ONNX and Python to C++: State-of-the-Art Graph Compilation

Nigel Drego
Co-founder & CTO
Quadric

Quadric

About Quadric

Pure play Semiconductor IP Licensing

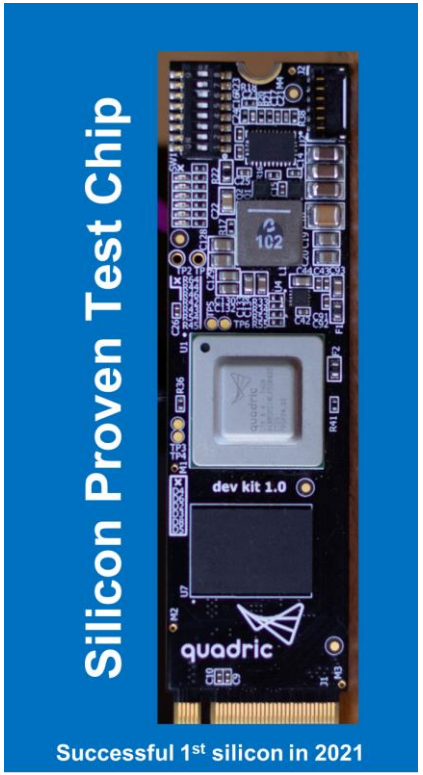
- **Processor IP & Software Tools**

Edge / device AI/ML Inference + DSP processing

HQ: Silicon Valley – Burlingame CA
Venture Capital funded

Mar 2021:	First working silicon
May 2023:	First IP delivery, DevStudio Online
Dec 2024:	First multicore cluster delivery

