Coexecutor Runtime

1 **Coexecutor Runtime** 3
1.1 Introduction
1.2 Validation
1.3 Use cases - Examples

1. Coexecutor Runtime

1.1 Introduction

The *Coexecutor Runtime* will be released once the research work done during the last year is published. In addition, we intend to publish several works related to the runtime, currently in progress, due to the importance and popularity that oneAPI has nowadays. We are currently studying the behavior of oneAPI for other architectures (apart from CPU and integrated GPU), as well as elaborating a new load balancing algorithm more appropriate for co-execution with this runtime, all thanks to the study of the behavior of the *Coexecutor Runtime* during its development and experimentation.

Nevertheless, if any reviewer would like to see the internal code of the *Coexecutor Runtime* to compile and execute some benchmarks, please, contact with Raúl Nozal. The code will only be provided for reviewing purposes, never to be published until we perform the official open-source releasing in the next months.

1.2 Validation

Coexecutor Runtime has been validated in another two architectures. One is a Desktop system, as a low profile computing node. Another is a powerful server node, part of a cluster infrastructure (Intel DevCloud).

```
CPU:

      Architecture:                     x86_64
      CPU op-mode(s):                   32-bit, 64-bit
      Byte Order:                       Little Endian
      Address sizes:                    39 bits physical, 48 bits virtual
      CPU(s):                           4
      On-line CPU(s) list:              0-3
      Thread(s) per core:               2
      Core(s) per socket:               2
      Socket(s):                        1
      NUMA node(s):                     1
      Vendor ID:                        GenuineIntel
      CPU family:                       6
      Model:                            78
      Model name:                       Intel(R) Core(TM) i5-6200U CPU @
       ↪  2.30GHz
      Stepping:                         3
      CPU MHz:                          600.053
      CPU max MHz:                      2800,0000
      CPU min MHz:                      400,0000
      BogoMIPS:                         4801.00
      Virtualization:                   VT-x
      L1d cache:                        64 KiB
      L1i cache:                        64 KiB
      L2 cache:                         512 KiB
      L3 cache:                         3 MiB
      NUMA node0 CPU(s):                0-3

Memory (MiB):
            total       used       free     shared  buff/cache   available
      Mem:   7844        927       5473        307        1443        6383
      Swap:     0          0          0
```

Listing 1: Desktop node

```
CPU:

  Architecture:                    x86_64
  CPU op-mode(s):                  32-bit, 64-bit
  Byte Order:                      Little Endian
  Address sizes:                   39 bits physical, 48 bits virtual
  CPU(s):                          12
  On-line CPU(s) list:             0-11
  Thread(s) per core:              2
  Core(s) per socket:              6
  Socket(s):                       1
  NUMA node(s):                    1
  Vendor ID:                       GenuineIntel
  CPU family:                      6
  Model:                           158
  Model name:                      Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz
  Stepping:                        10
  CPU MHz:                         4177.750
  CPU max MHz:                     4700.0000
  CPU min MHz:                     800.0000
  BogoMIPS:                        7399.70
  Virtualization:                  VT-x
  L1d cache:                       192 KiB
  L1i cache:                       192 KiB
  L2 cache:                        1.5 MiB
  L3 cache:                        12 MiB
  NUMA node0 CPU(s):               0-11

Memory (MiB):
         total      used      free    shared  buff/cache   available
  Mem:   64133       368     63538         1         226       63179
  Swap:   1951         0      1951
```

