Bachelor forsvar

En ISPC bagende til Futhark

Kristoffer A. Kortbaek

23. juni 2022

Fokuspunkt

1. Multicore “flattening”

Fokuspunkt

1. Multicore "flattening"
2. Udnyttelse af data parallelisme for `reduce`

Fokuspunkt

1. Multicore "flattening"
2. Udnyttelse af data parallelisme for `reduce`
3. Hvordan påvirker multicore flattening de data parallelle SOACs

Multicore flattening - sekventielle og parallelle SOACs

Et Futhark matrix-vector multiplication program kunne skrives som

```
def main m n  (X:[m][n]i32)  (v:[n]i32)  =
  map (\x ->
    reduce (+) 0 (map2 (*) x v)
  ) X
```

Multicore flattening - sekventielle og parallelle SOACs

Et Futhark matrix-vector multiplication program kunne skrives som

```
def main m n  (X:[m][n]i32)  (v:[n]i32)  =
  map (\x ->
    reduce (+) 0 (map2 (*) x v)
  ) X
```

Hver og bemærke ved programmet

Multicore flattening - sekventielle og parallelle SOACs

Et Futhark matrix-vector multiplication program kunne skrives som

```
def main m n  (X:[m][n]i32)  (v:[n]i32)  =
  map (\x ->
    reduce (+) 0 (map2 (*) x v)
  ) X
```

Hver og bemærke ved programmet

1. En ydre parallel `map`

Multicore flattening - sekventielle og parallelle SOACs

Et Futhark matrix-vector multiplication program kunne skrives som

```
def main m n  (X:[m][n]i32)  (v:[n]i32)  =
  map (\x ->
    reduce (+) 0 (map2 (*) x v)
  ) X
```

Hver og bemærke ved programmet

1. En ydre parallel `map`
2. En *nested* `redomap`

Multicore flattening - sekventielle og parallelle SOACs

Et Futhark matrix-vector multiplication program kunne skrives som

```
def main m n  (X:[m][n]i32)  (v:[n]i32)  =
  map (\x ->
    reduce (+) 0 (map2 (*) x v)
  ) X
```

Hver og bemærke ved programmet

1. En ydre parallel `map`
2. En *nested* `redomap`

Hvordan kan vi afvikle det parallelt på en CPU?

Multicore flattening - sekventielle og parallelle SOACs

Et Futhark matrix-vector multiplication program kunne skrives som

```
def main m n  (X:[m][n]i32)  (v:[n]i32)  =
  map (\x ->
    reduce (+) 0 (map2 (*) x v)
  ) X
```

Hver og bemærke ved programmet

1. En ydre parallel `map`
2. En *nested* `redomap`

Hvordan kan vi afvikle det parallelt på en CPU?

- Multicore bagenden kan vælge to metode

Multicore parallelle SOACs

Kør `map` parallelt, og den indre `reduce` kan blive scheduleret ud til ledige tråde

```
parfor{
  for(int map_i = 0; i < m; map_i++) {
    // schedule work for the nested reduce
    mem[map_i] = reduce_res;
  }
}
parfor{
  for(int red_i = 0;  red_i < n; red_i++) {
    int x = mem[map_i * n + red_i];
    int v_i = mem[red_i];
    int mul_res = x * v_i;
    red_res = mulres + red_res;
  }
}
```

Multicore parallelle SOACs

Kør `map` parallelt, og den indre `reduce` kan blive scheduleret ud til ledige tråde

```
parfor{
  for(int map_i = 0; i < m; map_i++) {
    // schedule work for the nested reduce
    mem[map_i] = reduce_res;
  }
}
parfor{
  for(int red_i = 0;  red_i < n; red_i++) {
    int x = mem[map_i * n + red_i];
    int v_i = mem[red_i];
    int mul_res = x * v_i;
    red_res = mulres + red_res;
  }
}
```

Observation: En CPU har forholdsvist få kerner, og det er nemt og "fylde" dem med arbejde

Multicore flattening af SOACs

Man kan kører kun selve `map` i parallel på alle de logiske kerner, og lade `reduce` blive sekvententialiseret

```
parfor(i = 0; i < n_cores; i++) {
  for(int map_i = 0; i < m; map_i++) {
    int red_res = 0; //Neutral element
    for(int red_i = 0;  red_i < n; red_i++) {
      int x = mem[map_i * n + red_i];
      int v_i = mem[red_i];
      int mul_res = x * v_i;
      red_res = mulres + red_res;
    }
    mem[map_i] = red_res;
  }
}
```

