Intel© Implicit SPMD Program Compiler

Acknowledge

Material from Parallel Computing lectures from Prof. Kayvon and Prof. Olukotun
Stanford University

What are we going to do?

1. Introduction to ISPC
2. How to improve single-core performance using PRAM
3. How to use all the available cores in the CPU

Introduction

The ISPC approach

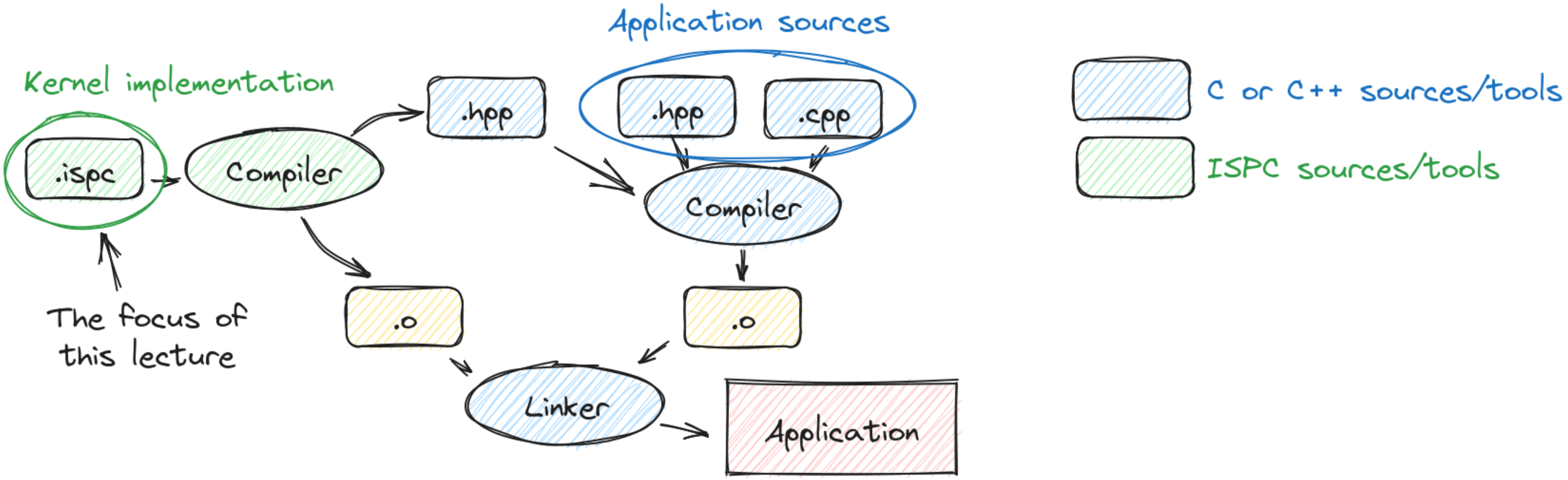
- C-based language for parallel computation
  - **SPMD** to express parallelism across SIMD lanes
  - **task**s to express parallelism across cores
- A great read:
  - “The Story of ISPC” (by Matt Pharr)
    https://pharr.org/matt/blog/2018/04/30/ispc-all.html

The ISPC compiler

Based on LLVM, freely available and documented

- Pre-compiled binaries: https://ispc.github.io/ispc.html
- User guide: https://ispc.github.io/ispc.html

Compilation process



Improve single-core performance

(attenzione stiamo considerando la x come un vettore -> insieme di elementi)

sin(x) example

Use the Taylor expansion up to N terms:

$$sin(x) = \sum_{n=0}^{N-1} \frac{-1^n}{(2n+1)!} \cdot x^{2n+1}$$

Example for *N* equal to 3:

$$sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120}$$

Cpp implementation

```cpp
void sinx(const float *vin, float *vout, const int size, const int terms) {
  for (int i = 0; i < size; i++) {
    // the approximated sinx value for terms == 1
    float sinx = vin[i]; // x^(2*0 + 1)/(1!)

    // initialize usefull values for computing higher terms
    float numerator = vin[i] * vin[i] * vin[i]; // x^(2*1 + 1)
    int denominator = 6;                        // (2*1 + 1)!
    int sign = -1;

    // loop over the higer terms
    for (int j = 1; j <= terms; j++) {
      sinx += sign * numerator / denominator; // update the sinx value

      // update the values for the higher term
      numerator *= vin[i] * vin[i];
      denominator *= (2 * j + 2) * (2 * j + 3);
      sign *= -1;
    }
    vout[i] = sinx;
  }
}
```