Listing 2: DevCloud node

1.2.1 Performance

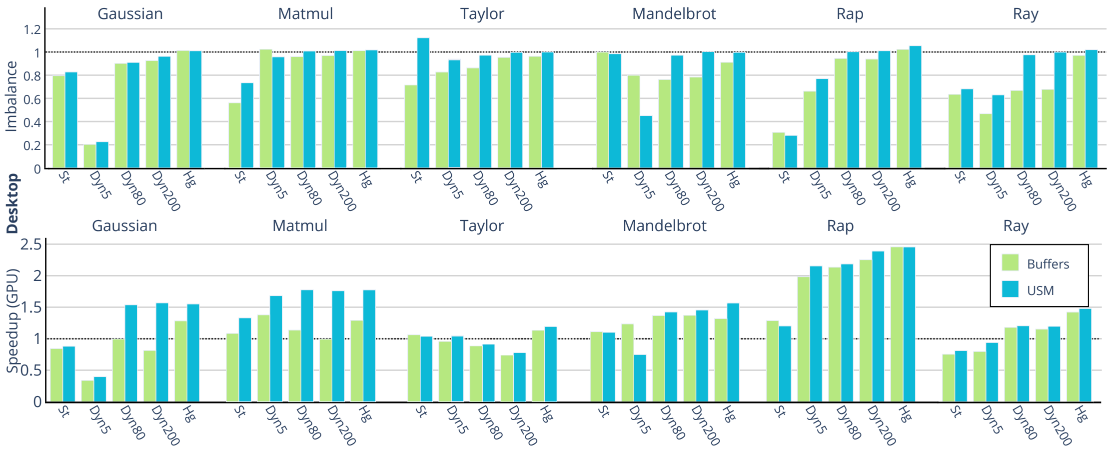

Figure 1.1: Balancing efficiency and Speedup (GPU/Coexecuting) for a set of benchmarks in a Desktop node.

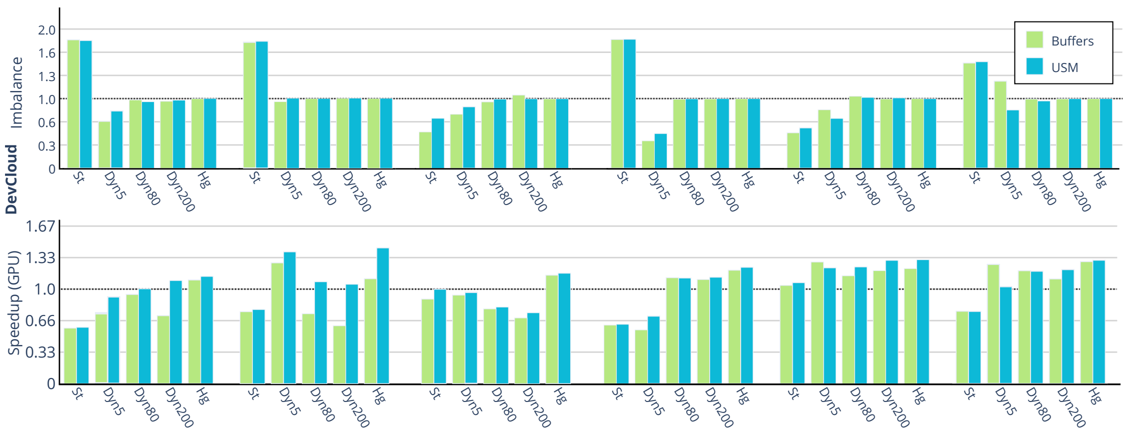

Figure 1.2: Speedup (GPU/Coexecuting) for a set of benchmarks in a Devcloud server.

1.2.2 Energy consumption and efficiency

The DevCloud does not allow using neither RAPL hardware counters nor perf-utils. Therefore, its energy measurements cannot be collected and analyzed.

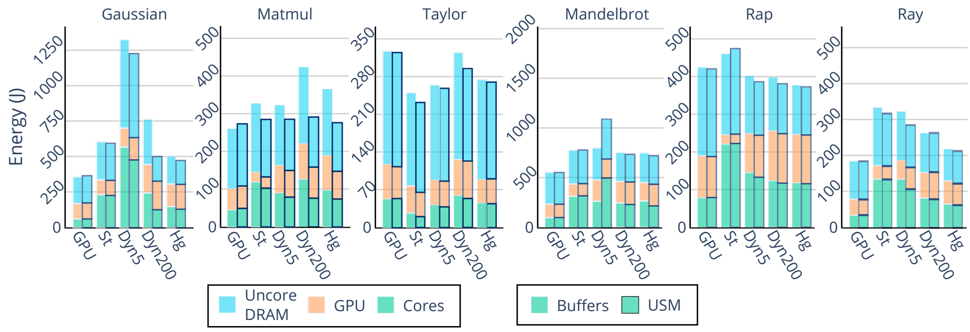

Figure 1.3: Energy consumption for a set of benchmarks in a Desktop system.