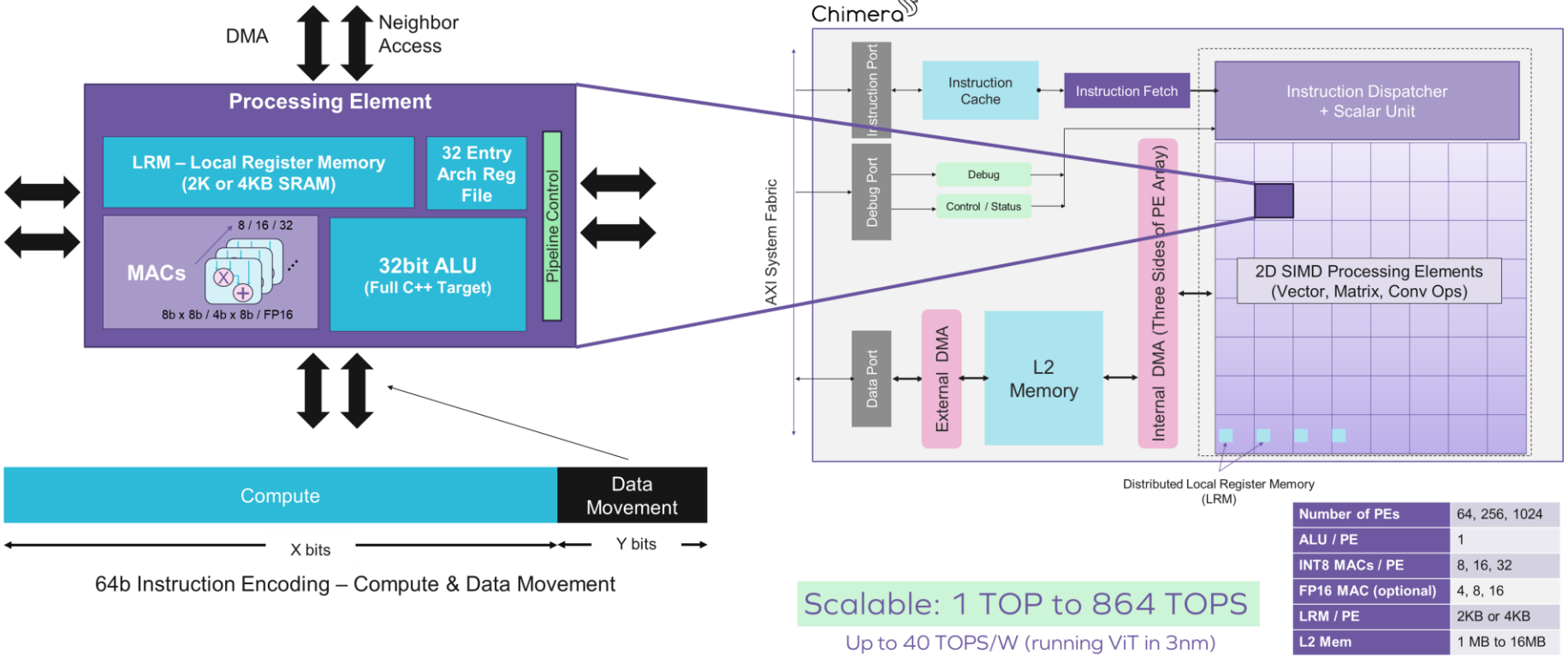
Patents: 28+ Granted



Quadric

Chimera GPNPU Block Diagram

A Hybrid Between a CPU and a Systolic Array



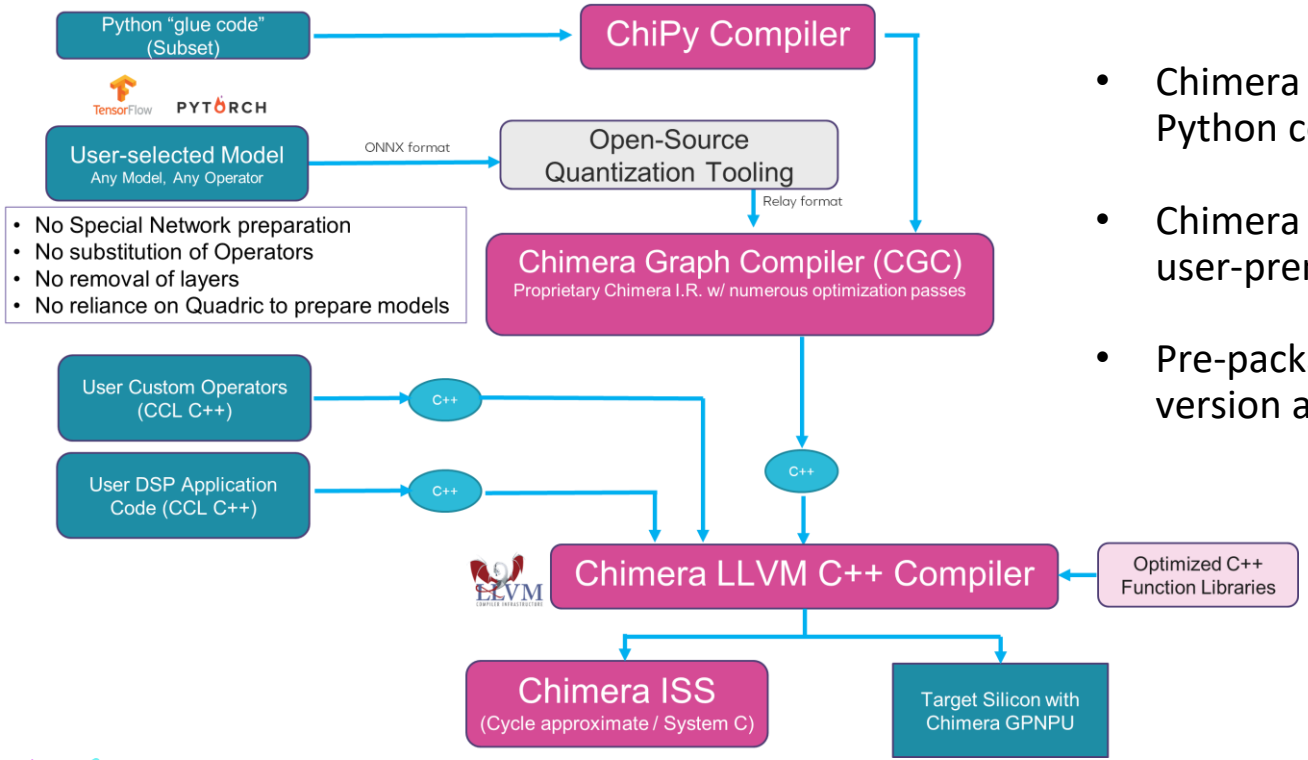
Number of PEs	64, 256, 1024
ALU / PE	1
INT8 MACs / PE	8, 16, 32
FP16 MAC (optional)	4, 8, 16
LRM / PE	2KB or 4KB
L2 Mem	1 MB to 16MB

