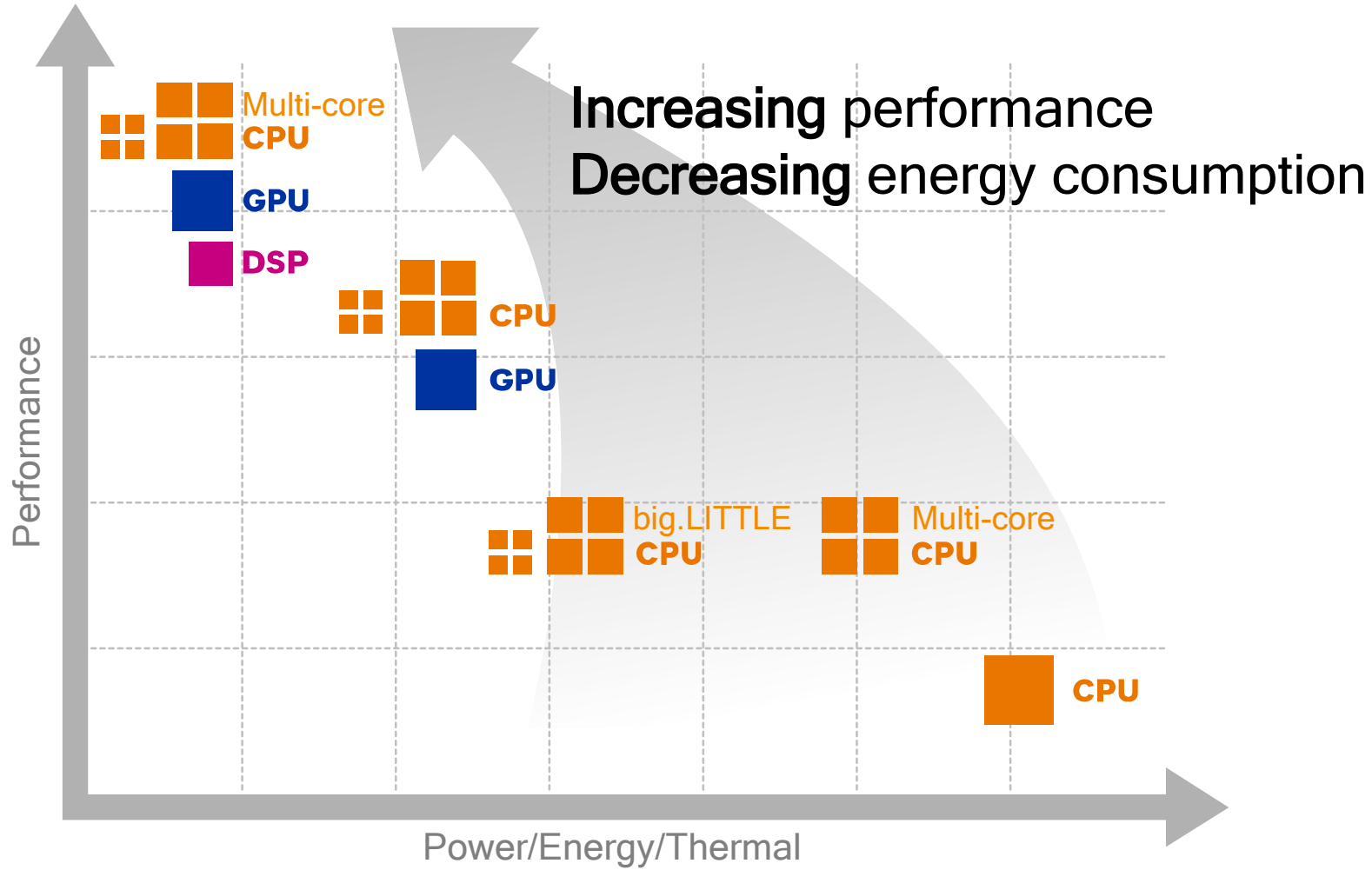
Heterogeneous Computing Made Easy: Qualcomm® Symphony System Manager SDK

Wenjia Ruan
Sr. Engineer, Advanced Content Group
Qualcomm Technologies, Inc.

May 2017

Heterogeneous Computing

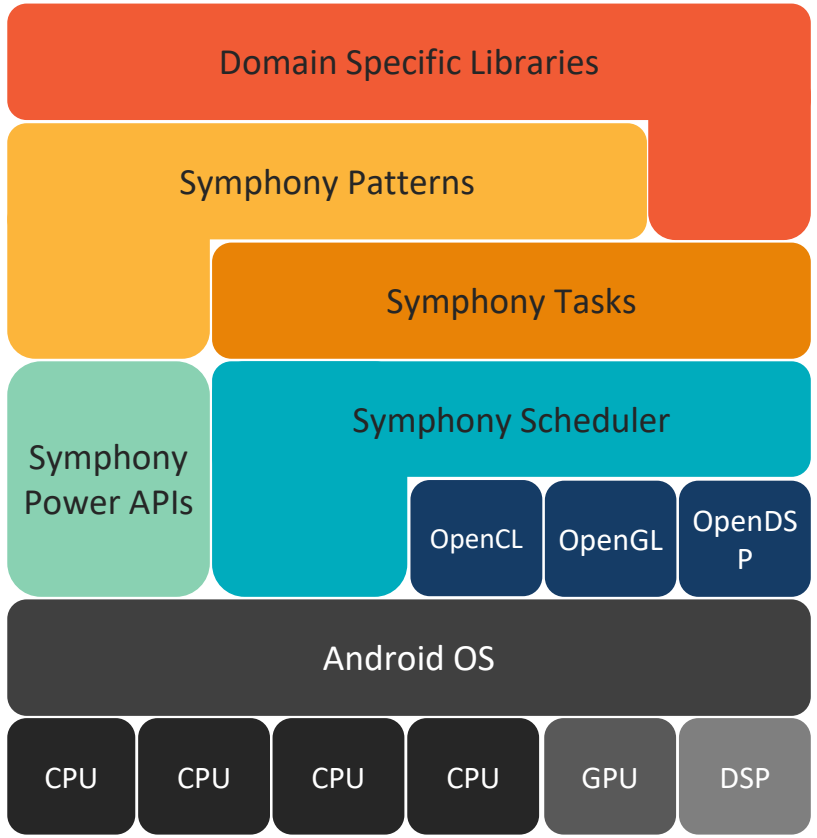
High Performance in a power constraint environment