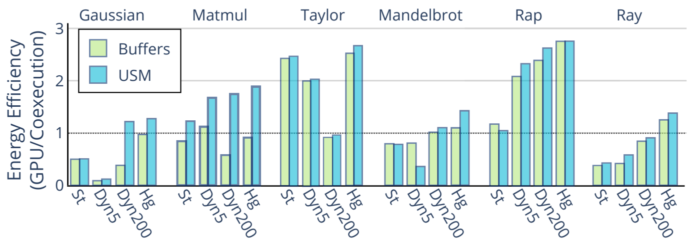

Figure 1.4: Energy efficiency (GPU/Coexecuting) for a set of benchmarks in a Desktop system.

1.2.3 Scalability

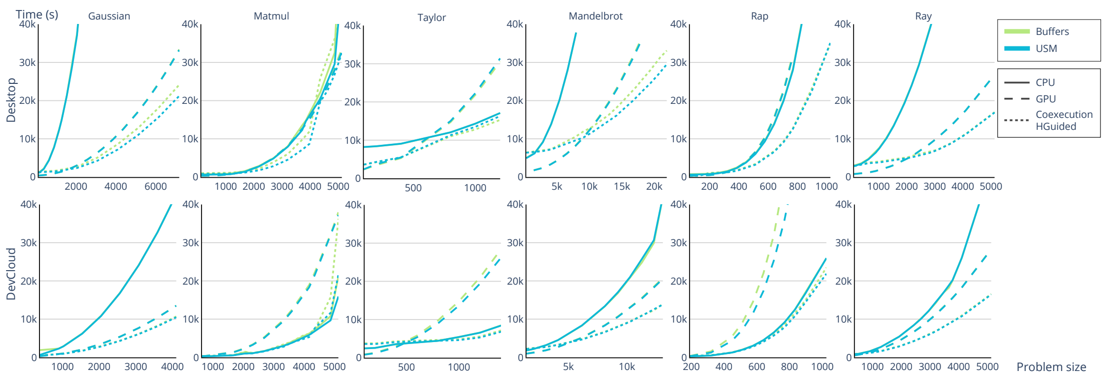

Figure 1.5: Scalability when using CPU-only, GPU-only and Coexecutor Runtime, for USM and Buffers memory models.

1.3 Use cases - Examples

1.3.1 SAXPY

Three independent SAXPY examples are depicted, showing the expressiveness of the *Coexecutor Runtime*. Two types of kernel definition are shown. One it is based on a explicit embedded context (lambda function), while the other allows accessing extended methods and callback functions, based on the `CommanderKernel` interface. In the latter, the programmer uses the init method and also the callback for kernel computation. Moreover, the `extended_computation_usm` function shows how the shared memory is managed from the point of view of the programmer. As can be seen in the three real examples, the *Coexecutor Runtime* hides all the implementation details, easing the use of its co-execution capabilities to exploit easily any oneAPI program.

```
class SAXPY : public CommanderKernel {
 public:
  SAXPY(int *x_ptr, int *y_ptr, int *out_ptr, size_t data_len,
        float scalar_float)
      : m_x_ptr(x_ptr), m_y_ptr(y_ptr), m_out_ptr(out_ptr),
        m_scalar_float(scalar_float), m_data_len(data_len) {}

  // pre-setup configuration - Director scope (shared internally between Coexecution Units)
  void init(coexecutor_unit *counit) {
    counit->add_buffer<int, 1>(0, m_x_ptr, sycl::range<1>(m_data_len));
  }

  // /* @callback */
  // void init_completed() {}

  program_size size() { return m_data_len; }

  void compute(coexecutor_unit *counit, package pkg) {
    std::cout << "[" << counit->id() << "] computing...\n";

    sycl::buffer<int, 1> buf_x =
        *counit->get_buffer<int, 1>(0); // use all buffer
    sycl::buffer<int, 1> buf_y(m_y_ptr + pkg.offset, sycl::range<1>(pkg.size));
    sycl::buffer<int, 1> buf_out(m_out_ptr + pkg.offset,
                                 sycl::range<1>(pkg.size));

    // communicate with the Commander to dispatch a transaction
    // Director-Coexecution Unit communication
    counit->dispatch([&](sycl::handler &h) {
      auto R = sycl::range<1>(pkg.size);

      auto x = buf_x.get_access<sycl::access::mode::read>(h);
      auto y = buf_y.get_access<sycl::access::mode::read>(h);
      auto out = buf_out.get_access<sycl::access::mode::discard_write>(h);