Multicore flattening af SOACs

Man kan kører kun selve `map` i parallel på alle de logiske kerner, og lade `reduce` blive sekvententialiseret

```
parfor(i = 0; i < n_cores; i++) {
  for(int map_i = 0; i < m; map_i++) {
    int red_res = 0; //Neutral element
    for(int red_i = 0;  red_i < n; red_i++) {
      int x = mem[map_i * n + red_i];
      int v_i = mem[red_i];
      int mul_res = x * v_i;
      red_res = mulres + red_res;
    }
    mem[map_i] = red_res;
  }
}
```

Hvis $n_{cores} < m$, så bliver alle CPU kerner mættede

Multicore flattening - sekventielle og parallelle SOACs

De to versioner bruges af multicore bagenden, og bliver valgt på runtime

Multicore flattening - sekventielle og parallelle SOACs

De to versioner bruges af multicore bagenden, og bliver valgt på runtime

1. Den flattenede/sekventialiserede version vælges hvis der er nok iterationer på den yderste SOAC til at mætte CPU kernerne

Multicore flattening - sekventielle og parallelle SOACs

De to versioner bruges af multicore bagenden, og bliver valgt på runtime

1. Den flattenede/sekventialiserede version vælges hvis der er nok iterationer på den yderste SOAC til at mætte CPU kernerne
2. Den uberørte parallelle version vælges ellers.

Multicore flattening - sekventielle og parallelle SOACs

De to versioner bruges af multicore bagenden, og bliver valgt på runtime

1. Den flattenede/sekventialiserede version vælges hvis der er nok iterationer på den yderste SOAC til at mætte CPU kernerne
2. Den uberørte parallelle version vælges ellers.

Kan vi finde mere parallelisme?

Multicore flattening - sekventielle og parallelle SOACs

De to versioner bruges af multicore bagenden, og bliver valgt på runtime

1. Den flattenede/sekventialiserede version vælges hvis der er nok iterationer på den yderste SOAC til at mætte CPU kernerne
2. Den uberørte parallelle version vælges ellers.

Kan vi finde mere parallelisme?

- Udnyttelse af SIMD indenfor hver tråd

Udvidelse af multicore flattening

Brug ISPC til at udvide både de sekventielle og nestede SOACs

Udvidelse af multicore flattening

Brug ISPC til at udvide både de sekventielle og nestede SOACs

1. Hver tråds arbejde skal køre i ISPC - en såkaldt *ISPC kernel*

Udvidelse af multicore flattening

Brug ISPC til at udvide både de sekventielle og nestede SOACs

1. Hver tråds arbejde skal køre i ISPC - en såkaldt *ISPC kernel*
2. Benyt `foreach` hvor muligt

Udvidelse af multicore flattening

Brug ISPC til at udvide både de sekventielle og nestede SOACs

1. Hver tråds arbejde skal køre i ISPC - en såkaldt *ISPC kernel*
2. Benyt `foreach` hvor muligt

Vi prøvede forskellige måde at udnytte language-c-quote til at generere selve ISPC koden

Udvidelse af multicore flattening

Brug ISPC til at udvide både de sekventielle og nestede SOACs

1. Hver tråds arbejde skal køre i ISPC - en såkaldt *ISPC kernel*
2. Benyt `foreach` hvor muligt

Vi prøvede forskellige måde at udnytte language-c-quote til at generere selve ISPC koden

1. Generer C kode og misbrug makroer til at lave ISPC kode

```
#define auto uniform
auto int foo = free->foo;
```

Udvidelse af multicore flattening

Brug ISPC til at udvide både de sekventielle og nestede SOACs

1. Hver tråds arbejde skal køre i ISPC - en såkaldt *ISPC kernel*
2. Benyt `foreach` hvor muligt

Vi prøvede forskellige måde at udnytte language-c-quote til at generere selve ISPC koden

1. Generer C kode og misbrug makroer til at lave ISPC kode

```
#define auto uniform
auto int foo = free->foo;
```

2. Udvid language-c-quote med ISPC sprog konstruktioner

```
[C.cstm|foreach ($foreachiters:bounds) {$items:body}|]
```

Udvidelse af multicore flattening

Brug ISPC til at udvide både de sekventielle og nestede SOACs

1. Hver tråds arbejde skal køre i ISPC - en såkaldt *ISPC kernel*
2. Benyt `foreach` hvor muligt

Vi prøvede forskellige måde at udnytte language-c-quote til at generere selve ISPC koden

1. Generer C kode og misbrug makroer til at lave ISPC kode

```
#define auto uniform
auto int foo = free->foo;
```

2. Udvid language-c-quote med ISPC sprog konstruktioner

```
[C.cstm|foreach ($foreachiters:bounds) {$items:body}|]
```

3. Brug ecaped statements

```
[C.cstms|$escstm:("foreach (i=0 ... end)") { $items:body
    }|]
```

Algoritmer til at udvide Futhark MC flattening

Hvordan kan man vektorisere de to forskellige versioner af en SOAC?