Symphony System Manager

A heterogeneous computing solution

- **Designed to Simplify** heterogeneous computing, i.e. **use all execution units**, on devices with Qualcomm® Snapdragon Mobile Platform
- Abstracts task scheduling, memory management, and kernel synchronization across the CPU, GPU, and DSP
- Integrates with OpenCL, OpenGL ES, and OpenDSP
- Supports cool running devices with long battery life through power and affinity APIs

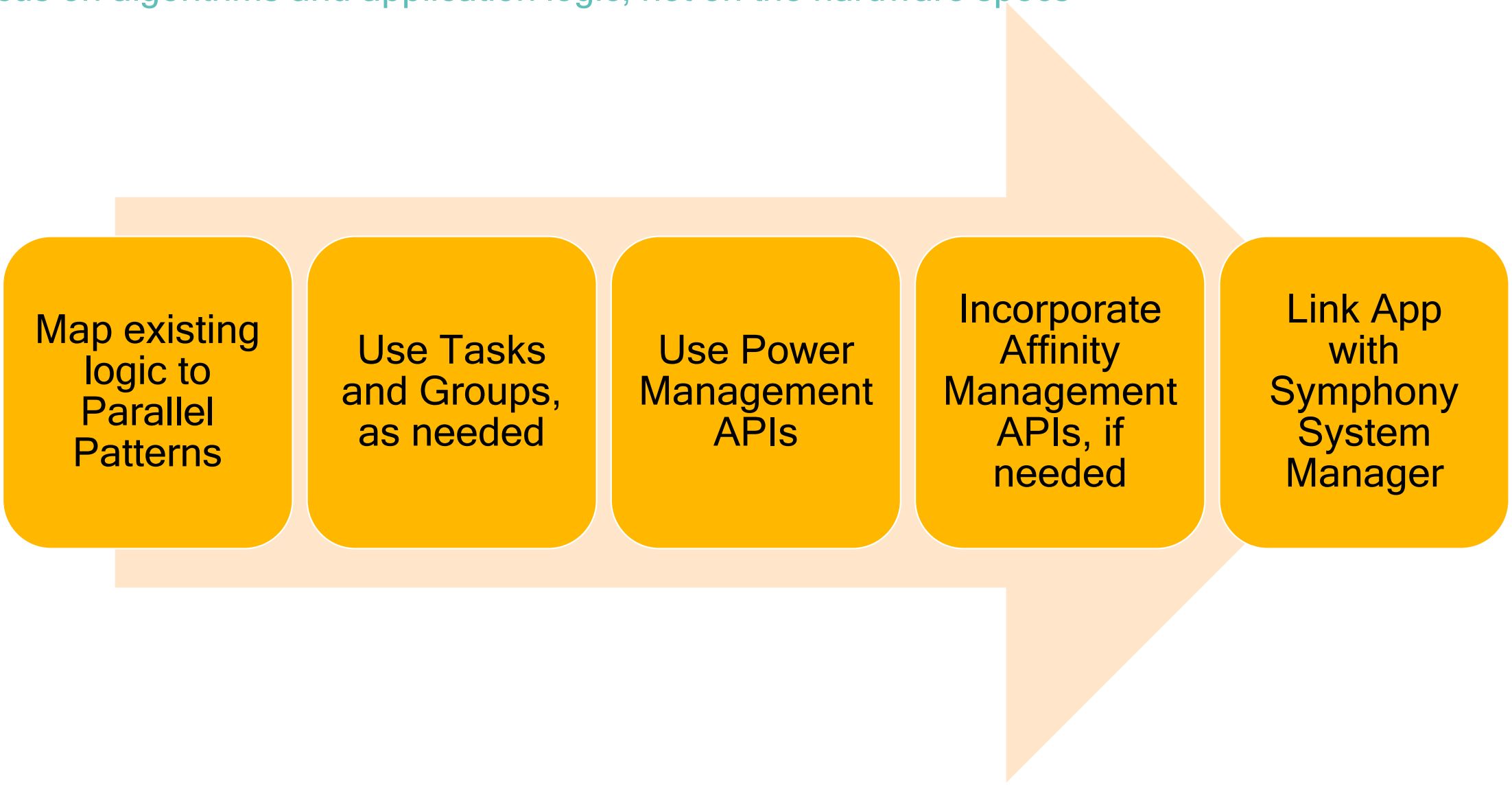


Components

compute	**Kernel:**	Computation to be executed on the CPU/GPU or DSP
data	**Buffer:**	Array-like data structure transparently accessible across CPU/GPU or DSP
	Pattern:	Implicit parallelism through structured control flow and data access
control	**Task:**	Computation construct bound with data; asynchronously executed
	Affinity:	Controls specific CPU cores/controls for use
platform	**Power:**	Control power dissipation to achieve quality of service