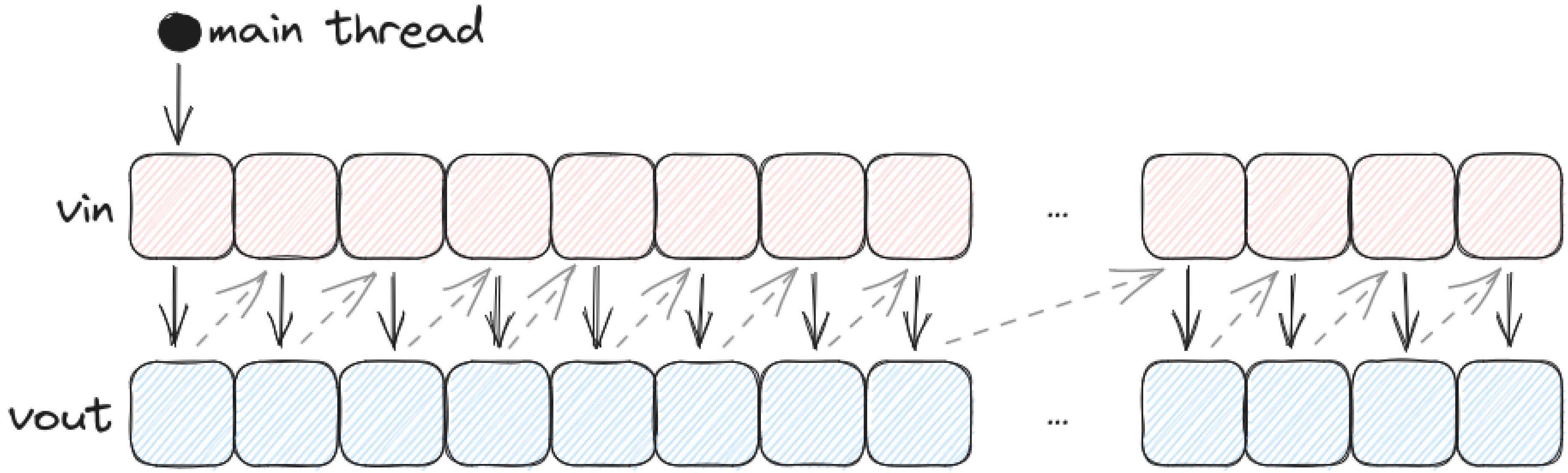
Cpp implementation (simplified)

```cpp
void sinx(const float *vin, float *vout, const int size, const int terms) {
  for (int i = 0; i < size; i++) {
    // the approximated sinx value for terms == 1
    float sinx = vin[i]; // taylor first term

    // increase accuracy

    // assign the final value
    vout[i] = sinx;
  }
}
```