Algoritmer til at udvide Futhark MC flattening

Hvordan kan man vektorisere de to forskellige versioner af en SOAC?

1. Giv hver SOAC en specifik vectoriseret algoritme afhængigt af operatoren

Algoritmer til at udvide Futhark MC flattening

Hvordan kan man vektorisere de to forskellige versioner af en SOAC?

1. Giv hver SOAC en specifik vectoriseret algoritme afhængigt af operatoren
2. Særligt har redomap fire forskellige kodegenereringer

Algoritmer til at udvide Futhark MC flattening

Hvordan kan man vektorisere de to forskellige versioner af en SOAC?

1. Giv hver SOAC en specifik vectoriseret algoritme afhængigt af operatoren
2. Særligt har redomap fire forskellige kodegenereringer
   - kommutative `reduce`
   - normal(ikke kommutative) `reduce`
   - `reduce` på en "mapped" operator
   - Ingen brug af vektorisering

Kommutative reduktioner

Kommutative Reduktioner genereres der særligt effektiv kode for

1. Den binære operator kan ske i vilkårlig rækkefølge: $a + b = b + a$
2. Hver *program instance* kan arbejde på sit eget segment af inputtet

Simple reduktion over input array


```
reduce (+) 0 as
```

Kommutative reduktioner

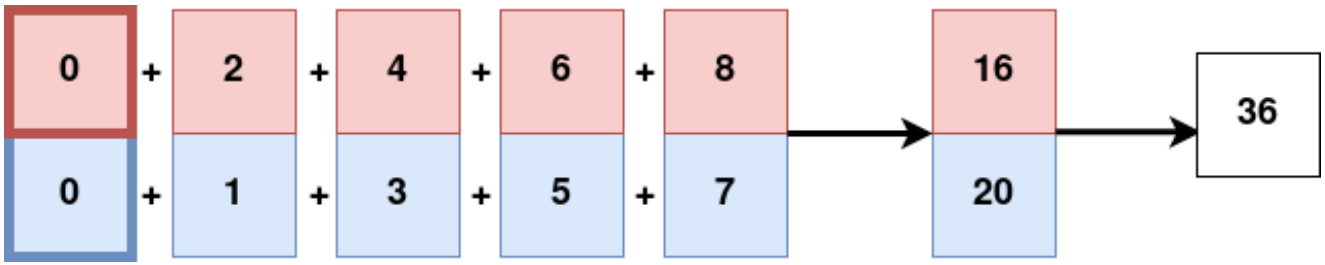
Kommutative Reduktioner genereres der særligt effektiv kode for

1. Den binære operator kan ske i vilkårlig rækkefølge: $a + b = b + a$
2. Hver *program instance* kan arbejde på sit eget segment af inputtet

Simple reduktion over input array


```
reduce (+) 0 as
```

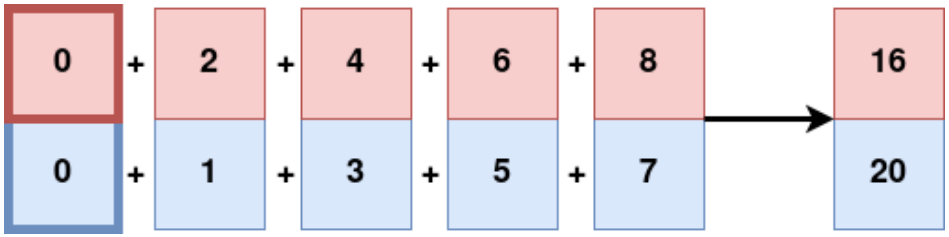
Den vectoriserede algoritme for kommutative `redomaps` har 2 skridt



Kommutativ reduction 1

Hver program instance laver sin egen reduktion.

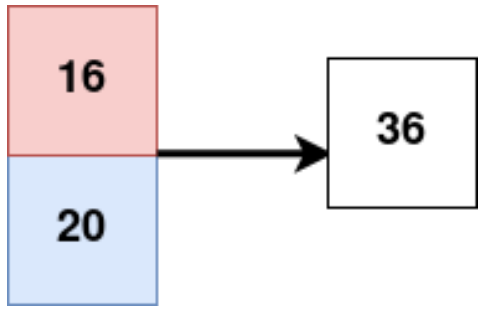
```
int acc = 0;
uniform int uni_acc = 0;
foreach(Reduce_i = 0 ... n) {
  int a = mem[Reduce_i];
  int res = acc + a;
  acc = res;
}
```



