Safe & Productive Performance with User-Schedulable Languages

Jonathan Ragan-Kelley
MIT EECS / CSAIL

High-performance programming
requires low-level control

High-performance programming
requires low-level control

Therefore it must be
unsafe & unproductive

High-performance programming **requires low-level control**

Therefore it must be **unsafe & unproductive**

High-performance programming **requires low-level control**

~~Therefore it must be **unsafe & unproductive**~~

Programming systems:
safety & productivity
through **abstraction**

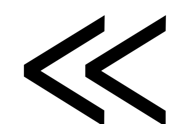
Programming systems:
safety & productivity
through **abstraction**

Programming systems: safety & productivity through **abstraction**

```
SELECT
    d.name,
    AVG(e.salary) as avg_salary
FROM employees e
JOIN departments d ON e.department = d.id
WHERE e.hire_date > '2020-01-01'
GROUP BY d.id, d.name
HAVING COUNT(e.id) >= 5
ORDER BY avg_salary DESC
LIMIT 5;
```

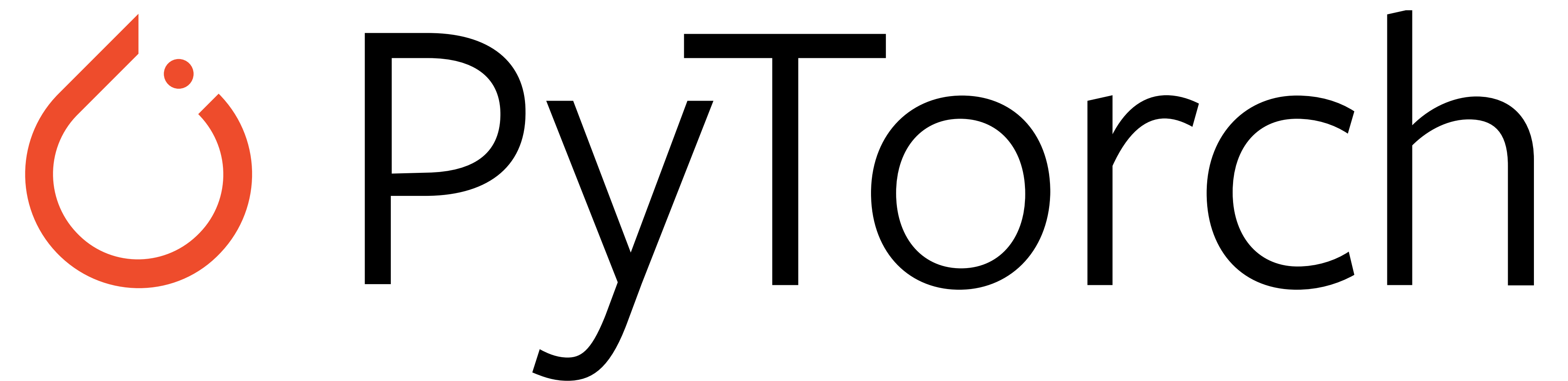
Programming systems: safety & productivity through **abstraction**

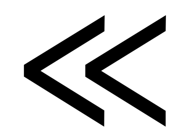
```sql
SELECT
    d.name,
    AVG(e.salary) as avg_salary
FROM employees e
JOIN departments d ON e.department = d.id
WHERE e.hire_date > '2020-01-01'
GROUP BY d.id, d.name
HAVING COUNT(e.id) >= 5
ORDER BY avg_salary DESC
LIMIT 5;
```



```cpp
    double total_salary = 0.0;
    int employee_count = 0;

    double avg_salary() const {
        return employee_count > 0 ? total_salary / employee_count : 0.0;
    }
};

std::vector<std::pair<std::string, double>> topDepartmentsByAvgSalary(
    const std::vector<Employee>& employees,
    const std::vector<Department>& departments,
    const std::chrono::system_clock::time_point& cutoff_date,
    int min_employees,
    int limit
) {
    // Build index for departments
    std::unordered_map<int, std::string> dept_index;
    for (const auto& dept : departments) {
        dept_index[dept.department_id] = dept.department_name;
    }

    // Filter, join, and aggregate
    std::unordered_map<int, DepartmentStats> stats;
    for (const auto& emp : employees) {
        if (emp.hire_date > cutoff_date) {
            auto it = dept_index.find(emp.department_id);
            if (it != dept_index.end()) {
                // zero-init if missing
                auto& dept_stat = stats[emp.department_id];
                dept_stat.department_name = it->second;
                dept_stat.total_salary += emp.salary;
                dept_stat.employee_count++;
            }
        }
    }

    // Filter aggregated results
    std::vector<std::pair<std::string, double>> result;
    for (const auto& [dept_id, dept_stat] : stats) {
        if (dept_stat.employee_count >= min_employees) {
            result.push_back({dept_stat.department_name, dept_stat.avg_salary()});
        }
    }

    // Sort by average salary
    std::sort(result.begin(), result.end(),
              [](const auto& a, const auto& b) { return a.second > b.second; });

    // Apply limit
    if (result.size() > limit) {
        result.resize(limit);
    }

    return result;
}
```

Programming systems:
safety & productivity
through **abstraction**





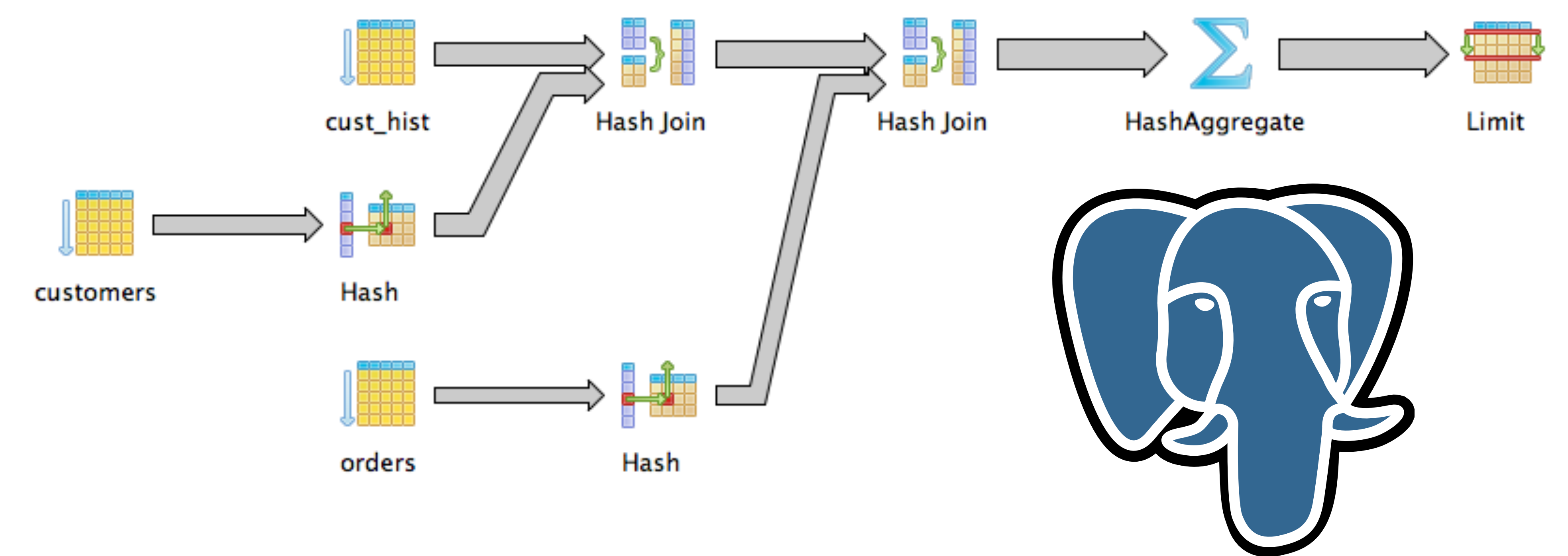
Programming systems:
safety & productivity
through **abstraction**

productive **performance**
through **automation**

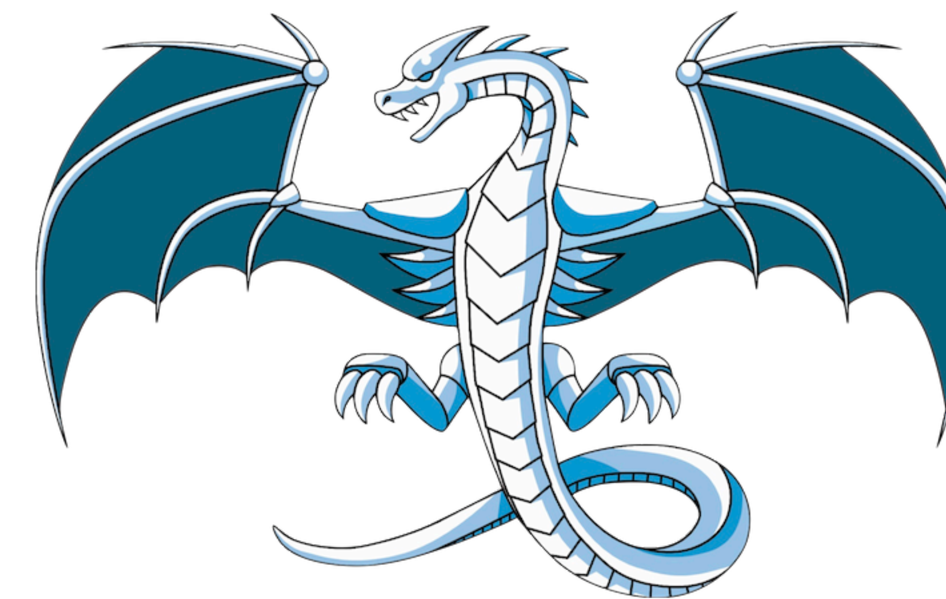
Programming systems:
safety & productivity
through **abstraction**

