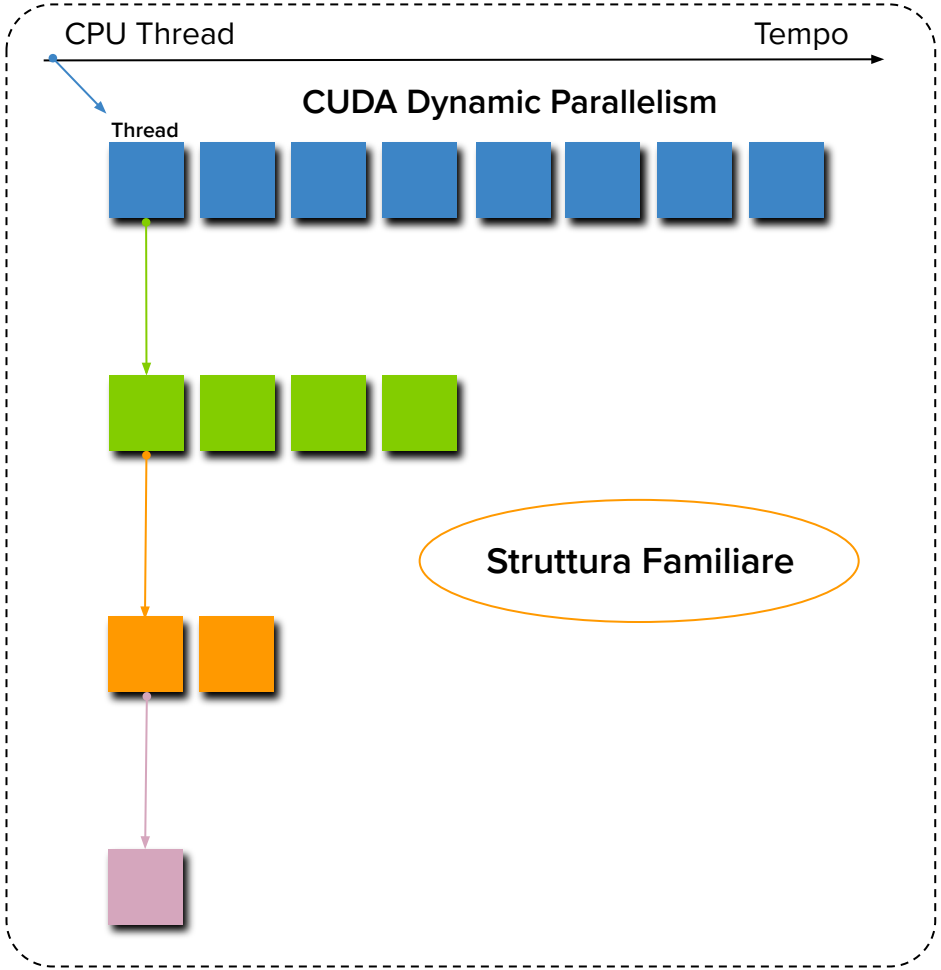
Riduzione Parallela - Confronto fra Kernel



Come possiamo implementare la riduzione parallela rispettando la sua ***gerarchia naturalmente ricorsiva****?*

Nested Reduction in CUDA





Nested Reduction in CUDA

Descrizione Generale

- La riduzione può essere espressa naturalmente come una **funzione ricorsiva**.
- In CUDA, la riduzione ricorsiva può essere implementata in modo altrettanto semplice rispetto a C, grazie al **CUDA Dynamic Parallelism**.