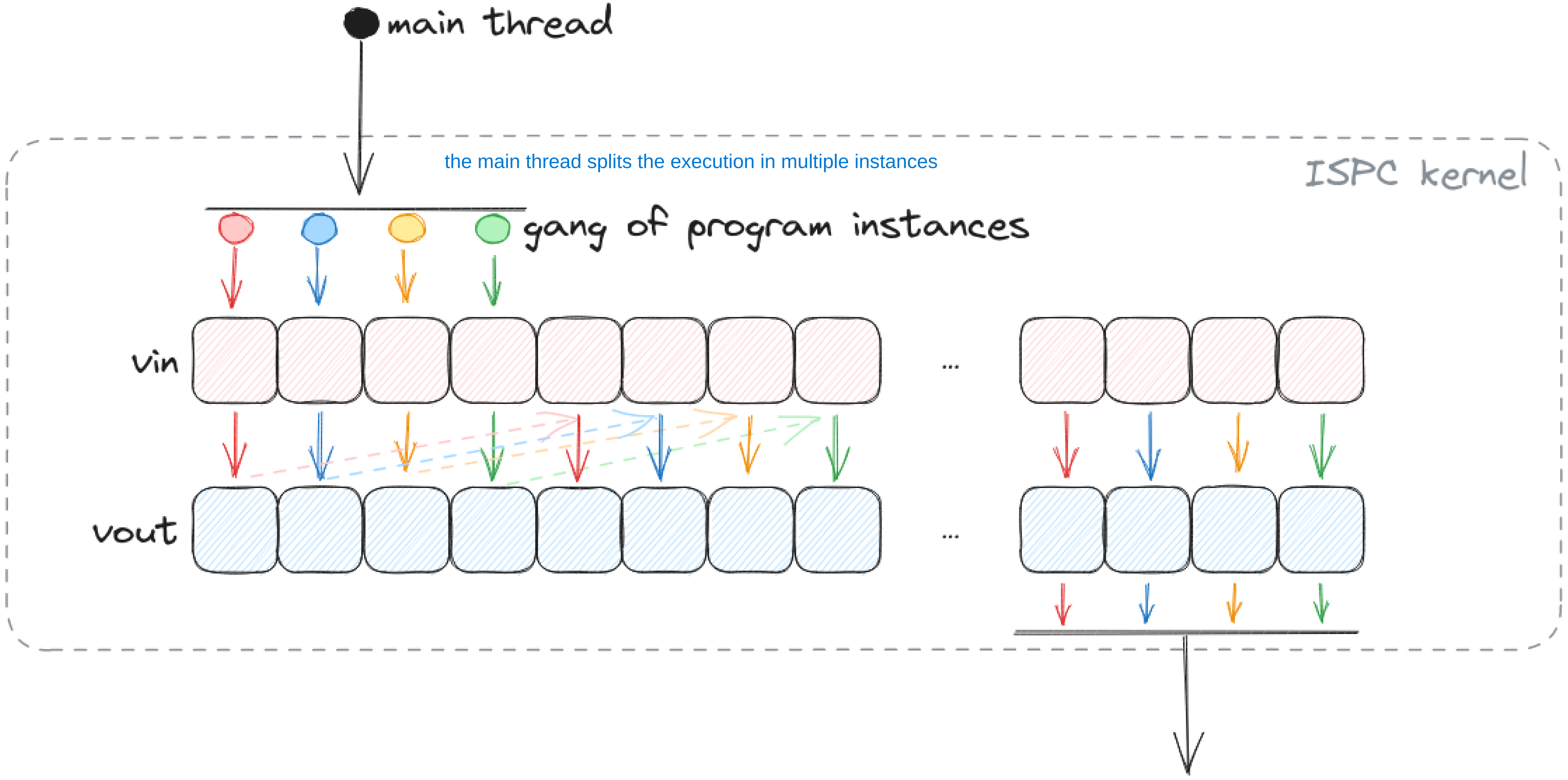
Execution Pattern



ISPC computation model

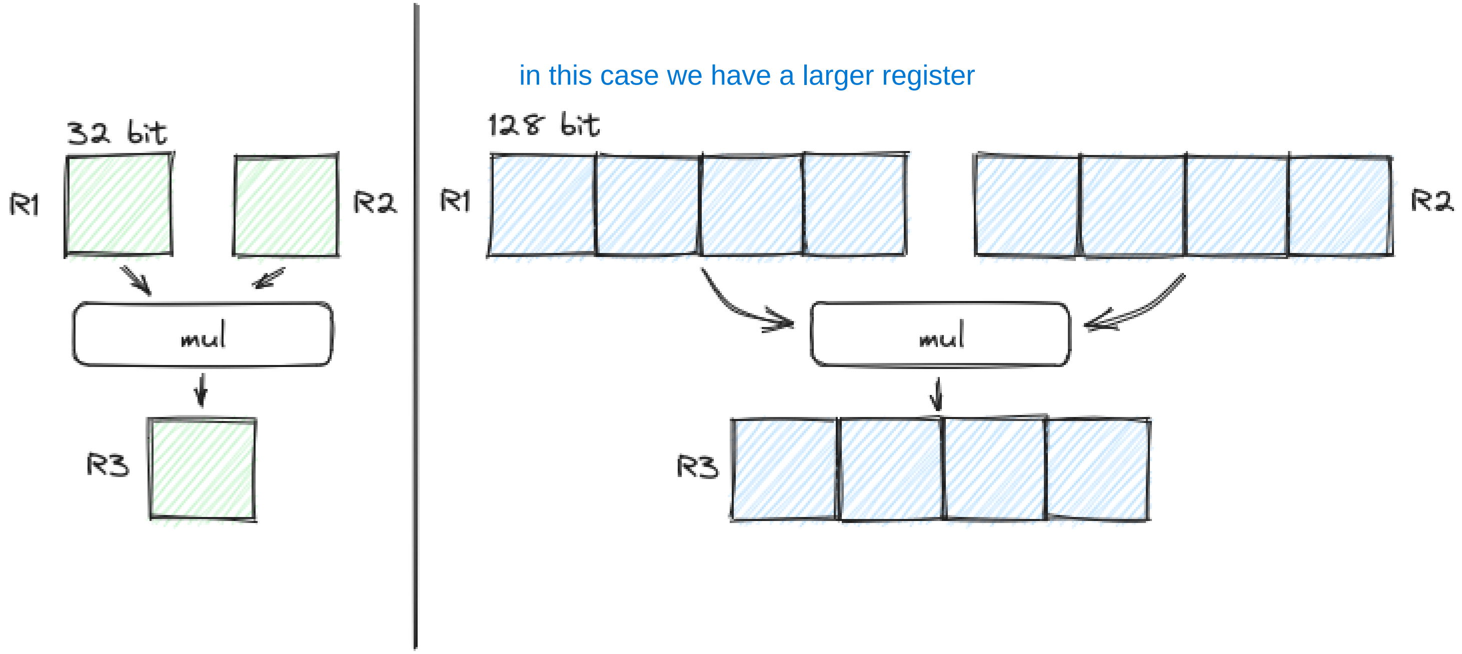
- The application code is executed by one processor, as usual
- The function(s) implemented in ISPC are executed by a **gang** of program instances

ISPC computation model (example)



SIMD hardware components

Apply the same instruction, e.g. mul, on more data*



*Condition applies, such as memory locality, access pattern, and alignment

The ISPC language

- C-based language
- Each local variable has one of these qualifiers:
  - **uniform** if the variable is "shared" across the gang
  - **varying** if the variable is "private" for each process (default)
- The ISPC language introduces ad-hoc statements to drive the parallelization

The *foreach* statement

Specify a loop over a possibly multi-dimensional domain of integer ranges

```
foreach(identifier = start ... end) { /* body */ }
```

the foreach -> splits the for across all the instances

> NOTE: the identifier assume values in [start, end)

Example:

```
forach(i = 0 ... 2) { vout[i] = i; }
```

> vout[0] == 0
> vout[1] == 1
> vout[2] == ?

ISPC implementation

```
export void sinx(
  const uniform float vin[],
  uniform float vout[],
  const uniform int size,
  const uniform int terms) {
  foreach (i = 0 ... size) {
    float value = vin[i];
    float numerator = vin[i] * vin[i] * vin[i];
    uniform int denominator = 6; // 3!
    uniform int sign = -1;
    for (uniform int j=1; j<=terms; j++) {
      value += sign * numerator / denominator;
      numerator *= vin[i] * vin[i];
      denominator *= (2*j+2) * (2*j+3);
      sign *= -1;
    }
    vout[i] = value;
  }
}
```

the var numerator is different for each program instance

Performance comparison

```
local $ ./main
[ function ispc::sinx took 12128971ns ]
[ function ispc::sinx took 5879860ns ]
[ function ispc::sinx took 5598166ns ]
[ function ispc::sinx took 5544776ns ]
[ function cpp::sinx took 41197113ns ]
[ function cpp::sinx took 33893258ns ]
[ function cpp::sinx took 32915954ns ]
[ function cpp::sinx took 32530714ns ]
```