productive **performance**
through **automation**

Programming systems:
safety & productivity
through **abstraction**



productive **performance**
through **automation**



Programming systems:
safety & productivity
through **abstraction**

productive **performance**
through **automation**

Programming systems:
safety & productivity
through **abstraction**

productive **performance**
through **automation**

Performance engineering:
peak performance
through low-level
control

Performance engineering:
peak performance through low-level
control

vectorization

Performance engineering:
peak performance through low-level
control

vectorization
cache locality

Performance engineering:
peak performance through low-level **control**

vectorization

cache locality

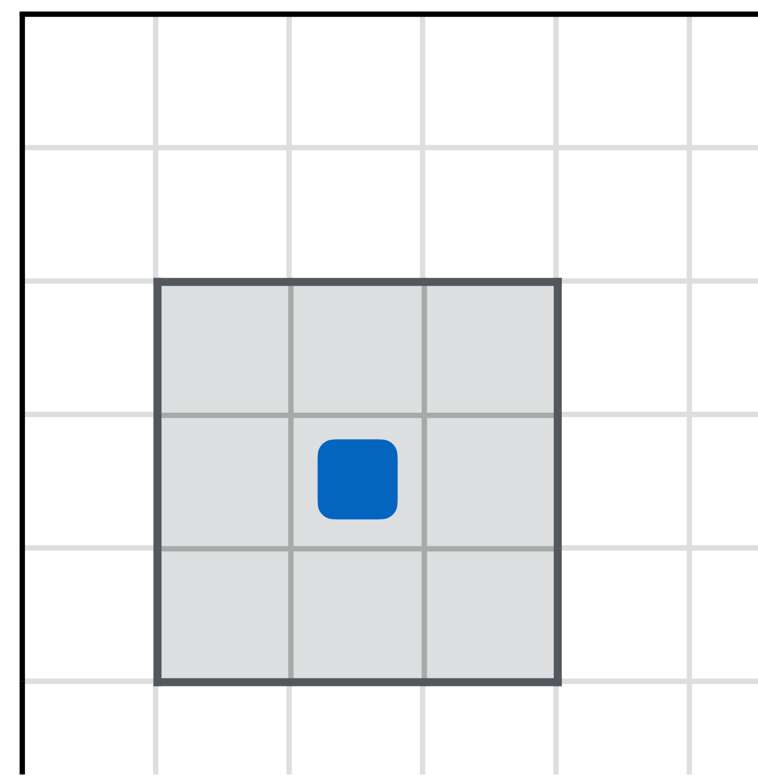
parallelism & synchronization

…

Performance engineering: peak performance through low-level **control**

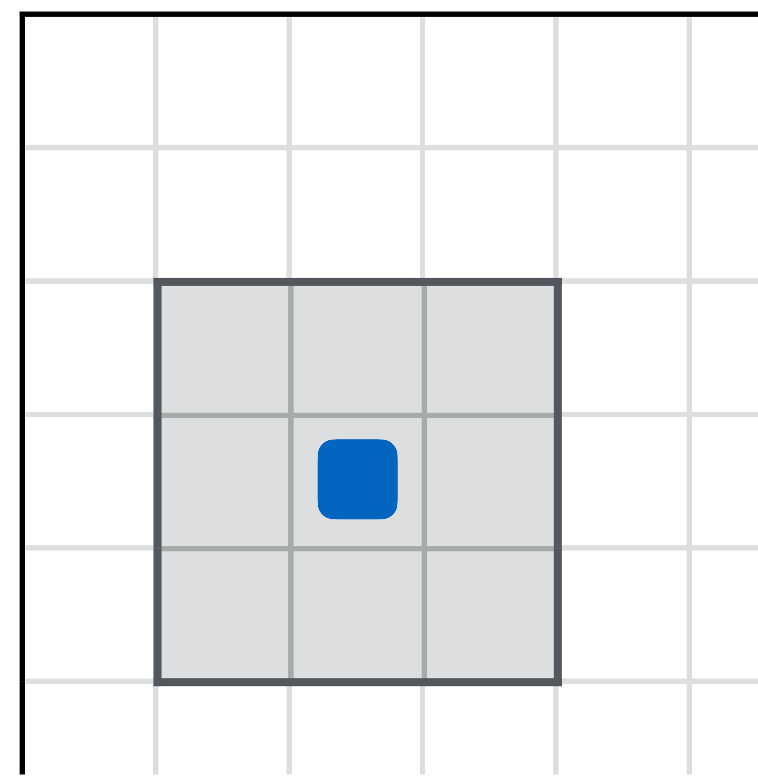
Example: blurring an image

Example: blurring an image



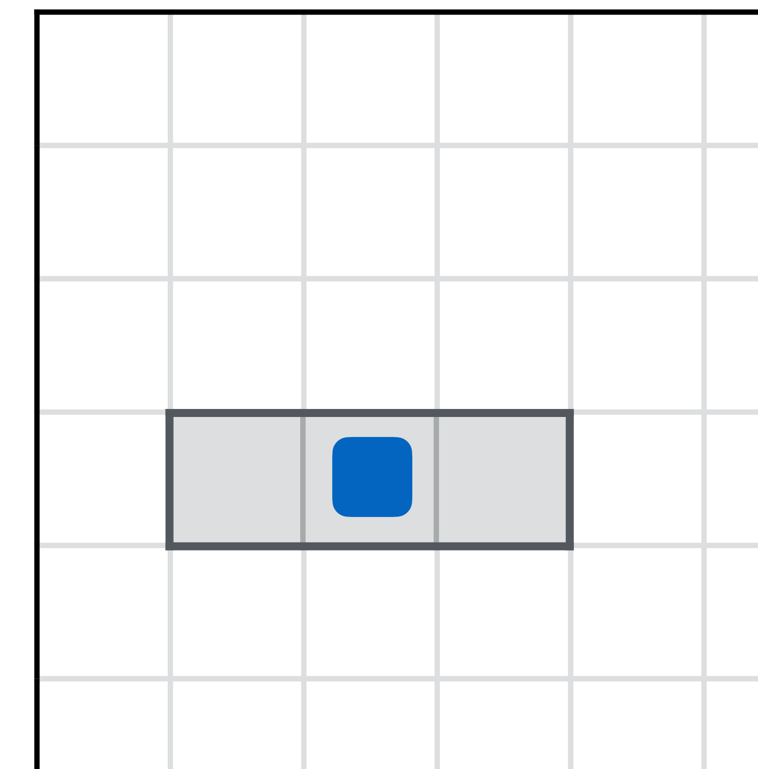
3×3
box filter

Example: blurring an image



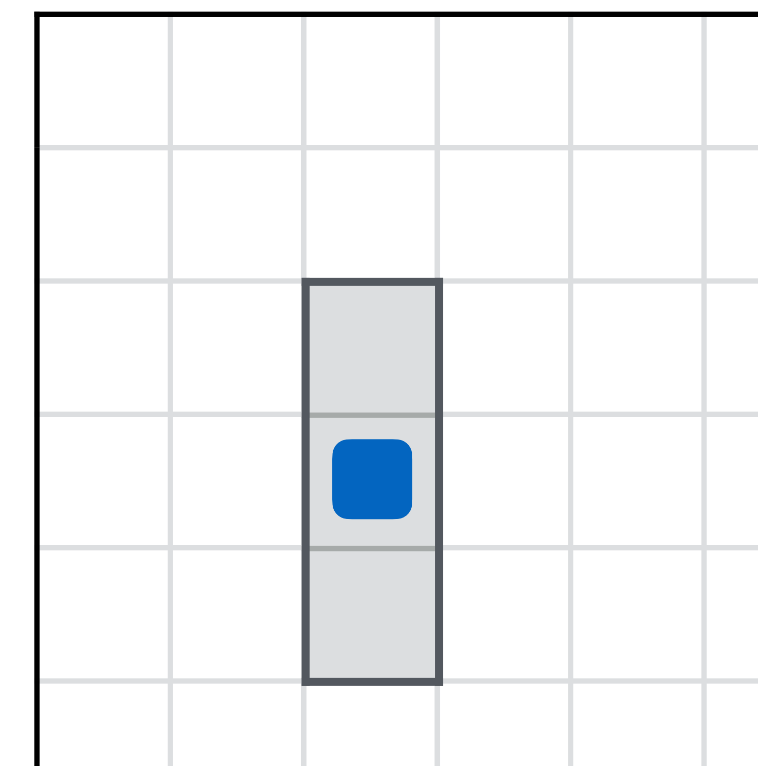
3×3
box filter

=



3×1
box filter

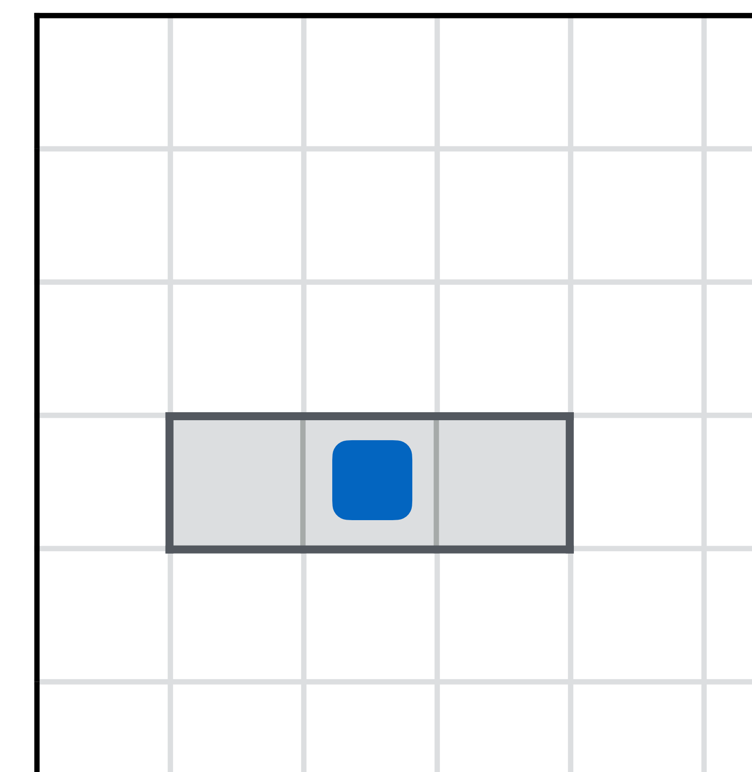
∘



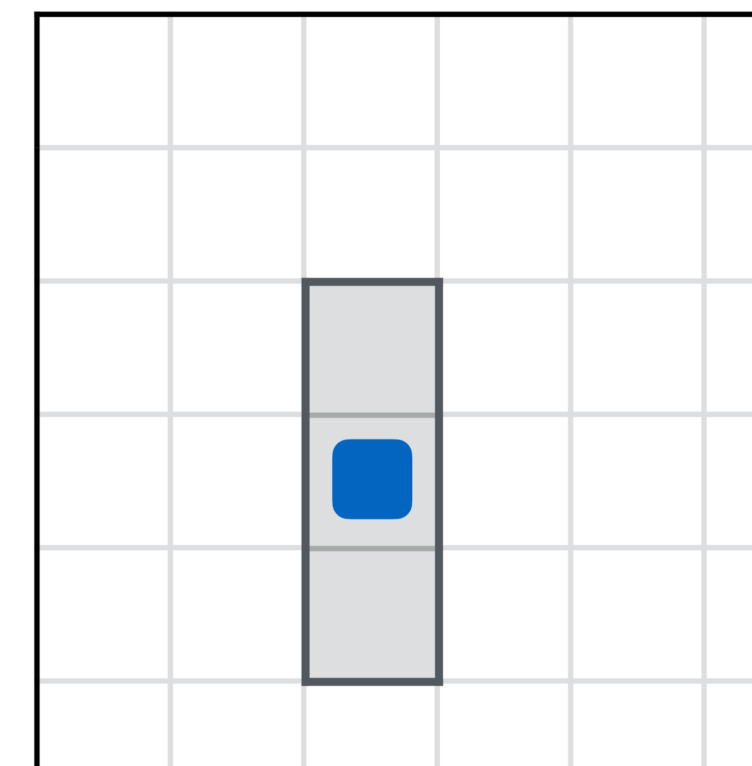
1×3
box filter

Example: blurring an image

```
void blur(int W, int H, const u16_t *in, u16_t *out) {
  u16_t *horiz = new u16_t[W*H];

  for (int y = 0; y < H; y++) {
    for (int x = 1; x < W-1; x++) {
      horiz[x + y*W] = (
        in[x-1 + y*W] + in[x + y*W] + in[x+1 + y*W]