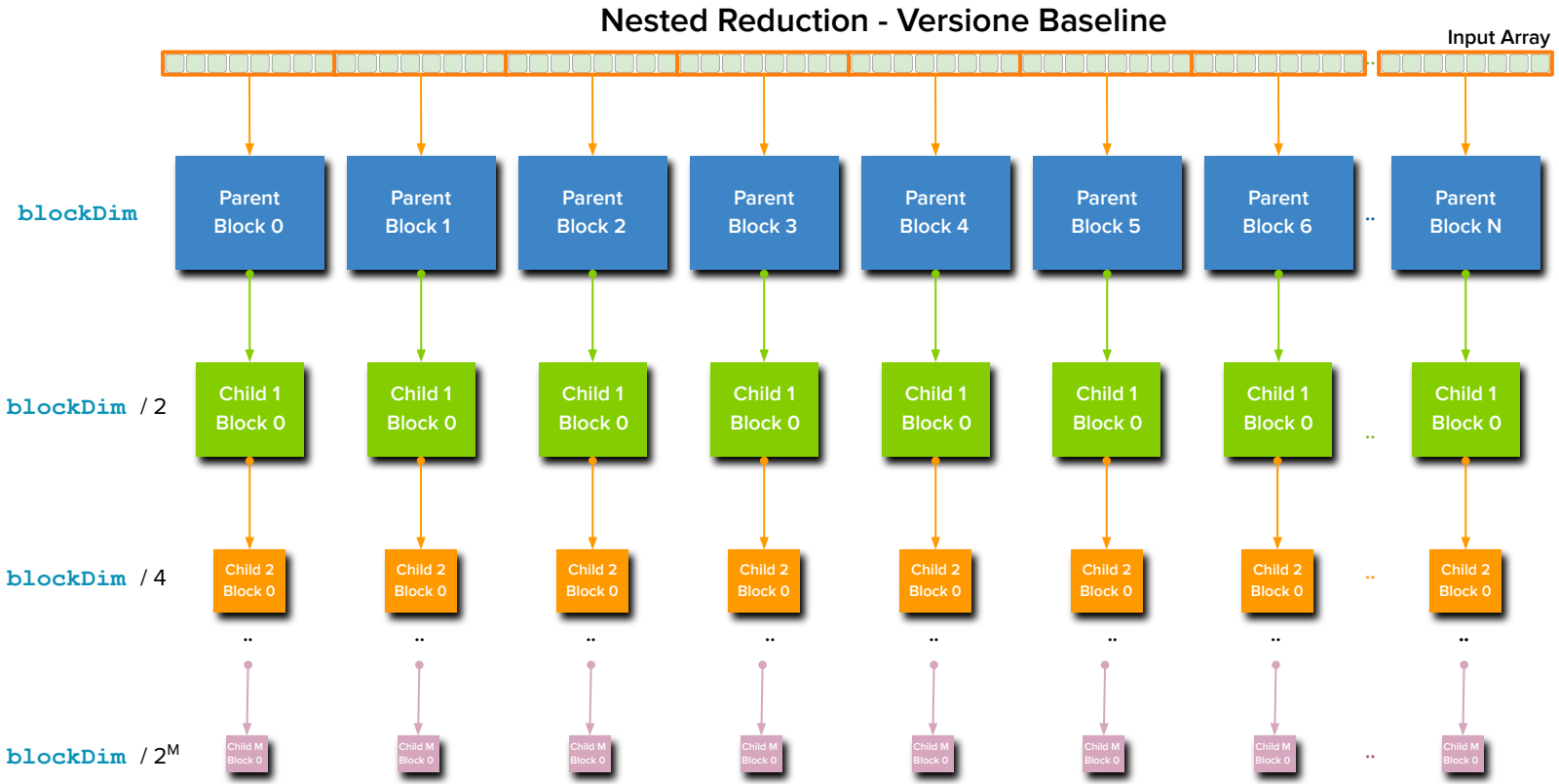
Implementazione del Kernel (Versione Baseline)

- **Passaggio 1**: Conversione dell'indirizzo di memoria globale `g_idata` in un indirizzo locale per ogni blocco di thread.
- **Passaggio 2**: Verifica della condizione di arresto:
  - **Se è una foglia** nell'albero di esecuzione nidificato, i risultati vengono copiati nella memoria globale e il controllo ritorna al kernel padre.
  - **Se non è una foglia**, viene calcolata la dimensione della riduzione locale e metà dei thread esegue una riduzione in-place.

Gestione dei Thread (Versione Baseline)

- **Sincronizzazione**: I thread nel blocco si sincronizzano dopo la riduzione in-place.
- **Generazione del Child Grid**: Viene creato un grid figlio con metà dei thread del blocco corrente.
- **Punto di Barriera**: Si imposta una barriera per sincronizzare con il grid figlio.