Workflow

Focus on algorithms and application logic, not on the hardware specs



Kernels: GPU kernels – Qualcomm ® Adreno™ GPU

Easy to import existing OpenCL/OpenGL kernels into Symphony

OpenCL

```
#define OCL_KERNEL(name, k) std::string const
name##_string = #k

OCL_KERNEL(vadd_kernel,
__kernel void vadd(__global float* A,
                   __global float* B,
                   __global float* C)
{
  unsigned int i = get_global_id(0);
  C[i] = A[i] + B[i];
});

// Create kernel using OpenCL string
auto gk = symphony::create_gpu_kernel
                  <symphony::buffer_ptr<float>,
                   symphony::buffer_ptr<float>,
                   symphony::buffer_ptr<float>>
                  (vadd_kernel_string, "vadd");
```

OpenGL

```
const char *vadd_shader_code = R"GLCODE(
  #version 310 es
  precision highp float;
  layout(local_size_x = 16) in;
…
  void main() {
    uint i = gl_GlobalInvocationID.x;
    output_data.elements[i] =
          input_data0.elements[i] +
input_data1.elements[i];
  }
)GLCODE";

// Create kernel using OpenGL string
auto gk = symphony::create_gpu_kernel
                  <symphony::buffer_ptr<float>,
                   symphony::buffer_ptr<float>,
                   symphony::buffer_ptr<float>>
                  (symphony::gl, vadd_shader_code);
```

Kernels: Poly kernels

Write many, run somewhere

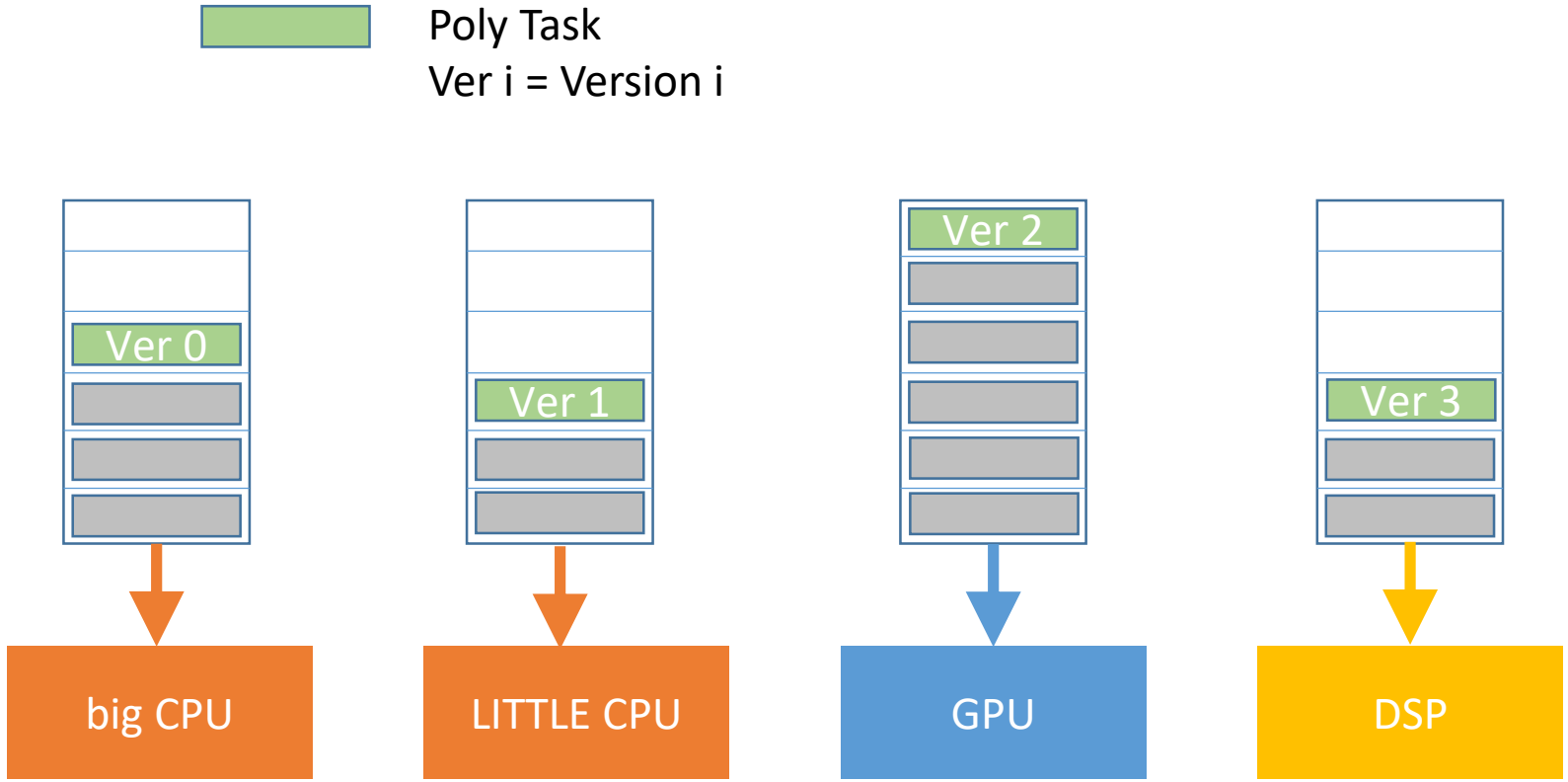
- Programmer/compiler specifies multiple implementations of the same algorithm, e.g. sorting
- Each implementation tailored to specific device: CPU/GPU/DSP
- Symphony dispatches implementation best suited to runtime conditions such as load on each device, thermal, etc.

```
auto t = symphony::beta::launch(std::tuple(ck, gk, hk), args…);

…

t->wait_for();
```