Kommutativ reduction 2

Derefter reduceres over resultaterne produceret af et *gang*

```
forneach_active{i = 0 ... programCount} {
  uniform int a = extract(acc, i);
  uniformint res = uni_acc + a
  uni_acc = res;
}
mem_out[out] = uni_acc;
```



Kommutativ reduktion restriktion

Vi har yderst god udnyttelse af SIMD

1. Hver program instance kører hele tiden

Kommutativ reduktion restriktion

Vi har yderst god udnyttelse af SIMD

1. Hver program instance kører hele tiden

Vi kan dog se at der ikke er garanti for rækkefølgen af reduktionen

Kommutativ reduktion restriktion

Vi har yderst god udnyttelse af SIMD

1. Hver program instance kører hele tiden

Vi kan dog se at der ikke er garanti for rækkefølgen af reduktionen

1. Den ene program instance regner $2 + 4 + 6 + 8$

Kommutativ reduktion restriktion

Vi har yderst god udnyttelse af SIMD

1. Hver program instance kører hele tiden

Vi kan dog se at der ikke er garanti for rækkefølgen af reduktionen

1. Den ene program instance regner $2 + 4 + 6 + 8$
2. Den anden regner $1 + 3 + 5 + 7$

Kommutativ reduktion restriktion

Vi har yderst god udnyttelse af SIMD

1. Hver program instance kører hele tiden

Vi kan dog se at der ikke er garanti for rækkefølgen af reduktionen

1. Den ene program instance regner $2 + 4 + 6 + 8$
2. Den anden regner $1 + 3 + 5 + 7$

Derved bliver vi nødt til at håndtere generelle associative operatorer anderledes

Associative redomap

Associative reduktioner leder til delvis SIMD udnyttelse

1. Den binære operator skal påføres i specifik rækkefølge
2. Vi kan ikke lade hver *program instance* arbejde på sit eget segment
3. Mapping functionen bliver vektoriseret og selve `reduce` kører sekventielt

Associative redomap

Associative reduktioner leder til delvis SIMD udnyttelse

1. Den binære operator skal påføres i specifik rækkefølge
2. Vi kan ikke lade hver *program instance* arbejde på sit eget segment
3. Mapping functionen bliver vektoriseret og selve `reduce` kører sekventielt

Vi håndterer korrekt nested parallelisme og sekventialiserede SOACs ved at *sekventialisere* selve `reduce` operatoren

```
uniform int scalar_accum = neutral;
foreach (i = 0 ... size) {
  int elem_i = arr[i];
  int mapped = map_op(elem_i);
  foreach_active (j) {
    uniform int elem_j = extract(mapped, j);
    scalar_accum = reduce_op(scalar_accum, elem_j); }
}
return scalar_accum;
```