      ) / 3;
  }}

  for (int y = 1; y < H-1; y++) {
    for (int x = 1; x < W-1; x++) {
      out[x + y*W] = ( horiz[x + (y-1)*W]
                     + horiz[x +  y   *W]
                     + horiz[x + (y+1)*W] ) / 3;
  }}

  delete[] horiz;
}
```

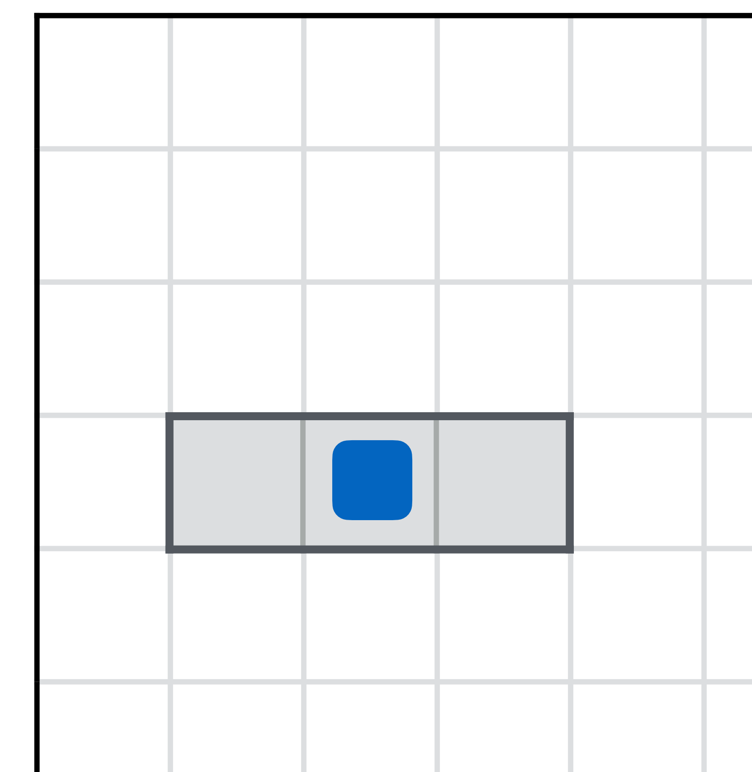


∘



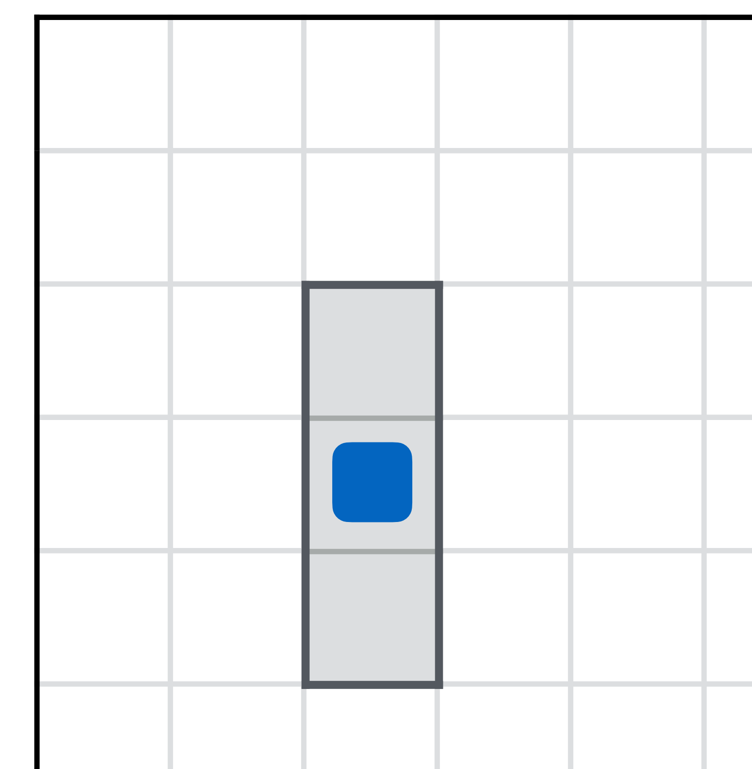
Example: blurring an image

```
void blur(int W, int H, const u16_t *in, u16_t *out) {
  u16_t *horiz = new u16_t[W*H];

  for (int y = 0; y < H; y++) {
    for (int x = 1; x < W-1; x++) {
      horiz[x + y*W] = (
        in[x-1 + y*W] + in[x + y*W] + in[x+1 + y*W]
      ) / 3;
  }}

  for (int y = 1; y < H-1; y++) {
    for (int x = 1; x < W-1; x++) {
      out[x + y*W] = ( horiz[x + (y-1)*W]
                     + horiz[x +  y   *W]
                     + horiz[x + (y+1)*W] ) / 3;
  }}

  delete[] horiz;
}
```



3×1
box filter

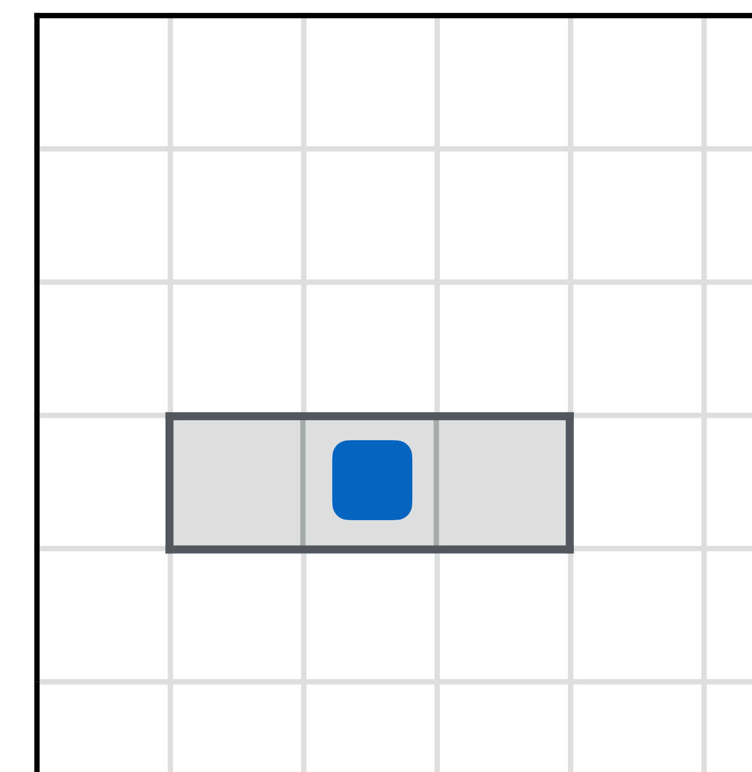
∘



1×3
box filter

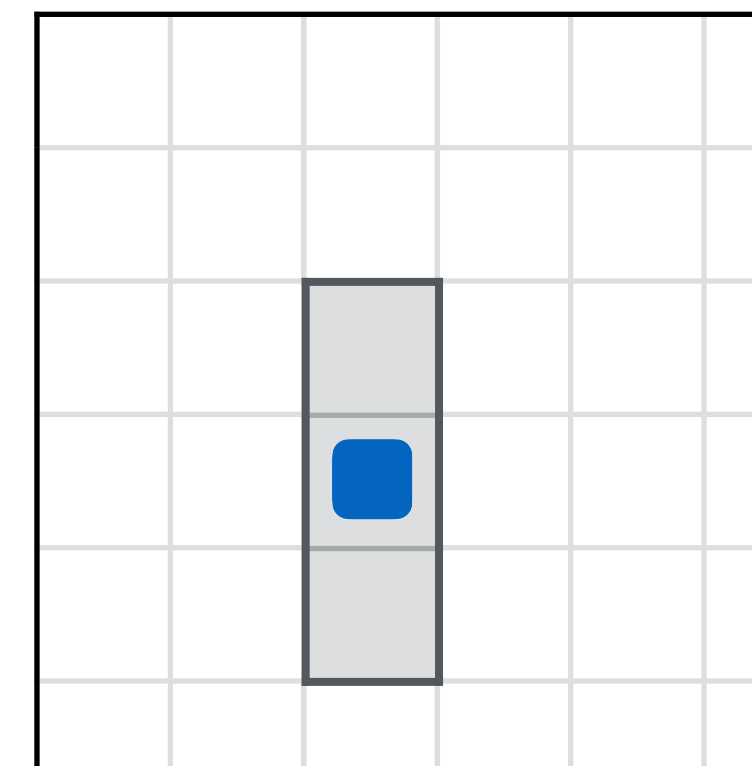
Example: blurring an image

```
void blur(int W, int H, const u16_t *in, u16_t *out) {
  u16_t *horiz = new u16_t[W*H];

  for (int y = 0; y < H; y++) {
    for (int x = 1; x < W-1; x++) {
      horiz[x + y*W] = (
        in[x-1 + y*W] + in[x + y*W] + in[x+1 + y*W]
      ) / 3;
  }}

  for (int y = 1; y < H-1; y++) {
    for (int x = 1; x < W-1; x++) {
      out[x + y*W] = ( horiz[x + (y-1)*W]
                     + horiz[x +  y   *W]
                     + horiz[x + (y+1)*W] ) / 3;
  }}

  delete[] horiz;
}
```