Scalable: 1 TOP to 864 TOPS

Up to 40 TOPS/W (running ViT in 3nm)

Quadric

Quadric SDK Overview



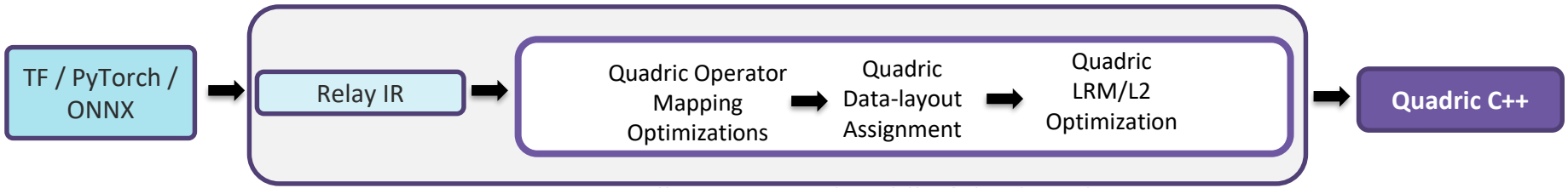
- Chimera runs Graph Code, C++ and Python code
- Chimera SDK runs on user-premises or user-cloud
- Pre-packaged Docker container version available

Chimera Graph Compiler (CGC)

- **Import**
  - Graph analysis (operator compatibility, quantization completeness, custom operator ID)
  - Graph simplification / canonicalization
  - Constant propagation
- **Graph optimization**
  - Quantization scheduling (with activation range analysis)
  - Graph serialization
  - Fuse operators and select implementations to minimize data movement costs
- **Memory optimization**
  - Tensor format layout analysis (both L2 and LRM)
  - Intelligent weight prefetching

Chimera Graph Compiler - High Level Architecture

- **Chimera Graph Compiler (CGC) leverages the TVM project framework**
- **Architecture-aware Quadric specific "middle-end"**
  - Injects Quadric-specific data layouts early in the lowering flow
  - Hardware aware mapping Op selection
  - Memory scheduling to maximize L2MEM & LRM utilization (lowers power)
- **Direct control flow mapping from Relay -> C++**
  - Create highly optimized code for GPNPU