Redomap på en *mapped* operator

Mapped operator

1. Kan være en `reduce` med en `map2` som operator
2. Særtilfælde hvor operatoren er på arrays, og vi generer ISPC kode

Redomap på en *mapped* operator

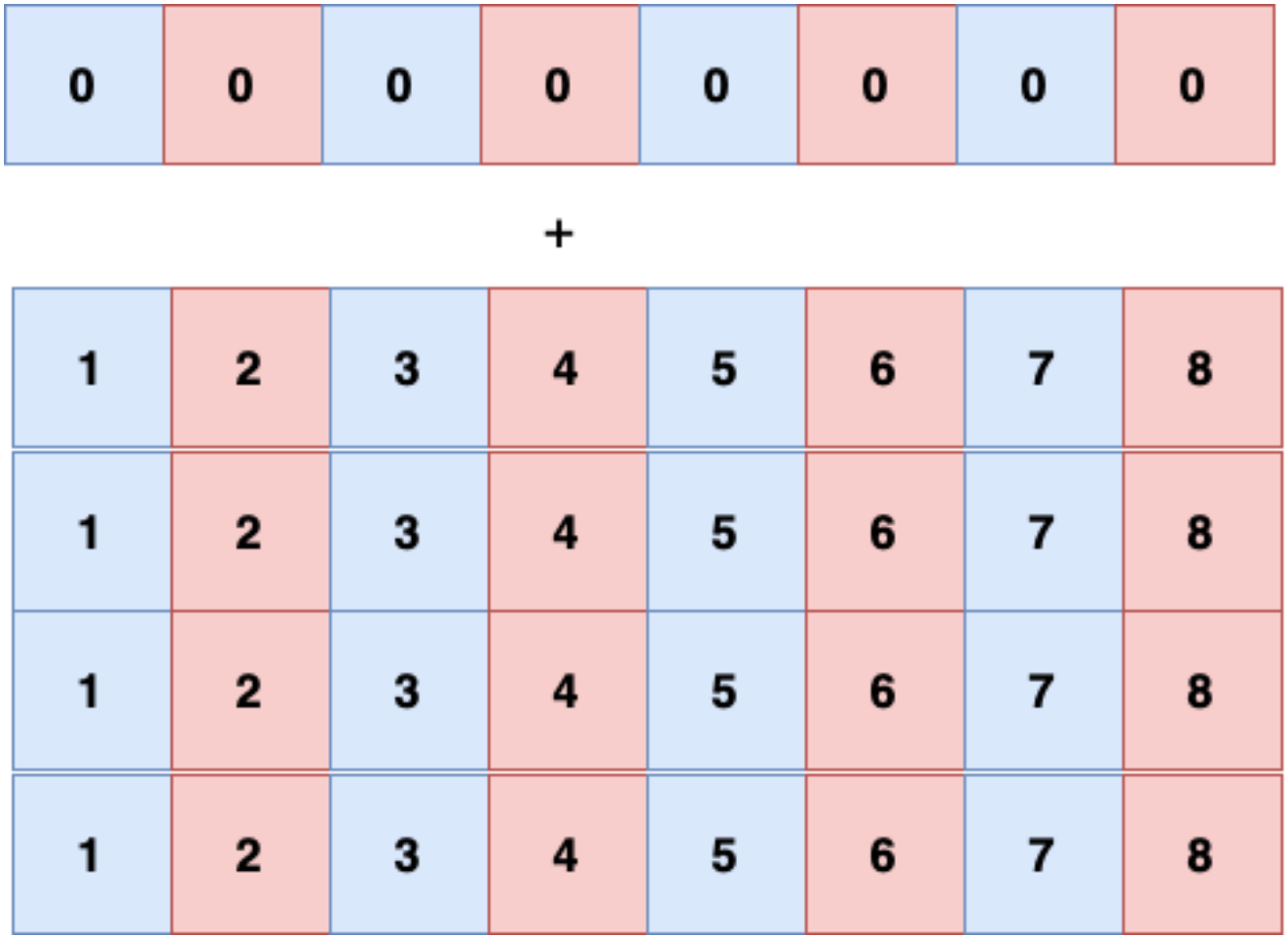
Mapped operator

1. Kan være en `reduce` med en `map2` som operator
2. Særtilfælde hvor operatoren er på arrays, og vi generer ISPC kode

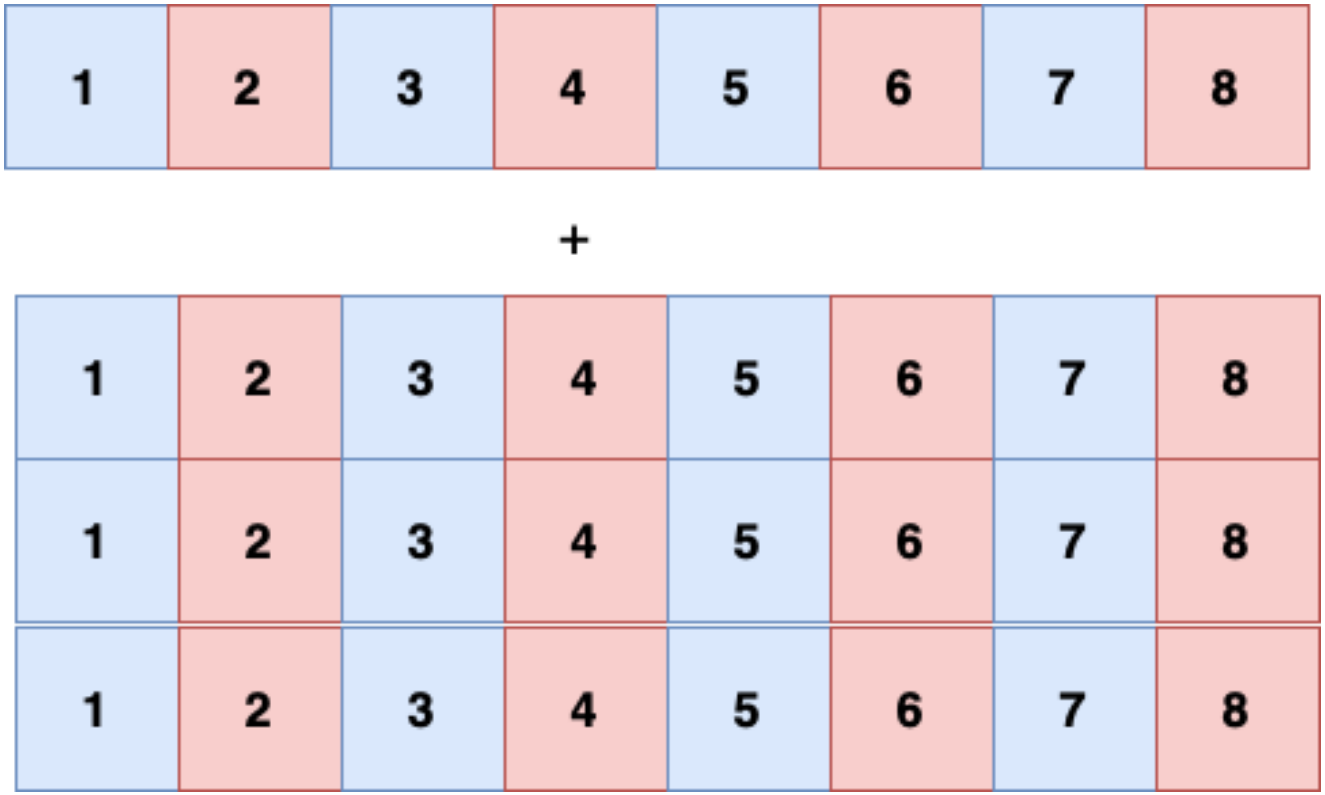
Simpelt program der tager en *mapped* operator i form af `map2`

```
entry mapped_plus [m][n] (X: [m][n]f32) =
  let a = replicate n 0
  in reduce (map2 (+)) a X
```