We need to help the Cpp compiler

```cpp
void sinx(const float *vin, float *vout, const int size, const int terms) {
  // derive the boundaries of the loop unrolling
  const int leftovers = size % 8;

  // process the leftovers using the normal operations
  for (int i = 0; i < leftovers; ++i) {
    // the approximated sinx value for terms == 1
    float sinx = vin[i]; // x^(2*0 + 1)/(1!)

    // initialize usefull values for computing higher terms
    float numerator = vin[i] * vin[i] * vin[i]; // x^(2*1 + 1)
    int denominator = 6;                        // (2*1 + 1)!
    int sign = -1;

    // loop over the higer terms
    for (int j = 1; j <= terms; j++) {
      sinx += sign * numerator / denominator; // update the sinx value

      // update the values for the higher term
      numerator *= vin[i] * vin[i];
      denominator *= (2 * j + 2) * (2 * j + 3);
      sign *= -1;
    }
    vout[i] = sinx;
  }

  // we assume that size is a multiple of 8 for simplicity
  for (int i = leftovers; i < size; i += 8) {
    // the approximated sinx value for terms == 1
    float sinx[8] = {vin[i],     vin[i + 1], vin[i + 2], vin[i + 3],
                 vin[i + 4], vin[i + 5], vin[i + 6], vin[i + 7]};

    // initialize usefull values for computing higher terms
    float numerator[8] = {vin[i + 0] * vin[i + 0] * vin[i + 0],
                          vin[i + 1] * vin[i + 1] * vin[i + 1],
                          vin[i + 2] * vin[i + 2] * vin[i + 2],
                          vin[i + 3] * vin[i + 3] * vin[i + 3],
                          vin[i + 4] * vin[i + 4] * vin[i + 4],
                          vin[i + 5] * vin[i + 5] * vin[i + 5],
                          vin[i + 6] * vin[i + 6] * vin[i + 6],
                          vin[i + 7] * vin[i + 7] * vin[i + 7]};
    int denominator = 6;
    int sign = -1;

    // loop over the higer terms
    for (int j = 1; j <= terms; j++) {
      sinx[0] += sign * numerator[0] / denominator;
      sinx[1] += sign * numerator[1] / denominator;
      sinx[2] += sign * numerator[2] / denominator;
      sinx[3] += sign * numerator[3] / denominator;
      sinx[4] += sign * numerator[4] / denominator;
      sinx[5] += sign * numerator[5] / denominator;
      sinx[6] += sign * numerator[6] / denominator;
      sinx[7] += sign * numerator[7] / denominator;

      // update the values for the higher term
      numerator[0] = vin[i] * vin[i];
      numerator[1] = vin[i + 1] * vin[i + 1];
      numerator[2] = vin[i + 2] * vin[i + 2];
      numerator[3] = vin[i + 3] * vin[i + 3];
      numerator[4] = vin[i + 4] * vin[i + 4];
      numerator[5] = vin[i + 5] * vin[i + 5];
      numerator[6] = vin[i + 6] * vin[i + 6];
      numerator[7] = vin[i + 7] * vin[i + 7];
      denominator *= (2 * j + 2) * (2 * j + 3);
      sign *= -1;
    }
    vout[i + 0] = sinx[0];
    vout[i + 1] = sinx[1];
    vout[i + 2] = sinx[2];
    vout[i + 3] = sinx[3];
    vout[i + 4] = sinx[4];
    vout[i + 5] = sinx[5];
    vout[i + 6] = sinx[6];
    vout[i + 7] = sinx[7];
  }
}
```

we're applying loop unrolling

Performance comparison

```
local $ ./main
[ function ispc::sinx took 11370797ns ]
[ function ispc::sinx took 5561399ns ]
[ function ispc::sinx took 5665879ns ]
[ function ispc::sinx took 5310879ns ]
[ function cpp::sinx took 10093868ns ]
[ function cpp::sinx took 4915559ns ]
[ function cpp::sinx took 4724109ns ]
[ function cpp::sinx took 4893244ns ]
```