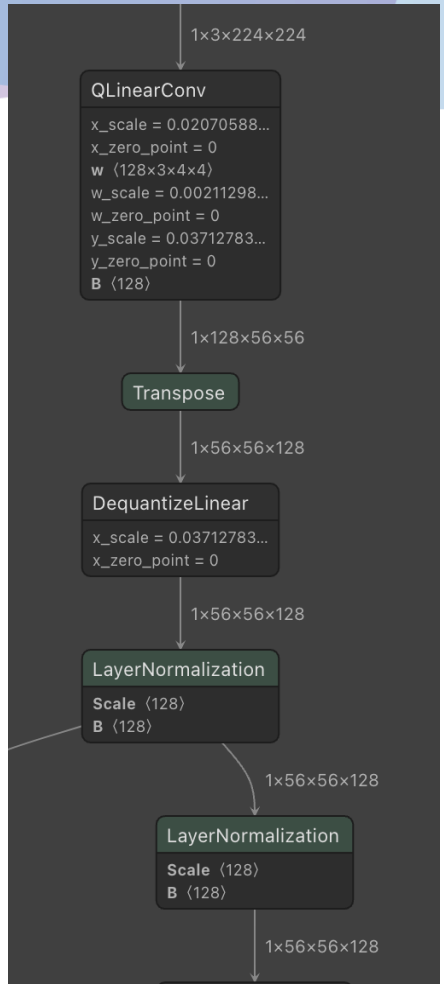


Fully customized middle & back-end in TVM

Layout Optimization

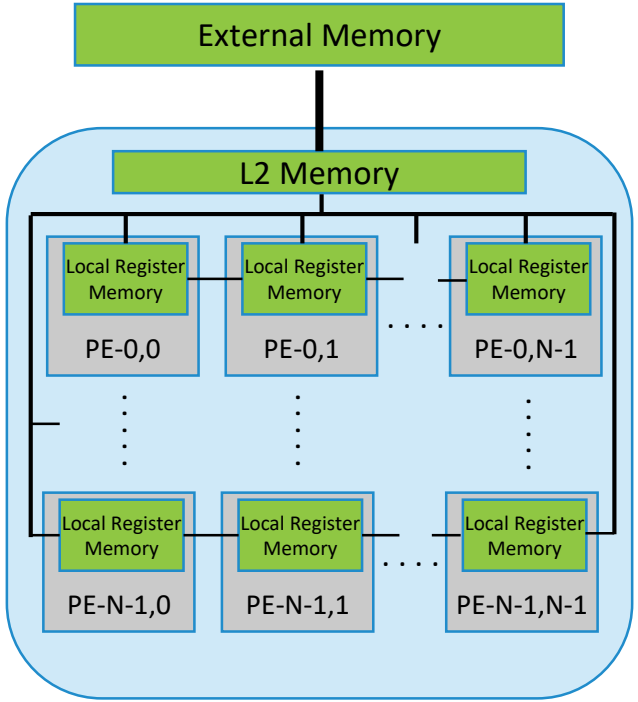
- **Transform non-NCHW formats to NCHW where possible**
  - Many shape transformation ops can be eliminated, even in Multi-Head Self Attention
- **Example: LayerNorm operating over channel dimension**
- **Solution: Analyze dimension dependencies, then sink transpose across ops**
  - Can handle arbitrary sequences of shape transformation operations