Kernels: Poly kernels

Write many, run somewhere

Scheduling technique: Alternative encoding + Idempotent scheduling + Deferred finish



Late Binding Based on Relative Speed of Execution of Computational Resources

Kernels: Point kernels

Write once, run everywhere

- C++, OpenCL C, QUALCOMM® Hexagon™ DSP C99 are all “C” code
- Point Kernel captures algorithm at a point in an iteration space
- Point Kernel defines a pure data-parallel programming model and is expressed in C99 (with some restrictions)

```
SYMPHONY_POINT_KERNEL_1D_6(vadd, int i, first, last,
   const float*, a, int, na, const float*, b, int, nb, float*, c, int, nc, {c[i] = a[i] + b[i];});
```

Auto-generate vadd_pk (point kernel)

```
symphony::cpu_kernel(
…
// CPU implementation
)
```

```
#define OCL_KERNEL(name, k) std::string
const name##_string = #k

OCL_KERNEL(vadd_kernel,
__kernel void vadd(__global float* A,
__global float* B, __global float* C) {
  unsigned int i = get_global_id(0);
  C[i] = A[i] + B[i];
});
// GPU implementation
```

```
symphony::hexagon_kerne
l(
…
// DSP implementation
)
```

Kernels: Point kernels

Write once, run everywhere

```
SYMPHONY_POINT_KERNEL_1D_6(vadd, int i, first, last,
   const float*, a, int, na, const float*, b, int, nb, float*, c, int, nc,

        {c[i] = a[i] + b[i];});

auto vadd_pk = symphony::beta::create_point_kernel<vadd_type>();

auto pfor = symphony::beta::pattern::create_pfor_each(vadd_pk, buf_a, buf_b, buf_c);

pfor(range_1d);
```

```
symphony@snapdragon820$top
   CPU0 [================================== ] 97%
   CPU1 [============================       ] 86%
   CPU2 [================================   ] 92%
   CPU3 [====================================] 99%
    GPU [============================       ] 87%
   DSP0 [==========================         ] 80%
   DSP1 [================================== ] 97%
   DSP2 [================================== ] 93%
```

Patterns

A pattern is a commonly occurring combination of control and data accesses

Pattern Name	Description
symphony::pfor_each	Processes the elements of a collection in parallel
symphony::ptransform	Performs a map operation on all elements of a collection, returns a new collection
symphony::pscan	Performs and in-place parallel prefix operation for all elements of a collection
symphony::preduce	Combines all the elements in a collection into one using an associative binary operator
symphony::pdivide_and_conquer	Divides problem into sub-problems, solves them, and merges their solutions in parallel
symphony::pipeline	A sequence of processing stages that can execute concurrently on a data stream

Patterns

Boosting performance using pfor_each with a simple change

```
void foo(vector const& a, vector const& b, vector &c) {
  for(size_t i = 0; i < b.size(); ++i ) {
     c[i] = alpha * a[i] + b[i];
  }
}

void foo(vector const& a, vector const& b, vector &c) {
  symphony::pfor_each(0, b.size(), [&](size_t i) {
    c[i] = alpha * a[i] + b[i];
  });
}
```

Programmer hints: pattern tuner settings

Customize pattern execution by using the Symphony System Manager tuner object

Member Function	Description
set_chunk_size (size_t sz)	Smallest granularity for load balancing. If computational kernel is small (e.g., parallel sum in preduce), set a large chunk size to minimize the synchronization overhead.
set_max_doc (size_t doc)	Max degree of concurrency, default is set to the number of available device threads.
set_static ( )	Use a static chunking algorithm as the parallelization backend.
set_dynamic ( )	Use a dynamic workload balancing algorithm as the parallelization backend.
set_serial ( )	Call the serial version of the computation.
set_shape(pattern::shape shape)	Set shape of workload distribution across range of work-items
set_cpu_load()	Set fraction of workload to schedule on CPU
set_gpu_load()	Set fraction of workload to schedule on GPU
set_dsp_load()	Set fraction of workload to schedule on DSP

Programmer hints: pattern tuner settings

Customize pattern execution by using the Symphony tuner object

```
SYMPHONY_POINT_KERNEL_1D_6(vadd, int i, first, last,
   const float*, a, int, na, const float*, b, int, nb, float*, c, int, nc,

        {c[i] = a[i] + b[i];});


auto vadd_pk = symphony::beta::create_point_kernel<vadd_type>();

auto pfor = symphony::beta::pattern::create_pfor_each(vadd_pk, buf_a, buf_b, buf_c);

pfor(range_1d, symphony::pattern::tuner().set_cpu_load(20).set_gpu_load(70).set_dsp_load(10));
```