Program instances communications

Overview

The ISPC language exposes two sets of functions:

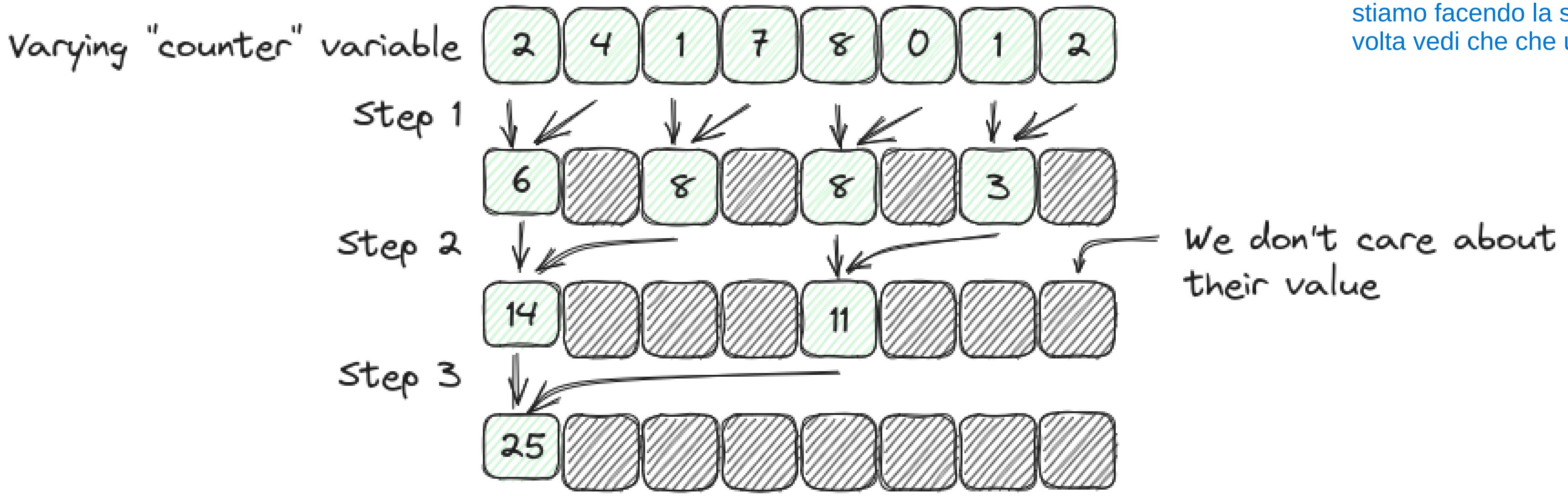
1. **Cross-Program Instance Operations** - low level communication facilities
   Eg. *broadcast*, *rotate*, *shuffle*
2. **Reductions** - high level communication facilities:
   Eg. *any*, *reduce_add*, *reduce_max*

Please refer to the documentation for the complete list

Vector counting example

Count how many elements of a given vector are above a given threshold

Example using low level interface



stiamo facendo la somma degli elementi utilizzando la reduction. Ogni volta vedi che che usiamo sempre meno celle dell'array

How to address an ISPC program instance

- The compiler implicitly defines two variables:
    - *programIndex*, to identify each instance in a gang
    - *programCount*, the gang size

How to exchange values

We hinge on the *shuffle* function

```
int32 shuffle(int32 value, int permutation)
```

> We pass as input parameter our *value* and the *programIndex* of the instance that we want to get its value, which is the return parameter.
>
> > **NOTE**: all the variables are *varying*

How to pick a value from an ISPC process

We hinge on the *extract* function

```
uniform int32 extract(int32 value, uniform int index)
```

> We pass as input parameter our *value* and the *programIndex* of the instance that we want to get its value, which is the return parameter.
>
> > **NOTE**: the index and return parameter are *uniform*