3×1
box filter

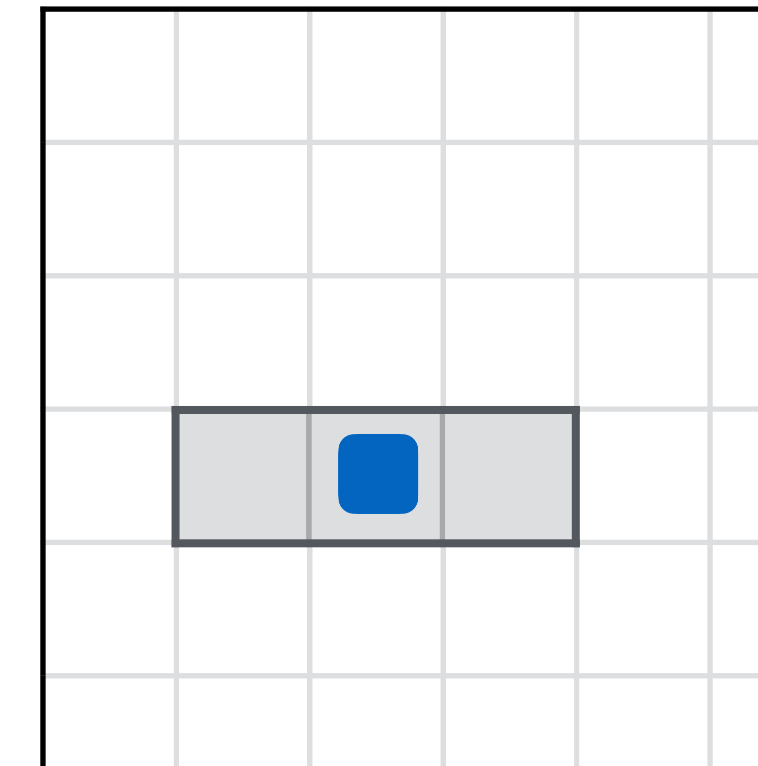
∘



1×3
box filter

Example: blurring an image

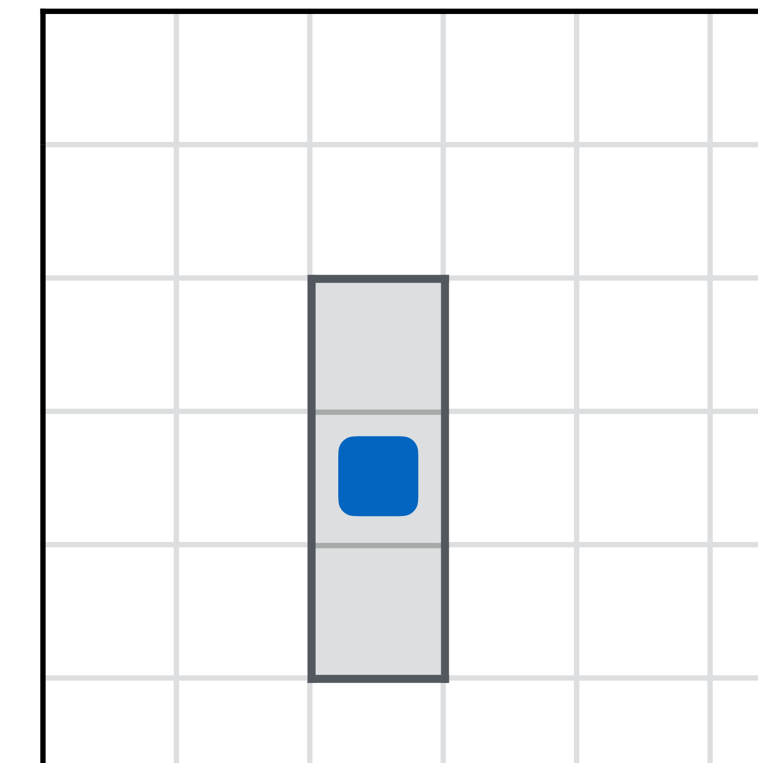
```
horiz = (in_[:, :-2] + in_[:, 1:-1] + in_[:, 2:]) / 3
```



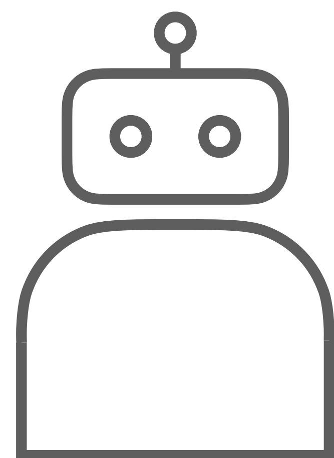
3×1
box filter

∘

```
out = (horiz[:-2, :] + horiz[1:-1, :] + horiz[2:, :]) / 3
```



1×3
box filter

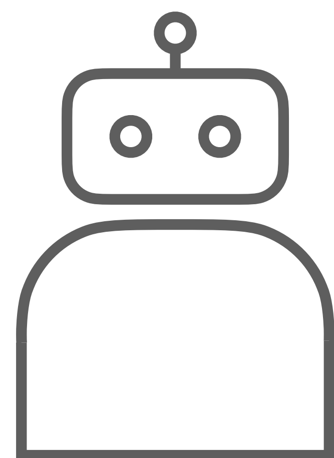


Automatically-delivered performance

```
void blur(int W, int H, const u16_t *in, u16_t *out) {
  u16_t *horiz = new u16_t[W*H];

  for (int y = 0; y < H; y++) {
    for (int x = 1; x < W-1; x++) {
      horiz[x + y*W] = (
        in[x-1 + y*W] + in[x + y*W] + in[x+1 + y*W]
      ) / 3;
  }}

  for (int y = 1; y < H-1; y++) {
    for (int x = 1; x < W-1; x++) {
      out[x + y*W] = ( horiz[x + (y-1)*W]
                     + horiz[x +  y   *W]
                     + horiz[x + (y+1)*W] ) / 3;
  }}

  delete[] horiz;
}
```

≈

```
horiz = (in_[:, :-2] + in_[:, 1:-1] + in_[:, 2:]) / 3
out = (horiz[:-2, :] + horiz[1:-1, :] + horiz[2:, :]) / 3
```



Automatically-delivered performance

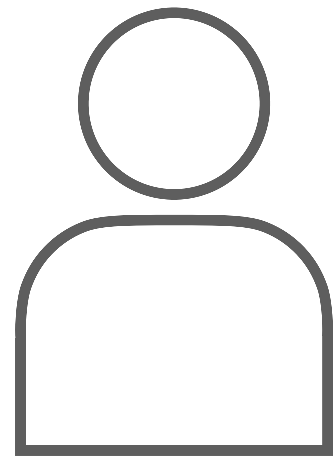
```
void blur(int W, int H, const u16_t *in, u16_t *out) {
  u16_t *horiz = new u16_t[W*H];

  for (int y = 0; y < H; y++) {
    for (int x = 1; x < W-1; x++) {
      horiz[x + y*W] = (
        in[x-1 + y*W] + in[x + y*W] + in[x+1 + y*W]
      ) / 3;
  }}

  for (int y = 1; y < H-1; y++) {
    for (int x = 1; x < W-1; x++) {
      out[x + y*W] = ( horiz[x + (y-1)*W]
                     + horiz[x +  y   *W]
                     + horiz[x + (y+1)*W] ) / 3;
  }}

  delete[] horiz;
}
```

≈

```
horiz = (in_[:, :-2] + in_[:, 1:-1] + in_[:, 2:]) / 3
out = (horiz[:-2, :] + horiz[1:-1, :] + horiz[2:, :]) / 3
```

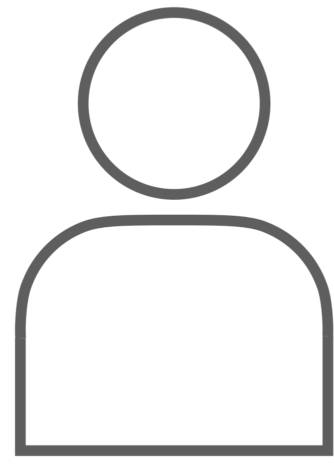
3 ms / megapixel

3 ms / megapixel



Hand-optimized performance

```
void blur_fast(int W, int H, const u16_t *in, u16_t *out) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < H-2; yTile += 32) {
    __m128i horiz[(128 / 8) * (32 + 2)];
    for (int xTile = 0; xTile < W-8; xTile += 128) {
      __m128i *horizPtr = horiz;
      for (int y = 0; y < 32 + 2; y++) {
        const size_t inIdx = xTile + (yTile+y)*W;
        const u16_t *inPtr = in+inIdx;
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128((const __m128i *)(inPtr));
          __m128i b = _mm_loadu_si128((const __m128i *)(inPtr + 1));
          __m128i c = _mm_loadu_si128((const __m128i *)(inPtr + 2));
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(horizPtr++, avg);
          inPtr += 8;
        }
      }
      horizPtr = horiz;
      for (int y = 0; y < 32; y++) {
        const size_t outIdx = xTile + (yTile+y)*(W-8);
        __m128i *outPtr = (__m128i *)(out+outIdx);
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128(horizPtr + (2 * 128) / 8);
          __m128i b = _mm_load_si128(horizPtr + 128 / 8);
          __m128i c = _mm_load_si128(horizPtr++);
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

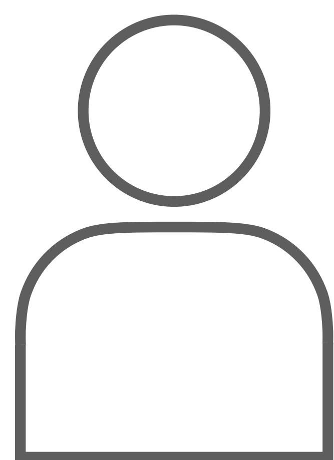


Hand-optimized performance