- 20% of all iterations in range_1d go to the CPU
- 70% iterations go to the GPU, and
- 10% iterations go to the DSP

Data: buffers for heterogeneous computing

Uniform access from host code and across tasks

Create a buffer with 100 floats

```
auto b = symphony::create_buffer<float>(100);
```

Access directly from host

```
for(int i = 0; i < b.size(); i++)
   b[i] = i;
```

Access within tasks across devices

```
auto cpu_task = symphony::launch(cpu_kernel, b);

auto gpu_task = symphony::launch(gpu_kernel, range, b);

auto hexagon_task = symphony::launch(hexagon_kernel, b);
```

Homogeneous task graph - CPU

1. Create buffers

```
auto buf_a = symphony::create_buffer<float>(1024);
auto buf_b = symphony::create_buffer<float>(buf_a->size());
```

2. Initialize buffers

```
auto ck = [](symphony::buffer_ptr<float> a, symphony::buffer_ptr<float> b){
  for (size_t i = 0; i < a->size(); ++i) {
    a[i] = i;
    b[i] = a->size() - i;
  }
};
```

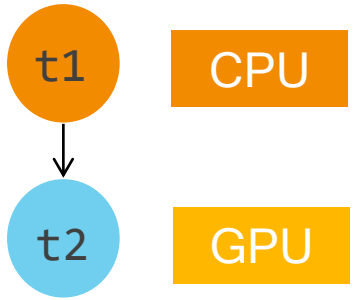
3. Launch task

```
auto init_task = symphony::launch(ck, buf_a, buf_b);
```

4. Wait for completion of task

```
init_task->wait_for();
```

Heterogeneous task graph - CPU + GPU



1. Create buffers

```
auto buf_a = symphony::create_buffer<float>(1024);
auto buf_b = symphony::create_buffer<float>(buf_a->size());
```

2. Initialize buffers

```
auto ck = [](symphony::buffer_ptr<float> a, symphony::buffer_ptr<float> b){
  for (size_t i = 0; i < a->size(); ++i) {
    a[i] = i;
    b[i] = a->size() - i;
  }
};
```

3. Create GPU Kernel; add vectors on GPU ; buf_c = buf_a + buf_b

```
auto buf_c = symphony::create_buffer<float>(buf_a->size());
auto gk = symphony::create_gpu_kernel<…>(vadd_kernel_string, “vadd”);
```

4. Create dependency and launch work

```
auto vadd_task = symphony::create_task(gk, symphony::range<1>(1024), buf_a, buf_b, buf_c);
auto init_task = symphony::launch(ck, buf_a, buf_b);
init_task->then(vadd_task);
vadd_task->launch();
vadd_task->wait_for();
```

Comparison with OpenCL (Vector Add)

```
foo(float *a, float *b, float *c, int size) {

  cl::Buffer buf_a(…, UHP, size, a);
  cl::Buffer buf_b(…, UHP, size, b);
  cl::Buffer buf_c(…, UHP, size, c);

  queue.enqueueWriteBuffer(buf_a, …, a);
  queue.enqueueWriteBuffer(buf_b, …, b);
  kernel.setArg(0, buf_a);
  kernel.setArg(1, buf_b);
  kernel.setArg(2, buf_c);
  kernel.setArg(3, size);
  queue.enqueueNDRangeKernel(kernel,
             cl::NullRange,
             cl::NDRange(size),
             cl::NullRange);


  queue.finish();
  queue.enqueueReadBuffer(buf_c, …, c);
}
```

Launch

Wait for results

```
foo(float *a, float *b, float *c, int size) {

  auto buf_a = symphony::create_buffer(a, size, false);
  auto buf_b = symphony::create_buffer(b, size, false);
  auto buf_c = symphony::create_buffer(c, size, false);

  auto t = symphony::launch(kernel,
                symphony::range<1>(size),
                buf_a, buf_b, buf_c, size);


  t->wait_for();
  buf_c.ro_sync();
}
```

Power Management

Symphony Power API

CPU/GPU Core & Frequency Control

- APIs available to programmer to help make these decisions runtime
- Static Power Management (CPU and GPU)
  - User chooses amongst 5 predefined power modes

```
symphony::power::request_mode(mode, duration, device_set)
```

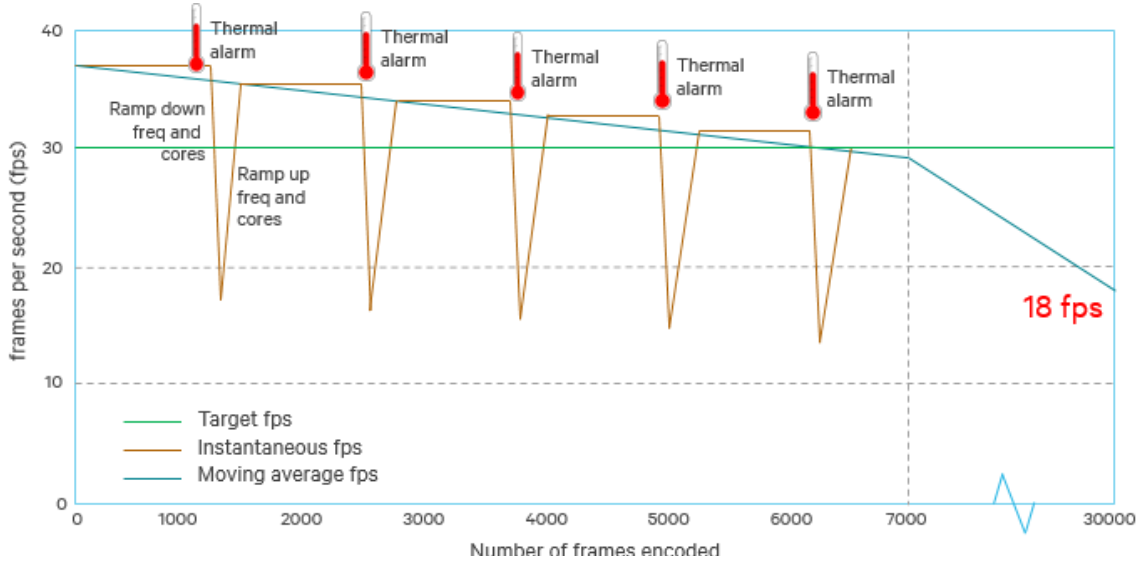
- Dynamic Power Management:
  - Minimize energy consumption while preserving user-defined Quality of Service
  - Works well with “main loop” based applications (games, streams, …)