Possible implementation

```
export uniform int count_greater_than_shuffle(
  const uniform float vin[],
  const uniform int size,
  const uniform int threshold) {

  // count how many number are above the threshold
  int counter = 0;
  foreach (i = 0 ... size) {
    if (vin[i] > threshold) {
      ++counter; // NOTE: some programs do not execute this statement
    }
  }

  // reduce the counter toward the pogram with index 0
  uniform int sender_stride = 1;
  const uniform int half_programCount = programCount / 2;
  while(sender_stride <= half_programCount) {
    const int sender_index = programIndex + sender_stride;
    if (sender_index < programCount) {
      const int neighbor_value = shuffle(counter,  sender_index);
      counter += neighbor_value;
    }
    sender_stride *= 2;
  }
  if (half_programCount*2 < programCount) { // for odd gangs
    counter += shuffle(counter, programCount - 1);
  }

  // return the value of the program with rank 0
  // NOTE: it's the only with the correct value
  return extract(counter, 0);
}
```

Using high-level reduction

```
export uniform int count_greater_than_reduce(
  const uniform float vin[],
  const uniform int size,
  const uniform int threshold) {

  // count how many number are above the threshold
  int counter = 0;
  foreach (i = 0 ... size) {
    if (vin[i] > threshold) {
      ++counter; // NOTE: some programs do not execute this statement
    }
  }

  // perform the reduction
  return reduce_add(counter);
}
```

Performance comparison

```
local $ ./main
[ function ispc::count_greater_than_shuffle took 581053ns ]
[ function ispc::count_greater_than_shuffle took 637846ns ]
[ function ispc::count_greater_than_shuffle took 689682ns ]
[ function ispc::count_greater_than_shuffle took 569139ns ]
[ function ispc::count_greater_than_reduce took 649901ns ]
[ function ispc::count_greater_than_reduce took 554918ns ]
[ function ispc::count_greater_than_reduce took 636612ns ]
[ function ispc::count_greater_than_reduce took 557713ns ]
```

Takeaway messages

- The ISPC compiler can use PRAM to improve performance
- Use the reductions when available,
- use the low-level interface otherwise

Scale across the available cores

The ISPC tasking model

- A task is a program that executes asynchronously
  - each task is executed by a gang of program instances
- The ISPC language uses two keywords to manage the task lifetime:
  - *launch[<n>]* to spawn the execution of *n* tasks
  - *sync* to wait until all the tasks terminate

> **NOTE**: the compiler automatically add a *sync* instruction before a return

The task function

- Any function that has the *task* prefix and that returns void

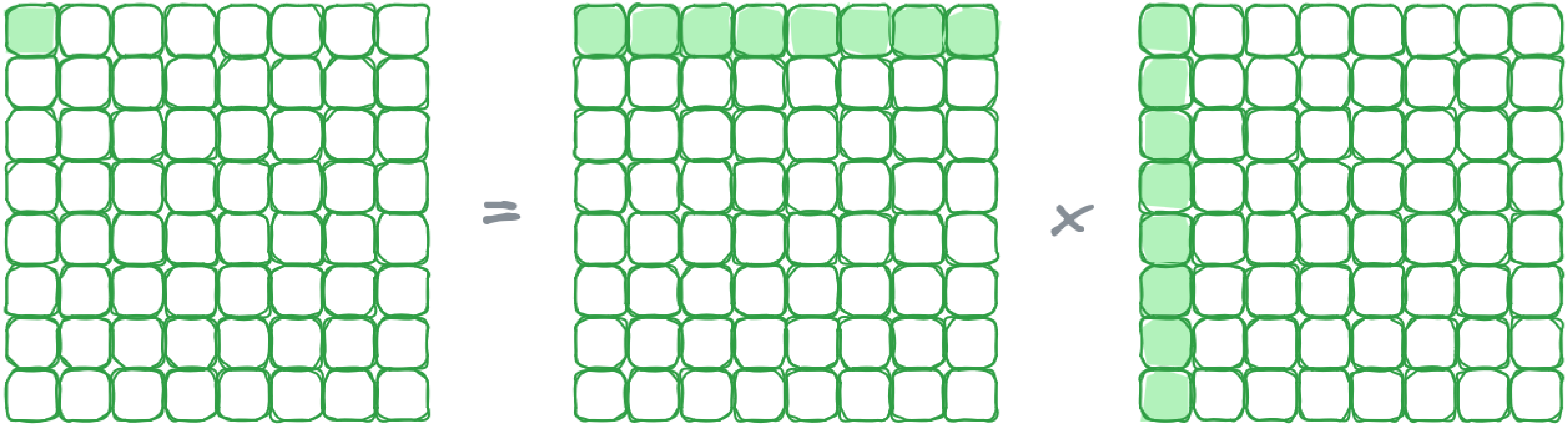
```
task void my_task_func( /* params */ ) { /* body */ }
```