```
void blur_fast(int W, int H, const u16_t *in, u16_t *out) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < H-2; yTile += 32) {
    __m128i horiz[(128 / 8) * (32 + 2)];
    for (int xTile = 0; xTile < W-8; xTile += 128) {
      __m128i *horizPtr = horiz;
      for (int y = 0; y < 32 + 2; y++) {
        const size_t inIdx = xTile + (yTile+y)*W;
        const u16_t *inPtr = in+inIdx;
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128((const __m128i *)(inPtr));
          __m128i b = _mm_loadu_si128((const __m128i *)(inPtr + 1));
          __m128i c = _mm_loadu_si128((const __m128i *)(inPtr + 2));
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(horizPtr++, avg);
          inPtr += 8;
        }
      }
      horizPtr = horiz;
      for (int y = 0; y < 32; y++) {
        const size_t outIdx = xTile + (yTile+y)*(W-8);
        __m128i *outPtr = (__m128i *)(out+outIdx);
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128(horizPtr + (2 * 128) / 8);
          __m128i b = _mm_load_si128(horizPtr + 128 / 8);
          __m128i c = _mm_load_si128(horizPtr++);
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

0.2 ms / megapixel / core

15x faster, scalable across cores

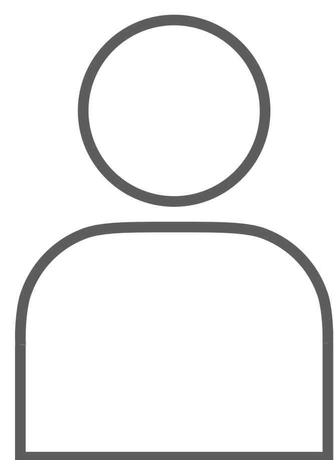


Hand-optimized performance

```
void blur_fast(int W, int H, const u16_t *in, u16_t *out) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < H-2; yTile += 32) {
    __m128i horiz[(128 / 8) * (32 + 2)];
    for (int xTile = 0; xTile < W-8; xTile += 128) {
      __m128i *horizPtr = horiz;
      for (int y = 0; y < 32 + 2; y++) {
        const size_t inIdx = xTile + (yTile+y)*W;
        const u16_t *inPtr = in+inIdx;
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128((const __m128i *)(inPtr));
          __m128i b = _mm_loadu_si128((const __m128i *)(inPtr + 1));
          __m128i c = _mm_loadu_si128((const __m128i *)(inPtr + 2));
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(horizPtr++, avg);
          inPtr += 8;
        }
      }
      horizPtr = horiz;
      for (int y = 0; y < 32; y++) {
        const size_t outIdx = xTile + (yTile+y)*(W-8);
        __m128i *outPtr = (__m128i *)(out+outIdx);
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128(horizPtr + (2 * 128) / 8);
          __m128i b = _mm_load_si128(horizPtr + 128 / 8);
          __m128i c = _mm_load_si128(horizPtr++);
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

0.2 ms / megapixel / core

15x faster, scalable across cores



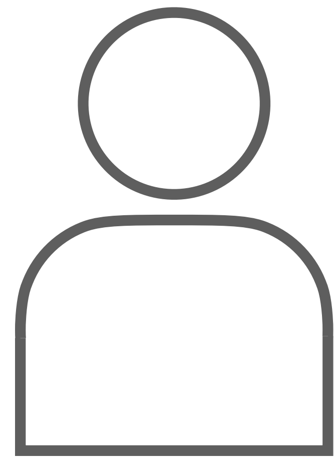
Hand-optimized performance

Tiled computation by 128×32

```
void blur_fast(int W, int H, const u16_t *in, u16_t *out) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < H-2; yTile += 32) {
    __m128i horiz[(128 / 8) * (32 + 2)];
    for (int xTile = 0; xTile < W-8; xTile += 128) {
      __m128i *horizPtr = horiz;
      for (int y = 0; y < 32 + 2; y++) {
        const size_t inIdx = xTile + (yTile+y)*W;
        const u16_t *inPtr = in+inIdx;
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128((const __m128i *)(inPtr));
          __m128i b = _mm_loadu_si128((const __m128i *)(inPtr + 1));
          __m128i c = _mm_loadu_si128((const __m128i *)(inPtr + 2));
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(horizPtr++, avg);
          inPtr += 8;
        }
      }
      horizPtr = horiz;
      for (int y = 0; y < 32; y++) {
        const size_t outIdx = xTile + (yTile+y)*(W-8);
        __m128i *outPtr = (__m128i *)(out+outIdx);
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128(horizPtr + (2 * 128) / 8);
          __m128i b = _mm_load_si128(horizPtr + 128 / 8);
          __m128i c = _mm_load_si128(horizPtr++);
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

0.2 ms / megapixel / core

15x faster, scalable across cores



Hand-optimized performance

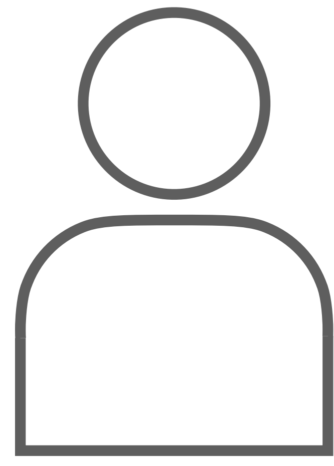
Tiled computation by 128×32

Interleaved computation of `horiz` per-tile

```
void blur_fast(int W, int H, const u16_t *in, u16_t *out) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < H-2; yTile += 32) {
  __m128i horiz[(128 / 8) * (32 + 2)];
  for (int xTile = 0; xTile < W-8; xTile += 128) {
   __m128i *horizPtr = horiz;
   for (int y = 0; y < 32 + 2; y++) {
    const size_t inIdx = xTile + (yTile+y)*W;
    const u16_t *inPtr = in+inIdx;
    for (int x = 0; x < 128; x += 8) {
     __m128i a = _mm_load_si128((const __m128i *)(inPtr));
     __m128i b = _mm_loadu_si128((const __m128i *)(inPtr + 1));
     __m128i c = _mm_loadu_si128((const __m128i *)(inPtr + 2));
     __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     __m128i avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(horizPtr++, avg);
     inPtr += 8;
    }
   }
   horizPtr = horiz;
   for (int y = 0; y < 32; y++) {
    const size_t outIdx = xTile + (yTile+y)*(W-8);
    __m128i *outPtr = (__m128i *)(out+outIdx);
    for (int x = 0; x < 128; x += 8) {
     __m128i a = _mm_load_si128(horizPtr + (2 * 128) / 8);
     __m128i b = _mm_load_si128(horizPtr + 128 / 8);
     __m128i c = _mm_load_si128(horizPtr++);
     __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     __m128i avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

0.2 ms / megapixel / core

15x faster, scalable across cores



Hand-optimized performance

Tiled computation by 128×32

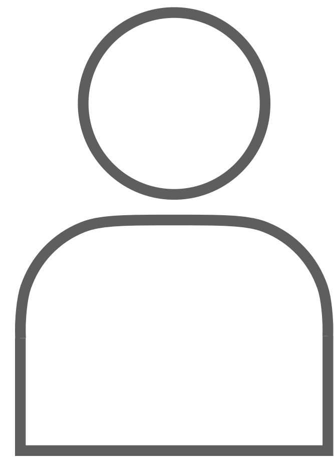
Interleaved computation of `horiz` per-tile

Vectorized across x

```
void blur_fast(int W, int H, const u16_t *in, u16_t *out) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < H-2; yTile += 32) {
    __m128i horiz[(128 / 8) * (32 + 2)];
    for (int xTile = 0; xTile < W-8; xTile += 128) {
      __m128i *horizPtr = horiz;
      for (int y = 0; y < 32 + 2; y++) {
        const size_t inIdx = xTile + (yTile+y)*W;
        const u16_t *inPtr = in+inIdx;
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128((const __m128i *)(inPtr));
          __m128i b = _mm_loadu_si128((const __m128i *)(inPtr + 1));
          __m128i c = _mm_loadu_si128((const __m128i *)(inPtr + 2));
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(horizPtr++, avg);
          inPtr += 8;
        }
      }
      horizPtr = horiz;
      for (int y = 0; y < 32; y++) {
        const size_t outIdx = xTile + (yTile+y)*(W-8);
        __m128i *outPtr = (__m128i *)(out+outIdx);
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128(horizPtr + (2 * 128) / 8);
          __m128i b = _mm_load_si128(horizPtr + 128 / 8);
          __m128i c = _mm_load_si128(horizPtr++);
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

0.2 ms / megapixel / core

15x faster, scalable across cores



Hand-optimized performance

Tiled computation by `128×32`

Interleaved computation of `horiz` per-tile

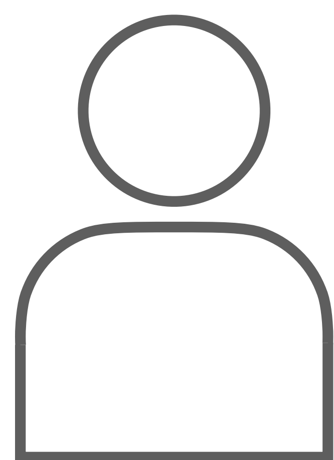
Vectorized across `x`

Parallelized rows of tiles

```
void blur_fast(int W, int H, const u16_t *in, u16_t *out) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < H-2; yTile += 32) {
    __m128i horiz[(128 / 8) * (32 + 2)];
    for (int xTile = 0; xTile < W-8; xTile += 128) {
      __m128i *horizPtr = horiz;
      for (int y = 0; y < 32 + 2; y++) {
        const size_t inIdx = xTile + (yTile+y)*W;
        const u16_t *inPtr = in+inIdx;
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128((const __m128i *)(inPtr));
          __m128i b = _mm_loadu_si128((const __m128i *)(inPtr + 1));
          __m128i c = _mm_loadu_si128((const __m128i *)(inPtr + 2));
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(horizPtr++, avg);
          inPtr += 8;
        }
      }
      horizPtr = horiz;
      for (int y = 0; y < 32; y++) {
        const size_t outIdx = xTile + (yTile+y)*(W-8);
        __m128i *outPtr = (__m128i *)(out+outIdx);
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128(horizPtr + (2 * 128) / 8);
          __m128i b = _mm_load_si128(horizPtr + 128 / 8);
          __m128i c = _mm_load_si128(horizPtr++);
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

0.2 ms / megapixel / core

15x faster, scalable across cores



Hand-optimized performance

Tiled computation by 128×32

Interleaved computation of `horiz` per-tile

Vectorized across x

Parallelized rows of tiles