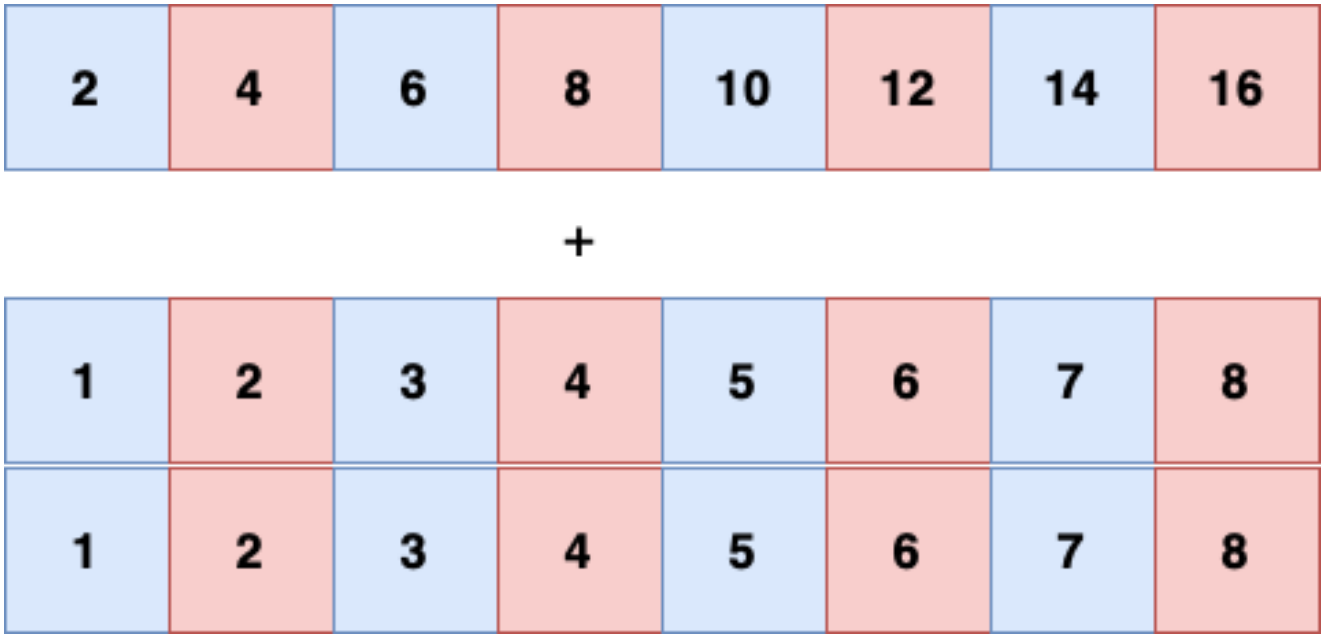
Redomap *mapped* operator eksempel



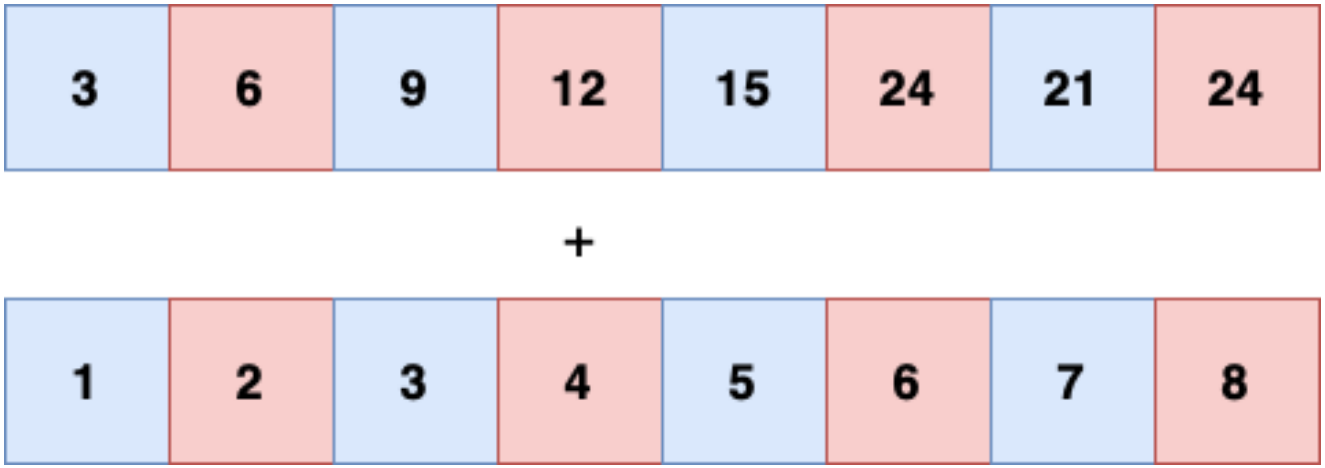
Redomap *mapped* operator eksempel



Redomap *mapped* operator eksempel



Redomap *mapped* operator eksempel



Redomap *mapped* operator eksempel

4	8	12	16	20	24	28	32

Redomap *mapped* operator algoritme

Observationer

1. Applikationen af $+$ er uafhængig af `map2` loop iterationer

Redomap *mapped* operator algoritme

Observationer

1. Applikationen af $+$ er uafhængig af `map2` loop iterationer
2. Istedet kører $+$ "på tværs" af hvert `reduce` loop iterationer

Redomap *mapped* operator algoritme

Observationer

1. Applikationen af $+$ er uafhængig af `map2` loop iterationer
2. Istedet kører $+$ "på tværs" af hvert `reduce` loop iterationer
3. Vi kan derved sikkert vektorisere selve $+$ operatoren

Redomap *mapped* operator algoritme

Observationer

1. Applikationen af + er uafhængig af `map2` loop iterationer
2. Istedet kører + "på tværs" af hvert `reduce` loop iterationer
3. Vi kan derved sikkert vektorisere selve + operatoren

redomap med en mapped operator

```
uniform int acc_mem[m] = //some memory
for(uniform int red_i = 0; scan_i < i; sca_i++) {
  foreach(nest_i = 0 ... m) {
    x_acc = acc_mem[nest_i];
    x = mem[red_i * n + nest_i];
    int res = x_acc + x;
    acc_mem[nest_i] = res;
  }
}
```

Kombinering af MC flattening og vektoriserede algoritmer

Er vores algoritmer en god udvidelse af MC flattening?