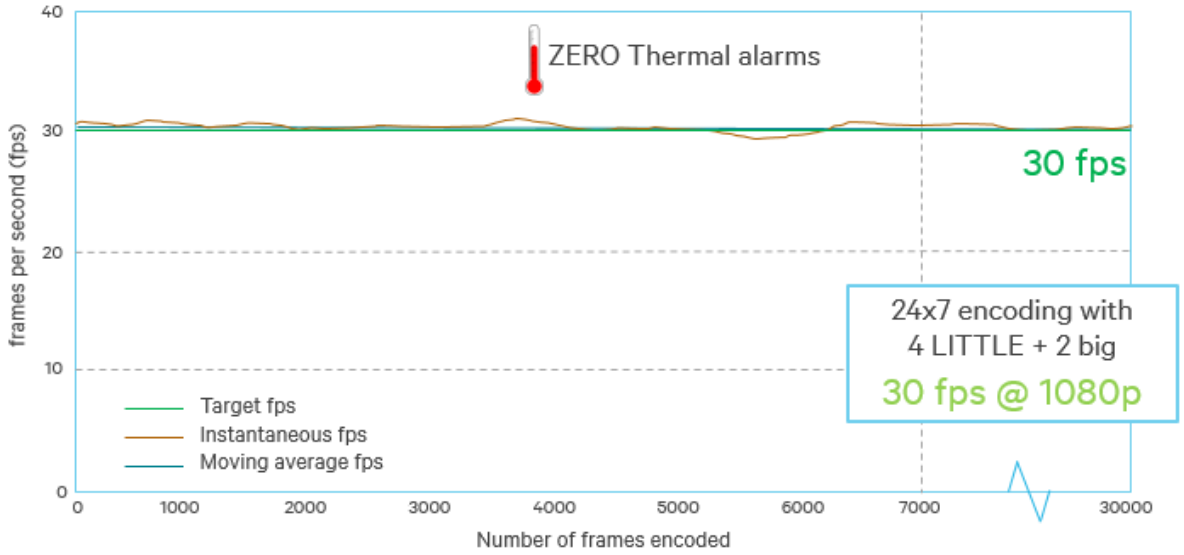
```
symphony::power::set_goal(desired, tolerance) // Before the main loop
symphony::power::regulate(measured)           // Within the main loop
symphony::power::clear_goal()                 // After the main loop
```

Real world use cases

Virtually always-on HD security camera

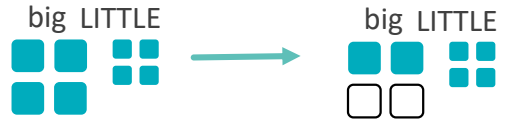
Custom software-based H264 video encoder needing to operate 24x7 at 30fps