Nested Reduction in CUDA



Nested Reduction in CUDA

Nested Reduction - Versione 1

```
__global__ void gpuRecursiveReduce(int *g_idata, int *g_odata, unsigned int isize) {
    unsigned int tid = threadIdx.x; // Imposta l'ID del thread

    // Converte il puntatore globale dei dati nel puntatore locale di questo blocco
    int *idata = g_idata + blockIdx.x * blockDim.x;
    int *odata = &g_odata[blockIdx.x];

    if (isize == 2 && tid == 0) { // Condizione di arresto
        g_odata[blockIdx.x] = idata[0] + idata[1];
        return;
    }

    // Invocazione annidata
    int istride = isize >> 1;  // Equivalente a isize / 2
    if (istride > 1 && tid < istride) {
        // Riduzione in-place
        idata[tid] += idata[tid + istride];
    }

    __syncthreads(); // Sincronizzazione a livello di blocco

    // Invocazione annidata per generare le child grid
    if (tid == 0) {
        gpuRecursiveReduce<<<1, istride>>>(idata, odata, istride);  // Lancia un nuovo kernel

        cudaDeviceSynchronize();  // Sincronizza tutte le griglie figlie lanciate in questo blocco
    }
    __syncthreads(); // Sincronizza nuovamente a livello di blocco
}
```

Nested Reduction in CUDA

Nested Reduction - Versione 1

```
__global__ void gpuRecursiveReduce(int *g_idata, int *g_odata, unsigned int isize) {
    unsigned int tid = threadIdx.x; // Imposta l'ID del thread

    // Converte il puntatore globale dei dati nel puntatore locale di questo blocco
    int *idata = g_idata + blockIdx.x * blockDim.x;
    int *odata = &g_odata[blockIdx.x];

    if (isize == 2 && tid == 0) { // Condizione
        g_odata[blockIdx.x] = idata[0] + idata
        return;
    }

    // Invocazione annidata
    int istride = isize >> 1;  // Equivalente a
    if (istride > 1 && tid < istride) {
        // Riduzione in-place
        idata[tid] += idata[tid + istride];
    }

    __syncthreads(); // Sincronizzazione a livello di blocco

    // Invocazione annidata per generare l
    if (tid == 0) {
        gpuRecursiveReduce<<<1, istride>>>
        cudaDeviceSynchronize(); // Sincr
    }
    __syncthreads(); // Sincronizza nuovam
}
```

- **Sincronizzazione eccessiva**: `__syncthreads()` e `cudaDeviceSynchronize()` bloccano il parallelismo, causando attese continue.
- Al lancio, i kernel child vedono la memoria in modo consistente con il thread padre e, poiché usano solo i suoi valori per la riduzione parziale, la sincronizzazione nel blocco prima del lancio è **superflua**.

- **Esplosione di kernel**: La ricorsione genera un numero eccessivo di lanci di kernel, sovraccaricando la GPU.
- **Basso utilizzo dei thread**: I thread (metà per blocco) rimangono inutilizzati con la diminuzione della dimensione del problema.

Nested Reduction in CUDA

Nested Reduction - Versione 2 (Rimozione della Sincronizzazione)

```
__global__ void gpuRecursiveReduceNosync(int *g_idata, int *g_odata, unsigned int isize) {
  unsigned int tid = threadIdx.x; // Imposta l'ID del thread

  // Converte il puntatore globale dei dati nel puntatore locale di questo blocco
  int *idata = g_idata + blockIdx.x * blockDim.x;
  int *odata = &g_odata[blockIdx.x];

  // Condizione di arresto
  if (isize == 2 && tid == 0) {
      g_odata[blockIdx.x] = idata[0] + idata[1];
      return;
  }
  // NO Synchronization
  // Invocazione annidata
  int istride = isize >> 1;
  if (istride > 1 && tid < istride) {
      idata[tid] += idata[tid + istride];

      // Lancia un nuovo kernel solo per il thread 0
      if (tid == 0)
          gpuRecursiveReduceNosync <<<1, istride>>>(idata, odata, istride);
  }
}
```