Kombinering af MC flattening og vektoriserede algoritmer

Er vores algoritmer en god udvidelse af MC flattening?

1. Sekventialiserede SOACs giver ofte *scatter* og *gather* operationer

Kombinering af MC flattening og vektoriserede algoritmer

Er vores algoritmer en god udvidelse af MC flattening?

1. Sekventialiserede SOACs giver ofte *scatter* og *gather* operationer
   Matrix vector multiplication programmet

```
def main m n  (X:[m][n]i32)  (v:[n]i32)  =
  map (\x ->
    reduce (+) 0 (map2 (*) x v)
  ) X
```

```
//Sekventialiseret/Flattened
foreach(map_i = 0 ... end) {
  for(uniform int red_i = 0;
      red_i < n; red_i++) {
    x = mem[map_i * n + red_i];
  }
}
```

```
//Nested reduce
uniform map_i; uniform n;
...
foreach(red_i = 0 ... end) {
  x = mem[map_i * n + red_i];
}
```

Kombinering af MC flattening og vektoriserede algoritmer

Er vores algoritmer en god udvidelse af MC flattening?

1. Sekventialiserede SOACs giver ofte *scatter* og *gather* operationer

```
//Sekventialiseret/Flattened
foreach(map_i = 0 ... end) {
  for(uniform int red_i = 0;
      red_i < n; red_i++) {
    x = mem[map_i * n + red_i];
  }
}
```

```
//Nested reduce
uniform map_i; uniform n;
...
foreach(red_i = 0 ... end) {
  x = mem[map_i * n + red_i];
}
```

For alle $n \neq 1$ vil det give en *gather* instruction da memory load ikke vil være *coherent*

Kombinering af MC flattening og vektoriserede algoritmer

Er vores algoritmer en god udvidelse af MC flattening?