```
void blur_fast(int W, int H, const u16_t *in, u16_t *out) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < H-2; yTile += 32) {
    __m128i horiz[(128 / 8) * (32 + 2)];
    for (int xTile = 0; xTile < W-8; xTile += 128) {
      __m128i *horizPtr = horiz;
      for (int y = 0; y < 32 + 2; y++) {
        const size_t inIdx = xTile + (yTile+y)*W;
        const u16_t *inPtr = in+inIdx;
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128((const __m128i *)(inPtr));
          __m128i b = _mm_loadu_si128((const __m128i *)(inPtr + 1));
          __m128i c = _mm_loadu_si128((const __m128i *)(inPtr + 2));
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(horizPtr++, avg);
          inPtr += 8;
        }
      }
      horizPtr = horiz;
      for (int y = 0; y < 32; y++) {
        const size_t outIdx = xTile + (yTile+y)*(W-8);
        __m128i *outPtr = (__m128i *)(out+outIdx);
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128(horizPtr + (2 * 128) / 8);
          __m128i b = _mm_load_si128(horizPtr + 128 / 8);
          __m128i c = _mm_load_si128(horizPtr++);
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

0.2 ms / megapixel / core

15x faster, scalable across cores

Performance engineering
requires low-level control

Performance engineering
requires low-level control

Performance engineering
requires low-level control

Low-level languages
conflate what with how

Performance engineering
requires low-level control

Low-level languages
conflate what with how

Performance engineering
requires low-level control

Low-level languages
conflate what with how

Performance engineering requires low-level **control**

Low-level languages **conflate what with how**

Writing **fast code** is **unsafe** & **unproductive**

User-schedulable languages

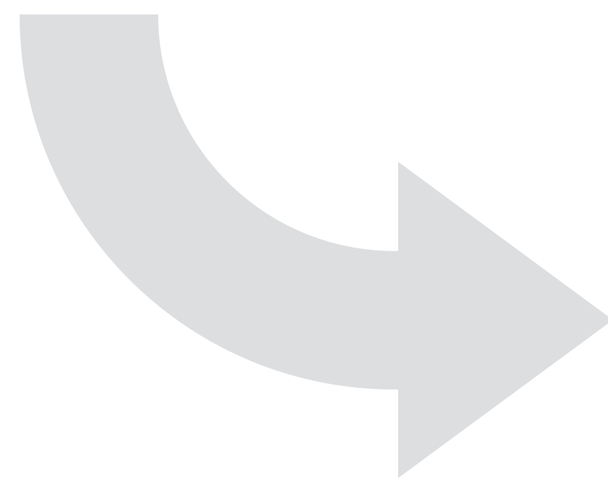
User-schedulable languages

decouple **what** to compute

User-schedulable languages
decouple **what** to compute
from **how** to compute it

User-schedulable languages
decouple **what** to compute
from **how** to compute it

User-schedulable languages decouple **what** to compute from **how** to compute it



exposed for programmer control

Halide

```
// The algorithm:
// no storage, no order
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;

// The schedule:
// defines order, locality, storage; implies bounds
out.tile(x, y, xo, yo, xi, yi, 128, 32)
   .vectorize(xi, 8).parallel(y);
horiz.compute_at(out, xo).store_at(out, xo).vectorize(x, 8);
```

Halide

```
// The algorithm:
// no storage, no order
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;
```

Halide

```
// The schedule:
// defines order, locality, storage; implies bounds
out.tile(x, y, xo, yo, xi, yi, 128, 32)
   .vectorize(xi, 8).parallel(y);
horiz.compute_at(out, xo).store_at(out, xo).vectorize(x, 8);
```

The **Schedule** controls **lowering functions to loops**

```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;
```

The **Schedule** controls **lowering functions to loops**

```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;
```

The **Schedule** controls **lowering functions to loops**

```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;

out.tile(x, y, xo, yo, xi, yi, 128, 32)
```

The **Schedule** controls **lowering functions to loops**

```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;

out.tile(x, y, xo, yo, xi, yi, 128, 32)
```

```
// for each tile
for out.yo:
  for out.xo:
```

The **Schedule** controls **lowering functions to loops**

```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;

out.tile(x, y, xo, yo, xi, yi, 128, 32)
```

```
// for each tile
for out.yo:
  for out.xo:
    // for pixel in tile
    for out.yi in [0, 32):
      for out.xi in [0, 128):
        compute out
```

The **Schedule** controls **lowering functions to loops**

```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;

out.tile(x, y, xo, yo, xi, yi, 128, 32)
```

```
// for each tile
for out.yo:
  for out.xo:
    // for pixel in tile
    for out.yi in [0, 32):
      for out.xi in [0, 128):
        compute out
```

The **Schedule** controls **lowering functions to loops**

```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;

out.tile(x, y, xo, yo, xi, yi, 128, 32)
horiz.compute_at(out, xo)
```

```
// for each tile
for out.yo:
  for out.xo:
    // for pixel in tile
    for out.yi in [0, 32):
      for out.xi in [0, 128):
        compute out
```

The **Schedule** controls **lowering functions to loops**

```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;

out.tile(x, y, xo, yo, xi, yi, 128, 32)
horiz.compute_at(out, xo)
```

compute ***here***

```
// for each tile
for out.yo:
  for out.xo:
    // for pixel in tile
    for out.yi in [0, 32):
      for out.xi in [0, 128):
        compute out
```

The **Schedule** controls **lowering functions to loops**

```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;

out.tile(x, y, xo, yo, xi, yi, 128, 32)
horiz.compute_at(out, xo)

// for each tile
for out.yo:
  for out.xo:
```

```
    // for pixel in tile
    for out.yi in [0, 32):
      for out.xi in [0, 128):
        compute out
```

The **Schedule** controls **lowering functions to loops**

```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;

out.tile(x, y, xo, yo, xi, yi, 128, 32)
horiz.compute_at(out, xo)

// for each tile
for out.yo:
  for out.xo:
    // for pixel in required tile
    for horiz.y:
      for horiz.x:
        compute horiz
    // for pixel in tile
    for out.yi in [0, 32):
      for out.xi in [0, 128):
        compute out
```

The **Schedule** controls **lowering functions to loops**

```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;

out.tile(x, y, xo, yo, xi, yi, 128, 32).parallel(y)
horiz.compute_at(out, xo).vectorize(x, 8)

// for each tile
parallel for out.yo:
  for out.xo:
    // for pixel in required tile
    for horiz.y:
      vec for horiz.x:
        compute horiz<8>
    // for pixel in tile
    for out.yi in [0, 32):
      for out.xi in [0, 128):
        compute out
```

Halide

200 µs/megapixel/core

```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;

out.tile(x, y, xo, yo, xi, yi, 128, 32)
   .vectorize(xi, 8).parallel(y);
horiz.compute_at(out, xo).store_at(out, xo).vectorize(x, 8);
```