Nested Reduction in CUDA

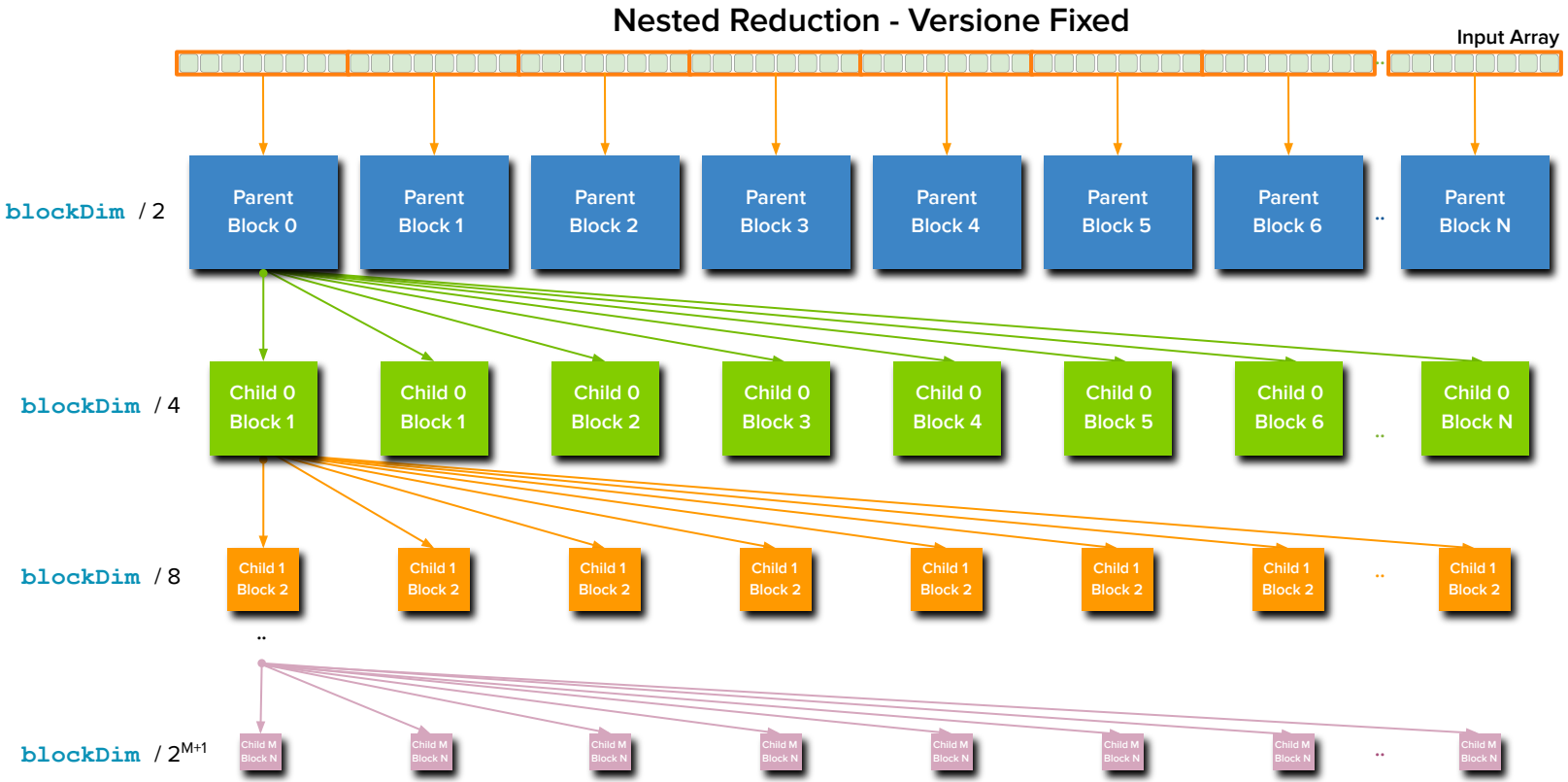
Nested Reduction - Versione 2 (Rimozione della Sincronizzazione)

```
__global__ void gpuRecursiveReduceNosync(int *g_idata, int *g_odata, unsigned int isize) {
  unsigned int tid = threadIdx.x; // Imposta l'ID del thread

  // Converte il puntatore globale dei dati nel puntatore locale di questo blocco
  int *idata
  int *odata

  // Condizio
  if (isize =
      g_odata
      return;
  }
  // NO Synch
  // Invocazi
  int istride
  if (istride
      idata[t

      // Lancia un nuovo kernel solo per il thread 0
      if (tid == 0)
          gpuRecursiveReduceNosync <<<1, istride>>>(idata, odata, istride);
  }
}
```