- We can use two built-in variables in the function body: *taskIndex* and *taskCount*
- The mapping between system threads and tasks is left to the user

> We use the one shipped in *<repo>/examples/common/tasksys.cpp*

Matrix multiplication example

i need 3 loops for performing it.



Serial implementation

```
export void matmul_serial(
  uniform float C[],
  const uniform float A[],
  const uniform float B[],
  const uniform int size) {
  foreach (row_index = 0 ... size, column_index = 0 ... size) {   primo e secondo loop per le righe e colonne
    float result = 0.0f;
    for(uniform int k = 0; k < size; ++k) {   terzo loop per ottenere il risultato finale.
      result += A[row_index*size + k] * B[k*size + row_index];
    }
    C[row_index*size + column_index] = result;
  }
}
```

Parallel implementation

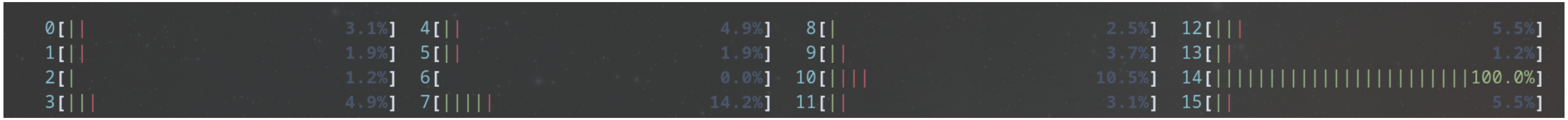
```
task void matmul(
  uniform float C[],
  const uniform float A[],
  const uniform float B[],
  const uniform int size) {
  const uniform int row_index = taskIndex;
  foreach (column_index = 0 ... size) {
    float result = 0.0f;
    for(uniform int k = 0; k < size; ++k) {
      result += A[row_index*size + k] * B[k*size + row_index];
    }
    C[row_index*size + column_index] = result;
  }
}

export void matmul_task(
  uniform float C[],
  const uniform float A[],
  const uniform float B[],
  const uniform int size) {
  launch[size] matmul(C, A, B, size);
}
```

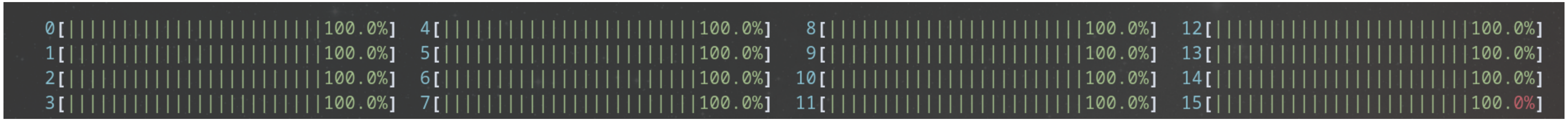
sto lanciando size tasks che vanno ad eseguire matmul

Resource usage comparison

Serial implementation



Parallel implementation



Takeaway messages

- Use tasks to scale across different cores
- The mapping between system threads and tasks is flexible
- However, consider the option to use external approaches to handle the cross-core parallelism, such as OpenMP, Intel TBB, or pthread