C++

200 µs/megapixel/core

```
void blur_fast(int W, int H, const u16_t *in, u16_t *out) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < H-2; yTile += 32) {
    __m128i horiz[(128 / 8) * (32 + 2)];
    for (int xTile = 0; xTile < W-8; xTile += 128) {
      __m128i *horizPtr = horiz;
      for (int y = 0; y < 32 + 2; y++) {
        const size_t inIdx = xTile + (yTile+y)*W;
        const u16_t *inPtr = in+inIdx;
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128((const __m128i *)(inPtr));
          __m128i b = _mm_loadu_si128((const __m128i *)(inPtr + 1));
          __m128i c = _mm_loadu_si128((const __m128i *)(inPtr + 2));
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(horizPtr++, avg);
          inPtr += 8;
      }}
      horizPtr = horiz;
      for (int y = 0; y < 32; y++) {
        const size_t outIdx = xTile + (yTile+y)*(W-8);
        __m128i *outPtr = (__m128i *)(out+outIdx);
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128(horizPtr + (2 * 128) / 8);
          __m128i b = _mm_load_si128(horizPtr + 128 / 8);
          __m128i c = _mm_load_si128(horizPtr++);
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

Halide

200 µs/megapixel/core

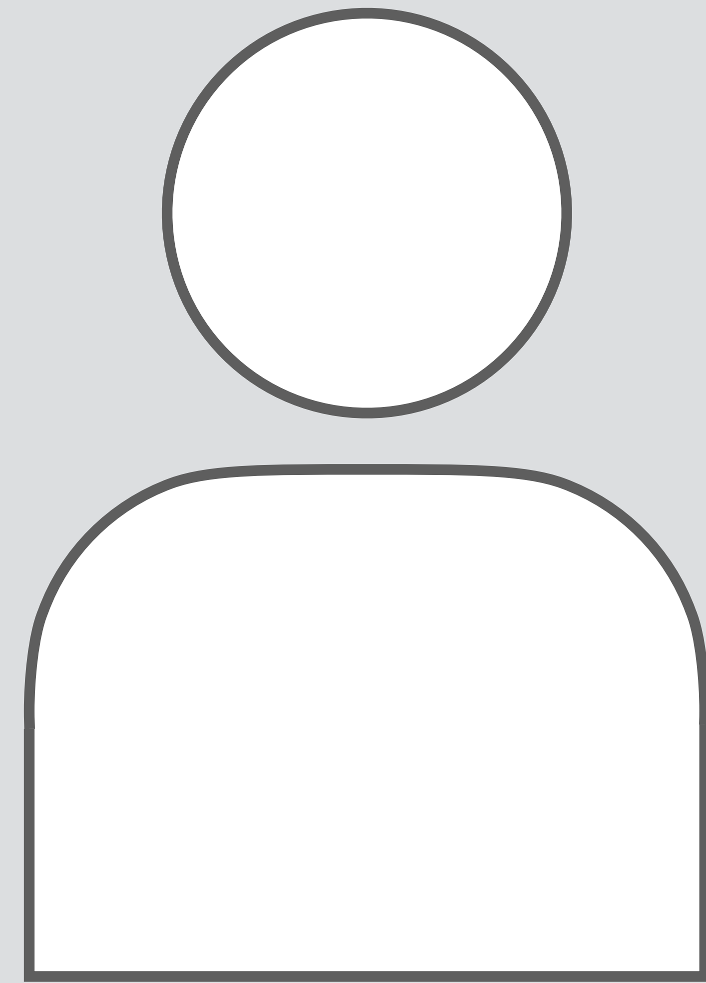
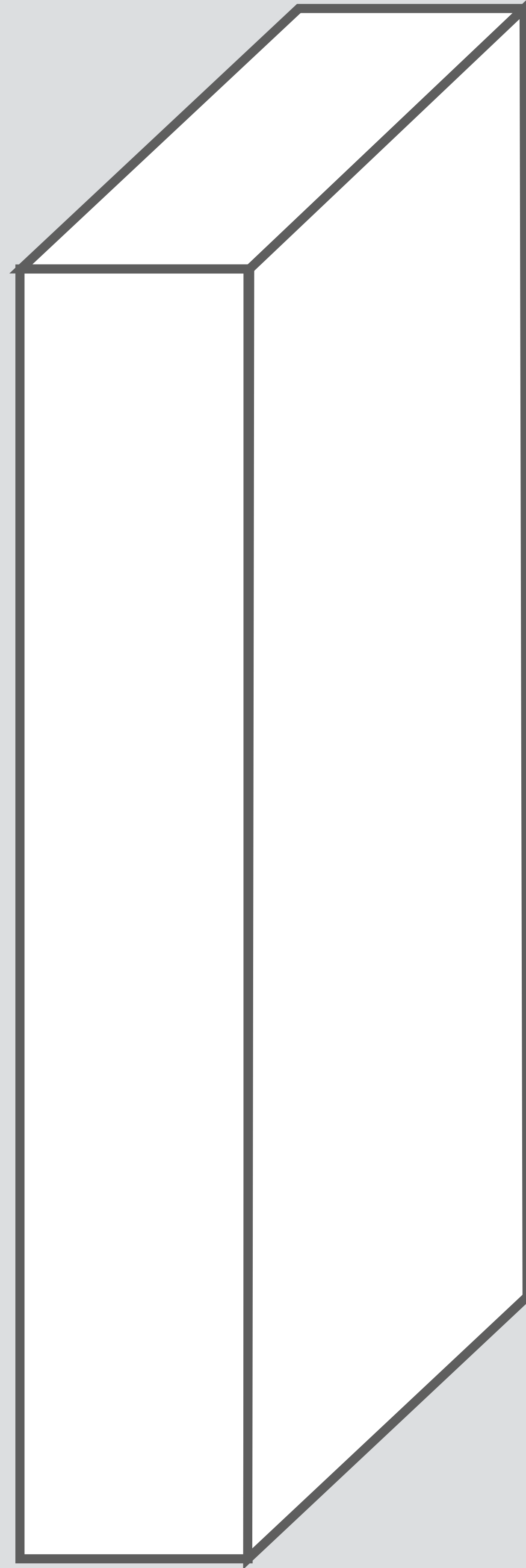
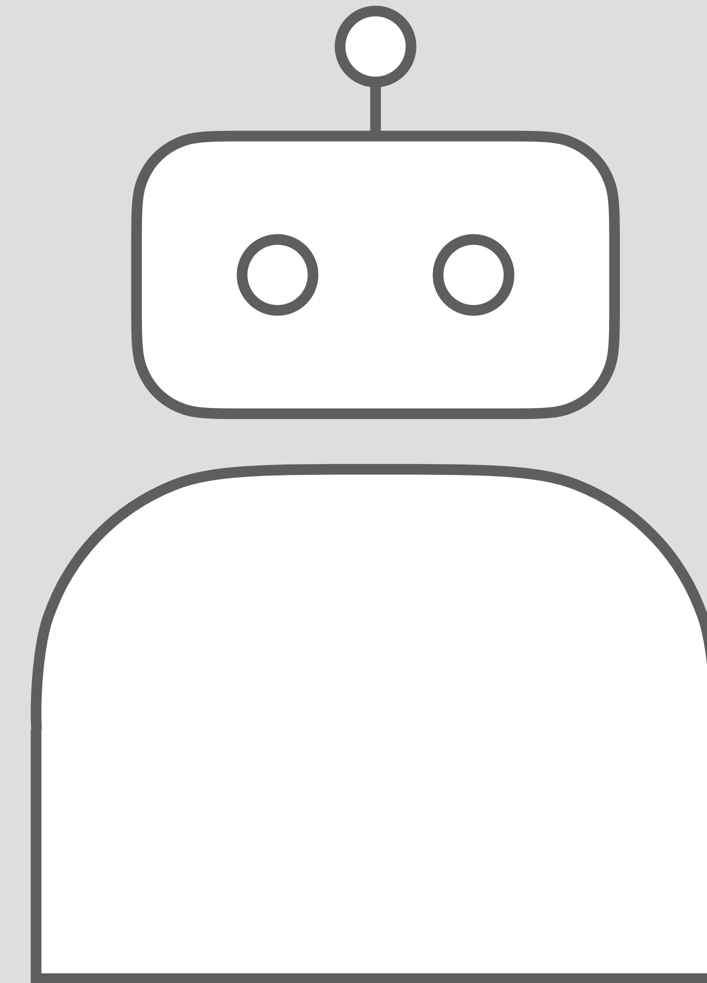
```
horiz(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
out(x, y) = (horiz(x, y-1) + horiz(x, y) + horiz(x, y+1))/3;

out.tile(x, y, xo, yo, xi, yi, 128, 32)
   .vectorize(xi, 8).parallel(y);
horiz.compute_at(out, xo).store_at(out, xo).vectorize(x, 8);
```

Language guarantee:
changing the schedule
can never change the result
or cause memory bugs.

C++

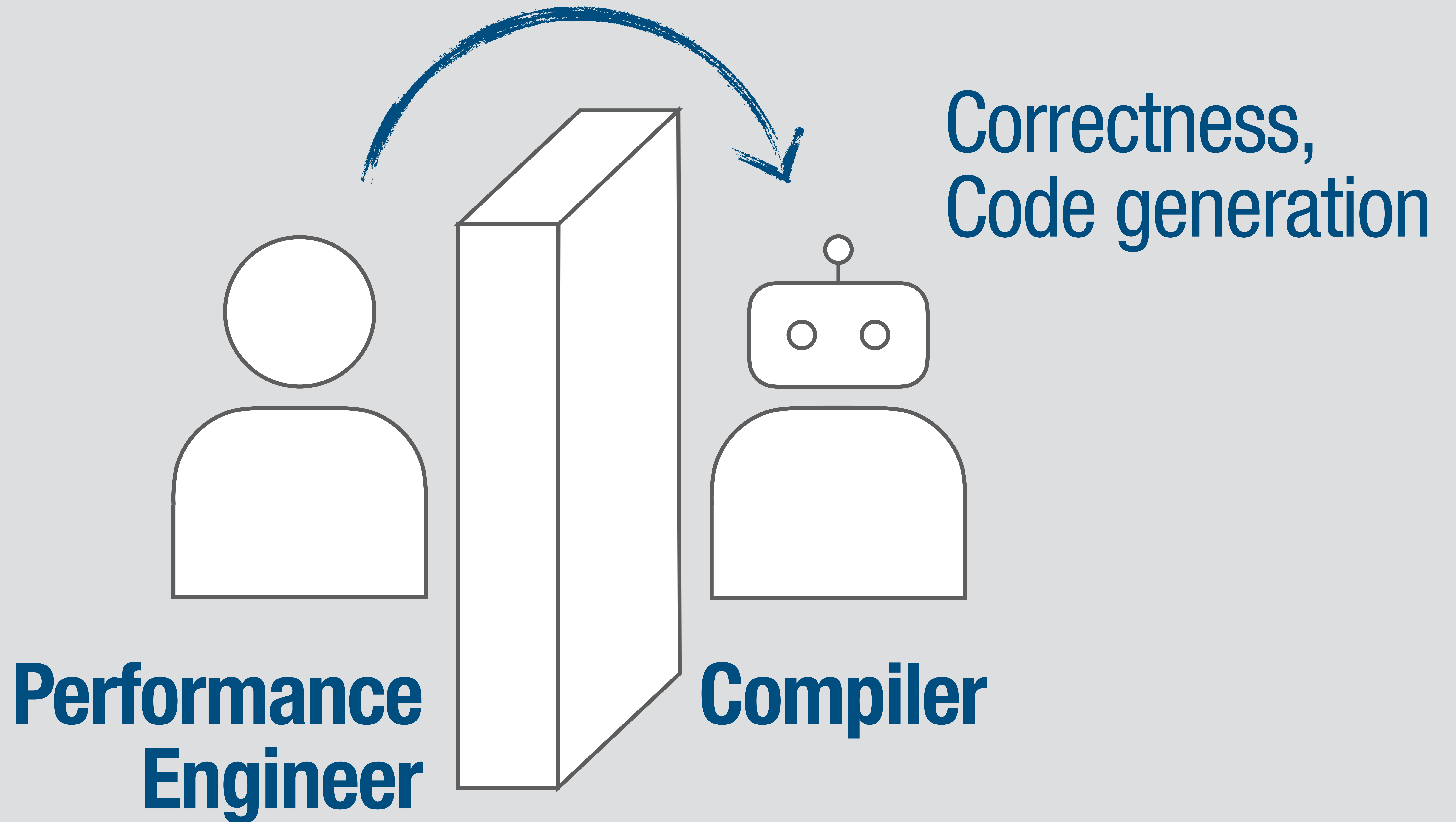
200 µs/megapixel/core

```
void blur_fast(int W, int H, const u16_t *in, u16_t *out) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < H-2; yTile += 32) {
    __m128i horiz[(128 / 8) * (32 + 2)];
    for (int xTile = 0; xTile < W-8; xTile += 128) {
      __m128i *horizPtr = horiz;
      for (int y = 0; y < 32 + 2; y++) {
        const size_t inIdx = xTile + (yTile+y)*W;
        const u16_t *inPtr = in+inIdx;
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128((const __m128i *)(inPtr));
          __m128i b = _mm_loadu_si128((const __m128i *)(inPtr + 1));
          __m128i c = _mm_loadu_si128((const __m128i *)(inPtr + 2));
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(horizPtr++, avg);
          inPtr += 8;
      }}
      horizPtr = horiz;
      for (int y = 0; y < 32; y++) {
        const size_t outIdx = xTile + (yTile+y)*(W-8);
        __m128i *outPtr = (__m128i *)(out+outIdx);
        for (int x = 0; x < 128; x += 8) {
          __m128i a = _mm_load_si128(horizPtr + (2 * 128) / 8);
          __m128i b = _mm_load_si128(horizPtr + 128 / 8);
          __m128i c = _mm_load_si128(horizPtr++);
          __m128i sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          __m128i avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