```

```
      auto scalar = (int)m_scalar_float; // caching

      // Parallel region for the workload package
      h.parallel_for(R, [=](sycl::item<1> it) {
        auto tid = it.get_linear_id();
        out[tid] = x[tid] * scalar + y[tid];
      });
    });
  }

  /* @callback */
  void compute_completed(coexecutor_unit *counit, package pkg) {
    // Statistics per package - related with the Coexecution Unit
    std::cout << "[" << counit->id() << "] package " << pkg.id
              << " computed with throughput " << pkg.throughput << "\n";
    counit->dump_statistics();
  }

 private: // organized data:
  int *m_x_ptr;
  int *m_y_ptr;
  int *m_out_ptr;
  size_t m_data_len;
  float m_scalar_float;
};

void extended_computation_usm() {
  coexecutor_runtime<dyn> runtime;
  runtime.config(CounitSet::CpuGpu, 12); // dynamic; 12 packages

  // input data
  size_t N = 10000000;
  std::vector<int> x;
  int* y;
  std::vector<int> out(N);
  float scalar = 3.14;

  // assigns
  x.assign(N, 1);
  y = runtime.alloc<int>(N);
  for (int i = 0; i < N; ++i) {
    x[i] = x[i] + i;
    y[i] = y[i] + 2 * i;
  }
  std::cout << "data x: ";
  for (int v : x) {
    std::cout << v << " ";
  }
  std::cout << "\n";
```

```
  std::cout << "data y: ";
  for (int i=0; i<N; ++i) {
    std::cout << y[i] << " ";
  }
  std::cout << "\n";

  SAXPY program{x.data(), y, out.data(), N, scalar};
  runtime.run(program);

  std::cout << "data output: ";
  for (int v : out) {
    std::cout << v << " ";
  }
  std::cout << "\n";

  runtime.free<int>(y);
}

void extended_computation() {
  // input data
  size_t N = 100000000;
  std::vector<int> x;
  std::vector<int> y;
  std::vector<int> out(N);
  float scalar = 3.14;

  // assigns
  x.assign(N, 1);
  y.assign(N, 2);
  for (int i = 0; i < N; ++i) {
    x[i] = x[i] + i;
    y[i] = y[i] + 2 * i;
  }
  std::cout << "data x: ";
  for (int v : x) {
    std::cout << v << " ";
  }
  std::cout << "\n";
  std::cout << "data y: ";
  for (int v : y) {
    std::cout << v << " ";
  }
  std::cout << "\n";

  coexecutor_runtime<dyn> runtime;
  runtime.config(CounitSet::CpuGpu, 20); // 20 packages given dynamically
  SAXPY program{x.data(), y.data(), out.data(), N, scalar};
  runtime.run(program);

```

```
  std::cout << "data output: ";
  for (int v : out) {
    std::cout << v << " ";
  }
  std::cout << "\n";
}

void immediate_computation() {
  // input data
  std::vector<int> data;
  float datav = 3.14;
  size_t N = 200000;

  // assigns
  data.assign(N, 1);
  for (int i = 0; i < N; ++i) {
    data[i] = data[i] + i;
  }
  std::cout << "data: ";
  for (int x : data) {
    std::cout << x << " ";
  }
  std::cout << "\n";

  {
    std::mutex mut;
    coexecutor_runtime<hg> runtime;
    runtime.config(CounitSet::CpuGpu, 0.35); // hguided; CPU has the 35% of the total computation power
    runtime.launch(data.size(), [&](coexecutor_unit *counit, package pkg) {
      {
        std::lock_guard<std::mutex> lk(mut);
        std::cout << "[" << counit->id() << "] computing " << pkg.size << " with offset " << pkg.offset << "\n";
      }

      sycl::buffer<int, 1> buf_input(data.data() + pkg.offset,
                                     sycl::range<1>(pkg.size));

      counit->dispatch([&](sycl::handler &h) {
        auto R = sycl::range<1>(pkg.size);

        auto input = buf_input.get_access<sycl::access::mode::read_write>(h);

        h.parallel_for(R, [=](sycl::item<1> it) {
          auto tid = it.get_linear_id();
          input[tid] = input[tid] * datav;
```

```
        });
      });
    });
  }

  std::cout << "data output: ";
  for (int x : data) {
    std::cout << x << " ";
  }
  std::cout << "\n";
}

int main(int argc, char *argv[]) {

  immediate_computation();
  extended_computation();
  extended_computation_usm();

  return 0;
}
```