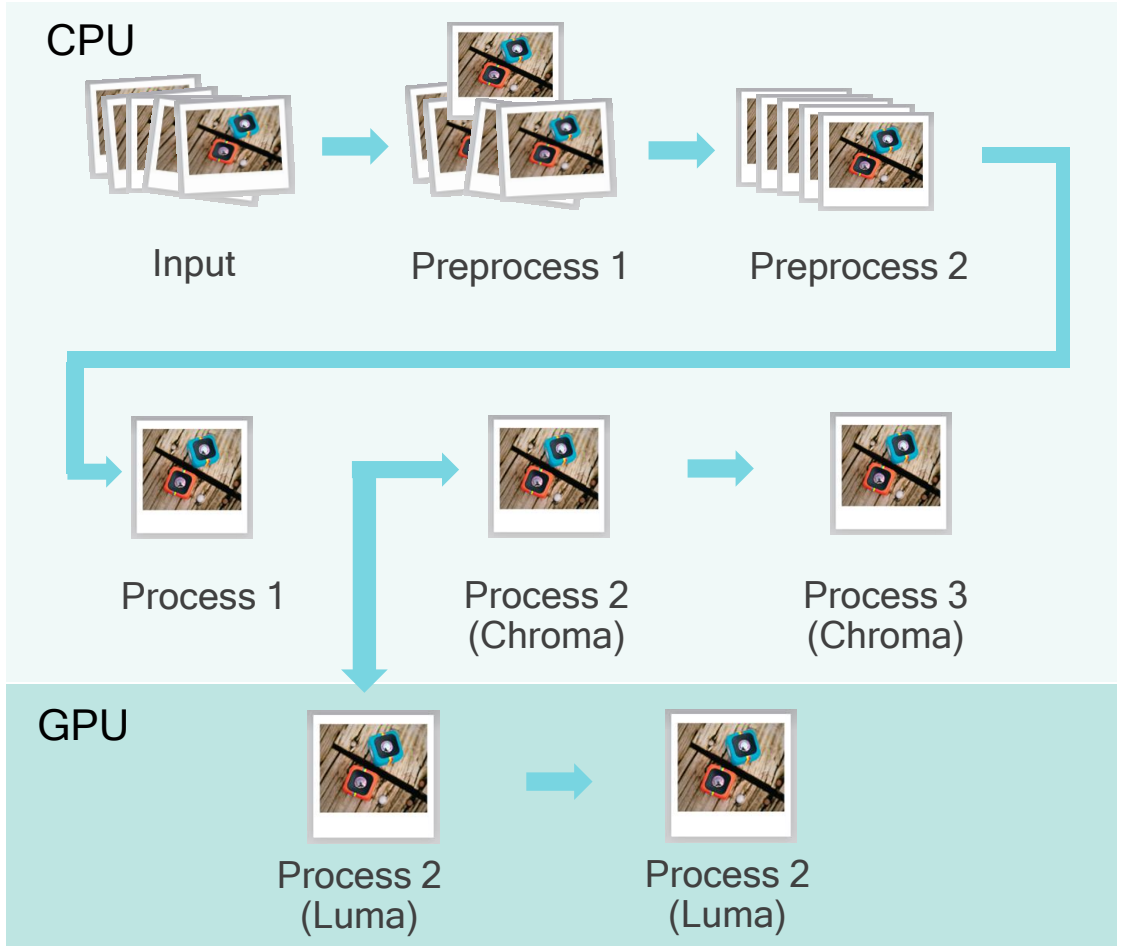
	Before Symphony	After Symphony
Min. frame rate	<18 fps	**30 fps**
Cores used	4 LITTLE + 4 big	**4 LITTLE + 2 big**
Thermal alarms	1 every ~1000 frames	**None**
Processor throttling	Yes	**No**

- Few lines of code change to use specific CPU cores using Symphony System Manager SDK
- Simple tuning process using Symphony APIs
- Platform: Snapdragon Platform (810)



Low light camera

Split data path so CPU and GPU can process images at the same time



	Before Symphony	After Symphony
Processing time	>8.0s	1.3s
Max. Power	3.6W	2.5W

- Few lines of code change to offload Luma processing to GPU
- 6.1x Performance gain
- 72% Energy Savings
- Simple tuning process using Symphony APIs
- Platform: Snapdragon Platform (808)

Bilateral Filter

Edge Preserving Low-Pass Filter – Compute Intensive Image Processing Algorithm

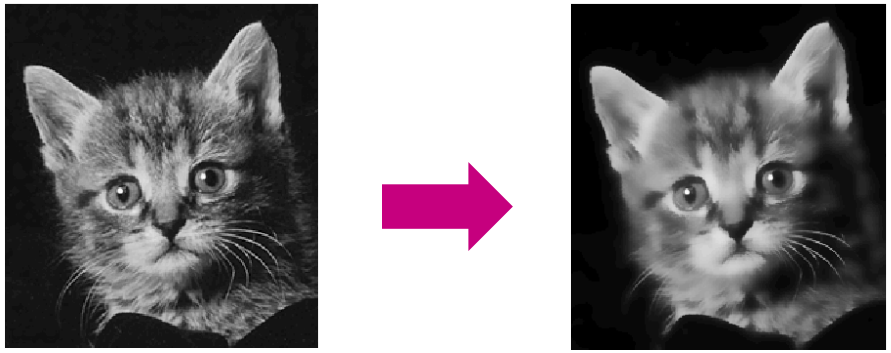
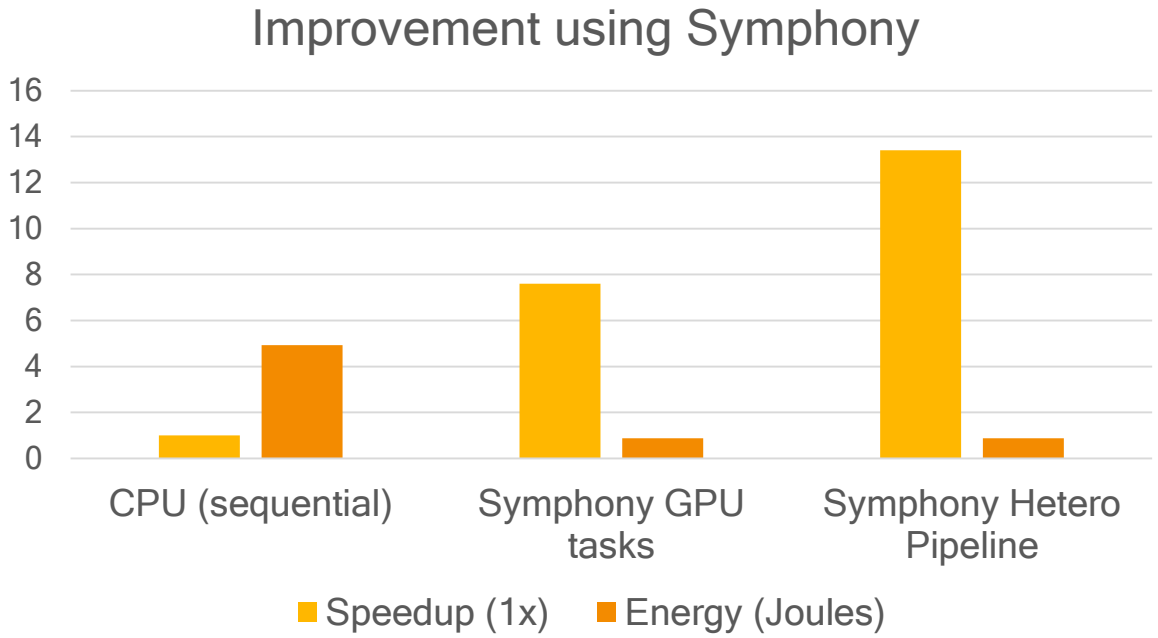
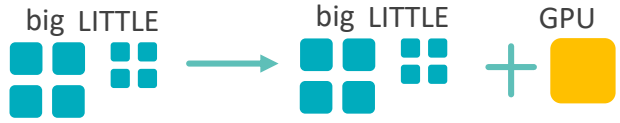