1. Sekventialiserede SOACs giver ofte *scatter* og *gather* operationer

```
//Sekventialiseret/Flattened
foreach(map_i = 0 ... end) {
  for(uniform int red_i = 0;
      red_i < n; red_i++) {
    x = mem[map_i * n + red_i];
  }
}
```

```
//Nested reduce
uniform map_i; uniform n;
...
foreach(red_i = 0 ... end) {
  x = mem[map_i * n + red_i];
}
```

$$\texttt{map\_i} \cdot n = \langle 0, 1, 2, 3 \rangle \cdot 2 = \langle 0, 2, 4, 6 \rangle$$

Kombinering af MC flattening og vektoriserede algoritmer

Er vores algoritmer en god udvidelse af MC flattening?

1. Sekventialiserede SOACs giver ofte *scatter* og *gather* operationer

```
//Sekventialiseret/Flattened
foreach(map_i = 0 ... end) {
  for(uniform int red_i = 0;
      red_i < n; red_i++) {
    x = mem[map_i * n + red_i];
  }
}
```

```
//Nested reduce
uniform map_i; uniform n;
...
foreach(red_i = 0 ... end) {
  x = mem[map_i * n + red_i];
}
```

2. Nested parallelisme vil derimod også have coherent memory accesses

Kombinering af MC flattening og vektoriserede algoritmer

Er vores algoritmer en god udvidelse af MC flattening?

1. Sekventialiserede SOACs giver ofte *scatter* og *gather* operationer

```
//Sekventialiseret/Flattened
foreach(map_i = 0 ... end) {
  for(uniform int red_i = 0;
      red_i < n; red_i++) {
    x = mem[map_i * n + red_i];
  }
}
```

```
//Nested reduce
uniform map_i; uniform n;
...
foreach(red_i = 0 ... end) {
  x = mem[map_i * n + red_i];
}
```

2. Nested parallelisme vil derimod også have coherent memory accesses

$$red\_i = \langle 0, 1, 2, 3 \rangle$$

Kombinering af MC flattening og vektoriserede algoritmer

Er vores algoritmer en god udvidelse af MC flattening?

1. Sekventialiserede SOACs giver ofte *scatter* og *gather* operationer
2. Nested parallelisme vil derimod også have coherent memory accesses

Benchmark		Multicore (ms)	Ours (ms)	Speedup
$m = 100, n = 100000$		1	3	x0.44
$m = 6, n = 10000000$		12	12	x0.97

Kombinering af MC flattening og vektoriserede algoritmer

Er vores algoritmer en god udvidelse af MC flattening?

1. Sekventialiserede SOACs giver ofte *scatter* og *gather* operationer
2. Nested parallelisme vil derimod også have coherent memory accesses
3. Soacs med mapped operator kan give loads/store

Kombinering af MC flattening og vektoriserede algoritmer

Er vores algoritmer en god udvidelse af MC flattening?

1. Sekventialiserede SOACs giver ofte *scatter* og *gather* operationer
2. Nested parallelisme vil derimod også have coherent memory accesses
3. Soacs med mapped operator kan give loads/store

```
entry mapped_plus [m][n] (X: [m][n]f32) : [n]f32 =
  let a = replicate n 0
  in reduce (map2 (f32.min)) a X
```

Kombinering af MC flattening og vektoriserede algoritmer

Er vores algoritmer en god udvidelse af MC flattening?

1. Sekventialiserede SOACs giver ofte *scatter* og *gather* operationer
2. Nested parallelisme vil derimod også have coherent memory accesses
3. Soacs med mapped operator kan give loads/store

```
uniform int acc_mem[m] = //some memory
for(uniform int red_i = 0; scan_i < i; sca_i++) {
  foreach(nest_i = 0 ... m) {
    x_acc = acc_mem[nest_i];
    x = mem[red_i * n + nest_i]; //vector load
    int res = x_acc + x;
    acc_mem[nest_i] = res;
  }
}
```

Benchmark		Multicore (ms)	Ours (ms)	Speedup
$m = 100000, n = 1000$		3.6	1.1	x3.23
$m = 1000, n = 100000$		84	81.2	x1.03

Konklusion

1. Vi har på en let måde fået ekstra speedup på multicore bagenden uden at ændre tidligere stadier af compileren

Konklusion

1. Vi har på en let måde fået ekstra speedup på multicore bagenden uden at ændre tidligere stadier af compileren
2. Identificeret steder hvor vi kan lave forbedringer til compileren, for at gøre memory accesses hurtigere