1.3.2 Gaussian blur

Gaussian blur performs a gaussian filter to an image. Only the relevant lines for the programmer are shown, discarding the image and filter initialization (gaussian_cpu code). Therefore, initialization of the gaussian program and its validations are omitted for clarifying purposes. The *Coexecutor Runtime* acts transparently for the programmer, managing all the needed resources regading oneAPI primitives and scheduling behavior.

```
#include "gaussian.h"

int main(int argc, char *argv[]) {
  // setup initial data, prepare filter and image
  int filterSize = 61;
  size_t N = 2048;
  int imageWidth = N, imageHeight = N;
  gaussian_cpu gaussian(imageWidth, imageHeight, filterSize);

  uchar4 *a_ptr = (uchar4 *)malloc(gaussian._total_size * sizeof(uchar4));
  float *b_ptr = (float *)malloc(gaussian._filter_total_size *
    ↪ sizeof(float));
  uchar4 *c_ptr = (uchar4 *)malloc(gaussian._total_size * sizeof(uchar4));

  gaussian.set_buffers(a_ptr, b_ptr, c_ptr);
  gaussian.build(); // compose images

  {
    coexecutor_runtime<dyn> runtime; // Coexecutor-Runtime
    // use the dynamic scheduler with CPU-GPU
```

```
    // co-execution using 32 packages
    runtime.config(CounitSet::CpuGpu, 32);
    runtime.launch(
        gaussian._total_size,
        [&](coexecutor_unit *counit, package pkg) { // launch the Gaussian
         ↪  bench
          std::cout << "[" << counit->id() << "] computing " << pkg.size
                    << " with offset " << pkg.offset << "\n";

          auto Rinput = sycl::range<1>(gaussian._total_size);
          auto Rfilter = sycl::range<1>(gaussian._filter_total_size);
          auto cols = imageWidth;
          auto rows = imageHeight;

          sycl::buffer<uchar4, 1> buf_input(gaussian._a, Rinput);
          sycl::buffer<float, 1> buf_filterWeight(gaussian._b, Rfilter);
          sycl::buffer<uchar4, 1> buf_blurred((gaussian._c + pkg.offset),
                                               sycl::range<1>(pkg.size));

          counit->dispatch([&](sycl::handler &h) {
            auto R = sycl::range<1>(pkg.size);

            auto input =
             ↪  buf_input.get_access<sycl::access::mode::read>(h);
            auto filterWeight =
                buf_filterWeight.get_access<sycl::access::mode::read>(h);
            auto blurred =

                 ↪  buf_blurred.get_access<sycl::access::mode::discard_write>(h);

            h.parallel_for(R, [=](sycl::item<1> it) {
              auto tid = it.get_linear_id() + pkg.offset;
              int r = tid / cols;
              int c = tid % cols;

              int middle = filterSize / 2;
              float blurX = 0.f;
              float blurY = 0.f;
              float blurZ = 0.f;

              int width = cols - 1;
              int height = rows - 1;

              for (int i = -middle; i <= middle; ++i) {
                for (int j = -middle; j <= middle; ++j) {

                  int h = r + i;
                  int w = c + j;
                  if (h > height || h < 0 || w > width || w < 0) {
```

```
                        continue;
                      }

                      int idx = w + cols * h;

                      float pixelX = (input[idx].x());
                      float pixelY = (input[idx].y());
                      float pixelZ = (input[idx].z());

                      idx = (i + middle) * filterSize + j + middle;
                      float weight = filterWeight[idx];

                      blurX += pixelX * weight;
                      blurY += pixelY * weight;
                      blurZ += pixelZ * weight;
                    }
                  }

                  tid -= pkg.offset;
                  blurred[tid].x() = (unsigned char)cl::sycl::round(blurX);
                  blurred[tid].y() = (unsigned char)cl::sycl::round(blurY);
                  blurred[tid].z() = (unsigned char)cl::sycl::round(blurZ);
                });
              });
            });

    // show runtime info
    runtime.dump_statistics();
  }

  if (gaussian.compare_gaussian_blur(THRESHOLD)) { // unit test
    std::cout << "Success\n";
  } else {
    std::cout << "Failure\n";
  }

  // explicit free memory
  free(a_ptr);
  free(b_ptr);
  free(c_ptr);

  return 0;
}
```