Image source: Bilateral Filtering for Gray and Color Images whitepaper; 1998 IEEE International Conference on Computer Vision, Bombay, India

	Without Symphony	Using Symphony	
		GPU Offload	Pipeline Pattern
Speedup	-	7.6x	13.4x
Energy	4.92J	0.87J	0.87J

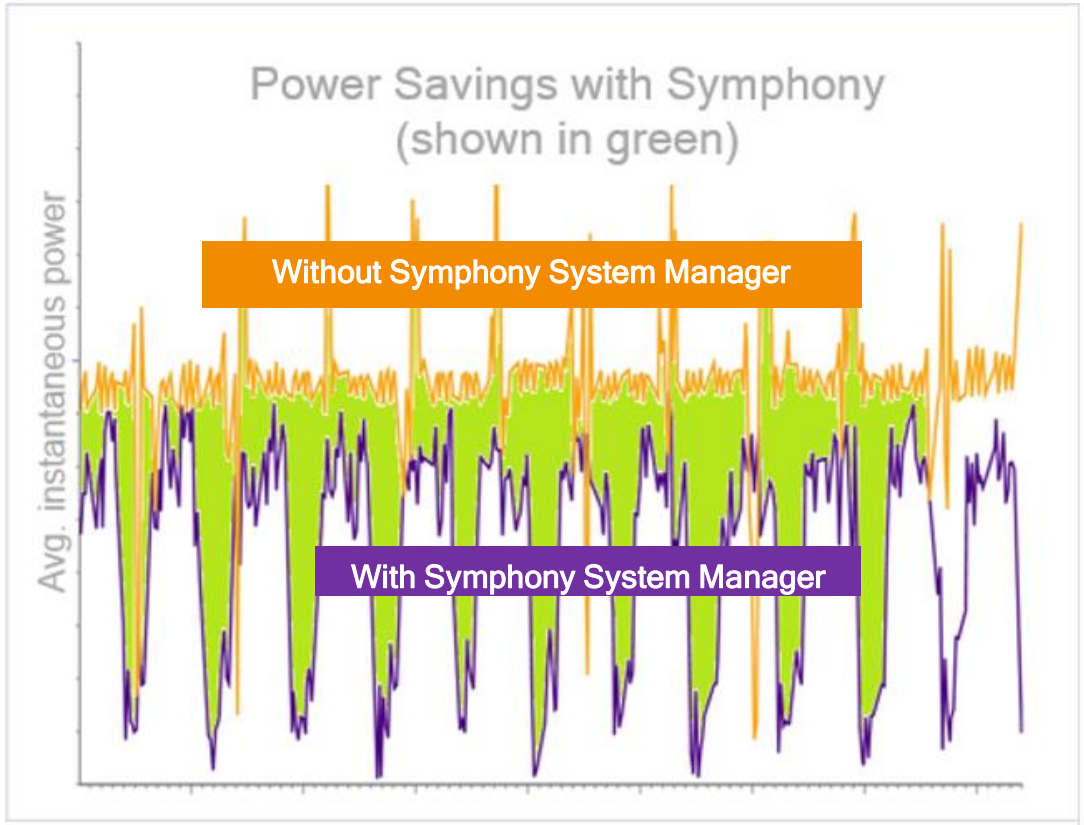


- Significant performance and energy savings
  - Symphony heterogeneous pipeline pattern - Offload specific stages of a pipeline to GPU
  - Symphony texture object
- Platform: Snapdragon Platform (821)



4K video image stabilization for smartphone camera

Dividing workload into fine-grain parallel tasks and using Symphony System Manager power APIs



	Before Symphony System Manager	After Symphony System Manager
Video processing time	63ms	18-20ms
Performance improvement		~65%
Power savings		40%

- Few lines of code change to use
  - Symphony parallel for each pattern
  - Symphony dynamic power APIs
- Simple tuning process using Symphony APIs
- Platform: Snapdragon Platform (800)

Qualcomm Developer Network

https://developer.qualcomm.com/

Download Symphony System Manager SDK

- Symphony Library

- User Guide

- Code Samples

Thank you

Follow us on:

For more information, visit us at:
www.qualcomm.com & www.qualcomm.com/blog