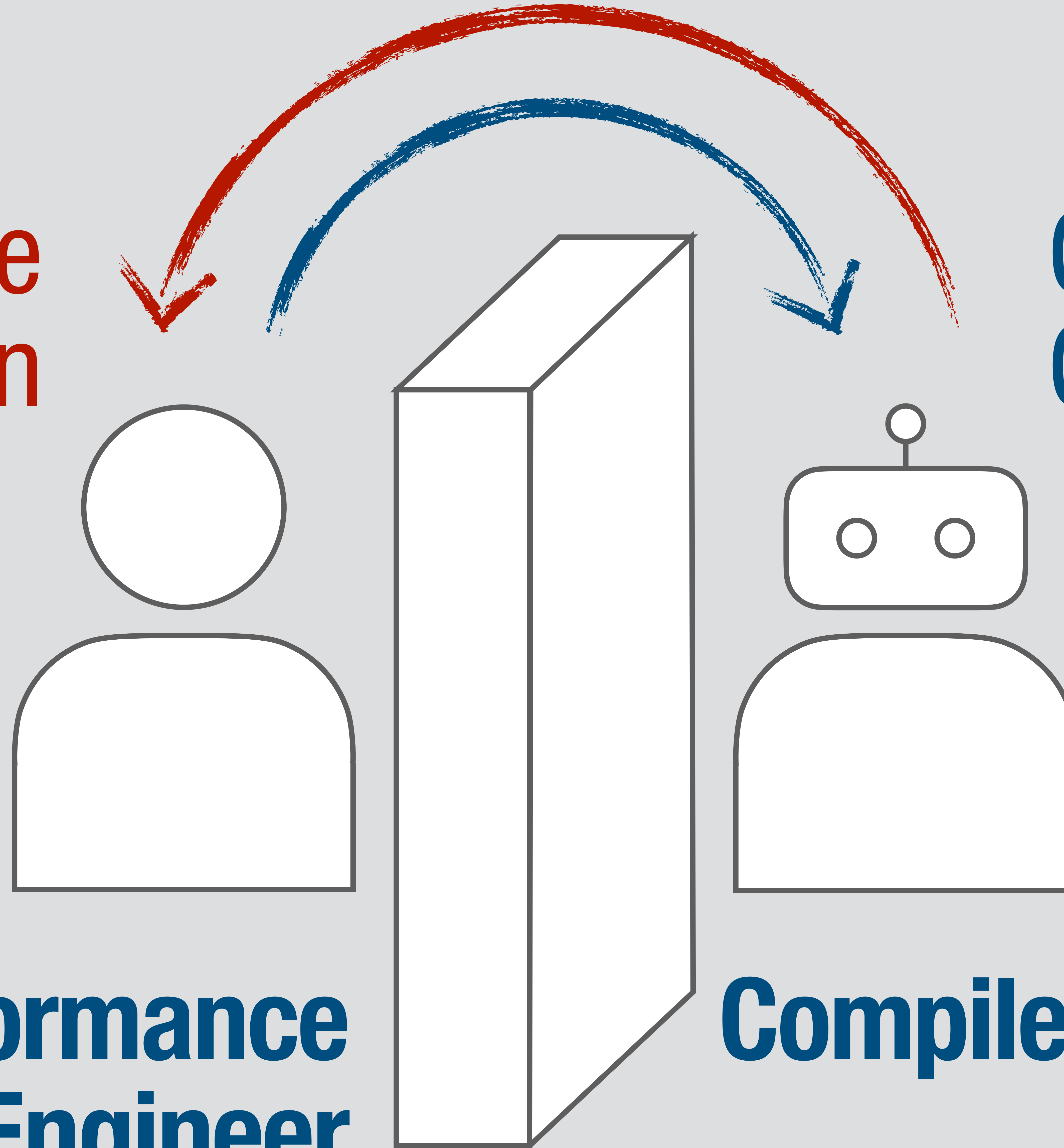




Performance Engineer

Compiler



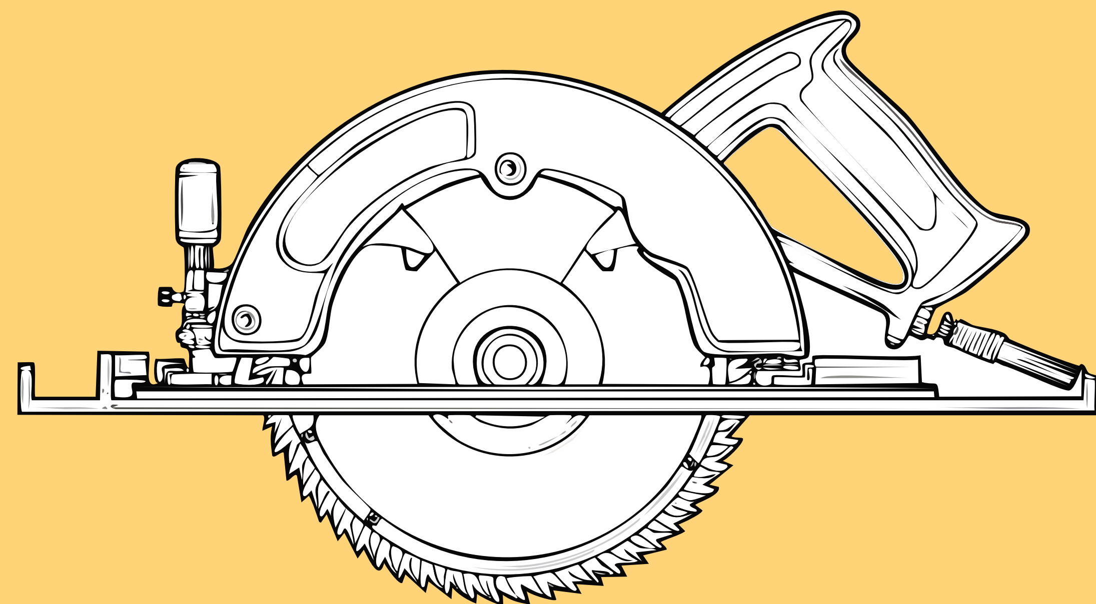
Correctness,
Code generation
Performance
Engineer
Compiler


Performance
optimization
Correctness,
Code generation
Performance
Engineer
Compiler

Halide

Halide	TVM	GraphIt
Taichi	TACO	LIFT/Elevate
Exo	MLIR Transform dialect	…

Halide	TVM	GraphIt
Taichi	TACO	LIFT/Elevate
Exo	MLIR Transform dialect	…

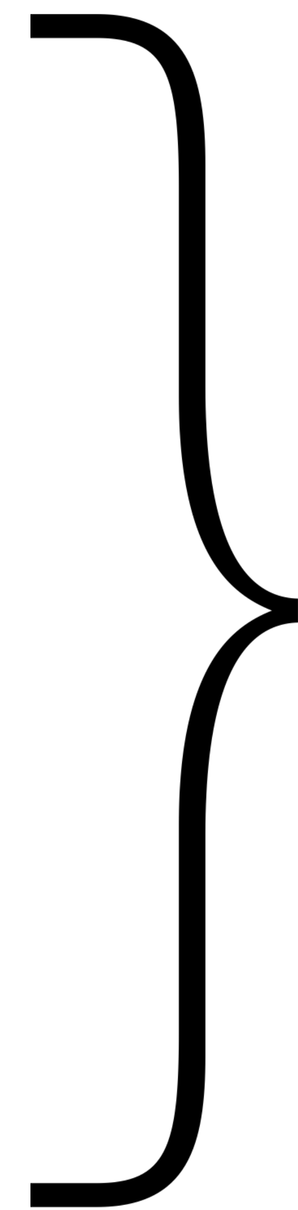


Safe & productive **control** for **high-performance**

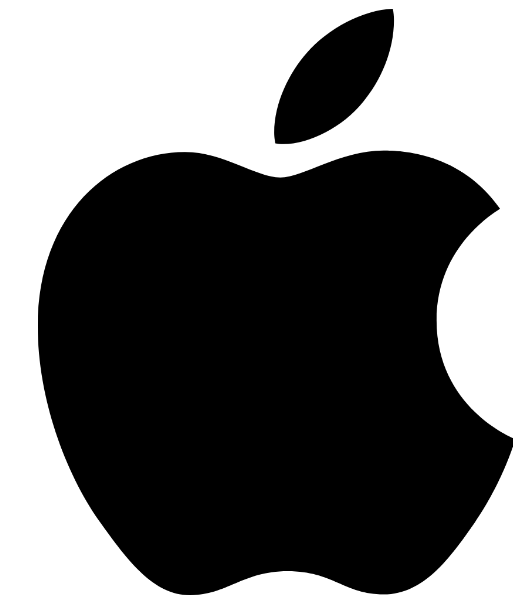
Halide

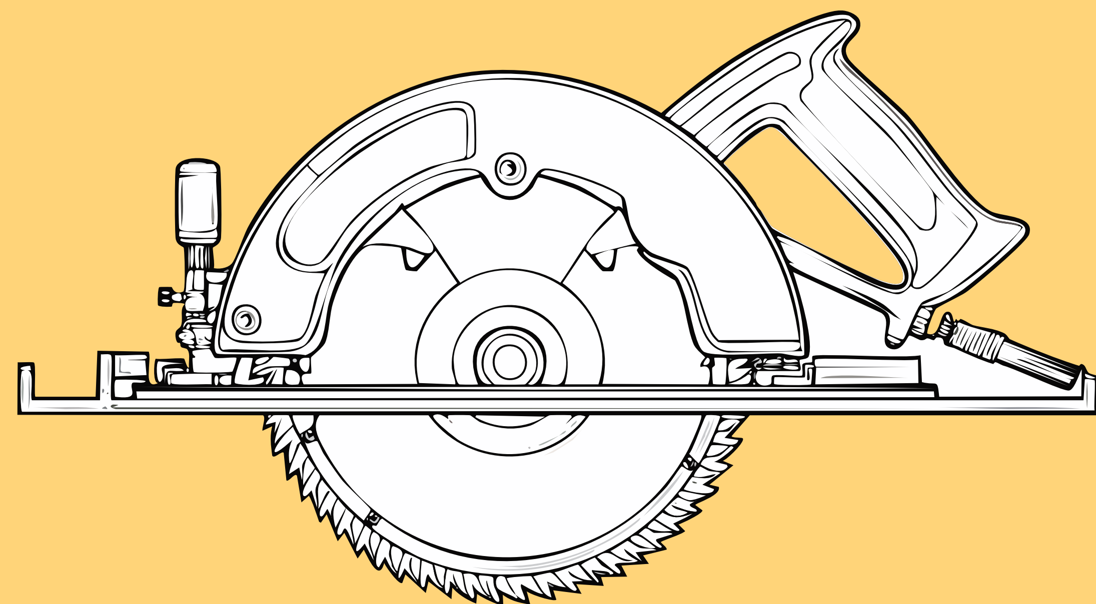
Taichi

Exo



Dozens of programmers serve **billions of users**





Safe & productive **control** for **high-performance**