Rimozione della Sincronizzazione

- Ogni thread figlio **eredita** una vista coerente della memoria dal genitore.
- Ogni thread opera su una porzione di dati **indipendente**, scrivendo il risultato in una locazione di memoria univoca.
- **Non si verificano race condition**, quindi l'ordine di esecuzione dei thread è irrilevante.
- Si elimina l'attesa forzata di tutti i thread, riducendo quindi l'esecuzione.
- **Problema Rimanente:** Ad ogni livello ricorsivo, metà dei thread di ogni blocco diventa **inattiva**. Risorse della GPU sprecate, riduzione dell'efficienza complessiva.

Nested Reduction in CUDA



Nested Reduction in CUDA

Nested Reduction - Versione 3 (Ottimizzazione dei Thread Inattivi)

```
__global__ void gpuRecursiveReduceFixed(int *g_idata, int *g_odata, int iStride, int const iDim) {
    // Converte il puntatore globale dei dati nel puntatore locale di questo blocco
    int *idata = g_idata + blockIdx.x * iDim;

    // Condizione di arresto
    if (iStride == 1 && threadIdx.x == 0) {
        g_odata[blockIdx.x] = idata[0] + idata[1];
        return;
    }

    // Riduzione in-place
    idata[threadIdx.x] += idata[threadIdx.x + iStride];

    // Invocazione annidata per generare griglie figlie
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        gpuRecursiveReduce2 <<<gridDim.x, iStride/2>>>(g_idata, g_odata, iStride /2, iDim);
    }
}

int main(int argc, char **argv) {
    // ...

    gpuRecursiveReduceFixed<<<grid, block/2>>>(d_idata, d_odata, block/2, block);

    // Continua
}
```

Permette ad ogni thread di calcolare il **corretto offset in memoria globale** per la sua porzione di lavoro

Metà dei thread per blocco, zero sprechi.

Nested Reduction - Confronto fra Kernel

NVIDIA Nsight Compute

Dim. Array (2^{22}), Dim. Blocco (512)

GPU RTX 3090
CPU i9-10920X

Kernel	Istruzioni Eseguite (M)	No Eligible Warps (%)	Stall Barrier (Cicli per Warp)	Runtime (ms)	Cumulative Speedup
Recursive Interleaved (CPU)	–	–	–	8,448	–
Neighbored (Divergence)	38,39	38,27	3,30	0,514*	16,44
Nested Sync	1497,24	92,12	25,96	109,319*	0,078
Nested NoSyn	875,15	87,46	0,02	42,94*	0,197
Nested Fixed	138,32	43,17	0,24	0,734*	11,51

* Tempistiche rilevate con Timer CPU: (Calcolo Somme Parziali [Device] + Calcolo Somma Finale [Host])

Nota: Gli strumenti nsys e nvprof non supportano il tracciamento dei kernel CDP per le architetture GPU Volta e superiori

- **Riduce l'overhead** dovuto al gran numero di invocazioni delle child grid.
- Tutti i thread inattivi vengono rimossi da ogni lancio del kernel.
- Mantiene lo **stesso livello di parallelismo** delle versioni precedenti.

Riferimenti Bibliografici

Testi Generali

- Cheng, J., Grossman, M., McKercher, T. (2014). **Professional CUDA C Programming**. Wrox Pr Inc. (1^ edizione)
- Kirk, D. B., Hwu, W. W. (2013). **Programming Massively Parallel Processors**. Morgan Kaufmann (3^ edizione)

NVIDIA Docs

- CUDA Programming:
  http://docs.nvidia.com/cuda/cuda-c-programming-guide/
- CUDA C Best Practices Guide
  http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/
- Optimizing Parallel Reduction in CUDA (Mark Harris)
  https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Risorse Online

- Corso GPU Computing (Prof. G. Grossi): Dipartimento di Informatica, Università degli Studi di Milano
  - http://gpu.di.unimi.it/lezioni.html