NCHW = **N**umber of samples, **C**hannels, **H**eight, **W**idth



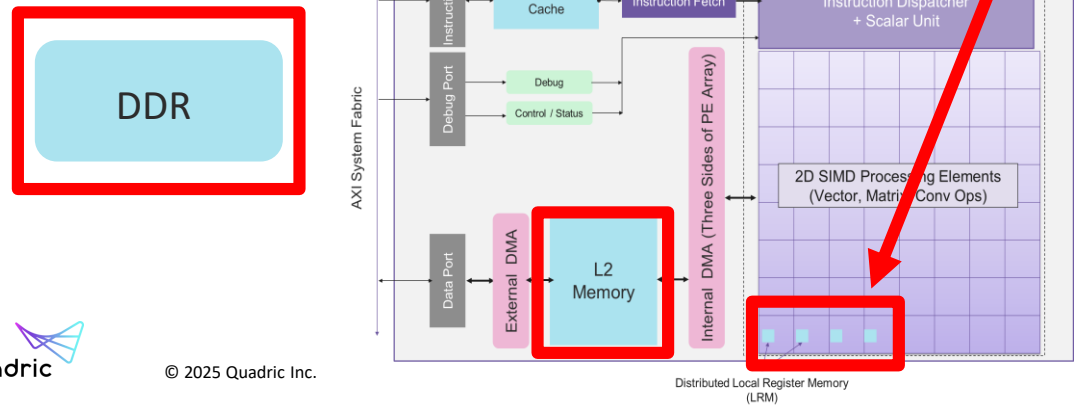
Layout Optimization

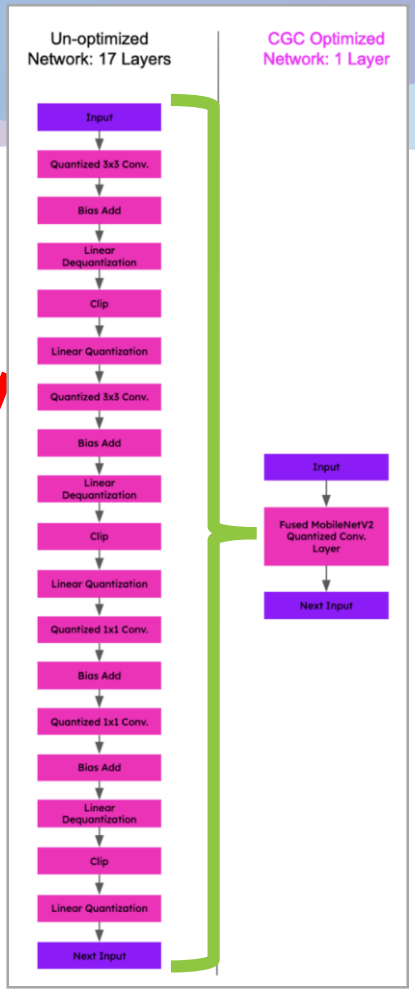
- **Process graph in an edge-wise manner:**
  - An edge has a producing operator, i.e., Conv and a consuming operator, i.e., ReLU
- **Utilize information about op's consumed layout(s) and produced layout(s):**
  - "Layout" refers to data placement in PE-array, not L2 or external memory
  - Op can have multiple implementations: different consumed / produced layouts
  - Enumerate producer/consumer layout pairs and select pairs with lowest cost



Deep LRM-based Fusion

- **Other NPUs refer to "fusion" as keeping data on-chip**
- **Chimera fusion keeps data within PE's LRM**
  - Element-wise ops fused by keeping in registers
  - Other ops via LRM-based fusion
    - Convolutions
    - MHSA
    - Normalizations (e.g., LayerNorm, GroupNorm, etc)





Fusion Example: [Blog link]

Fusion in Local Memory (FILM) **saves power, increases performance**

CGC Generated Code Example

Architected to Run a Single Code Stream For Scalar, Vector & Matrix Computations

Scalar Variables

Flow/DMA APIs

Vector/Matrix Variables

Fused Elementwise
Any activation can be coded up
De-quantize math

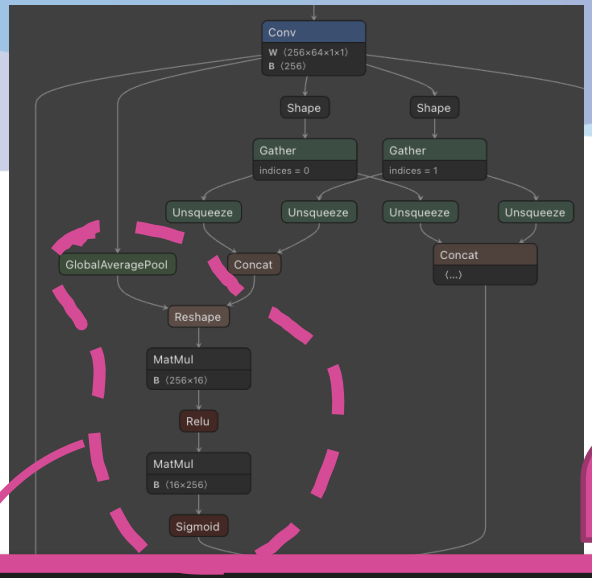
Fused Convolution

```
for (int32_t tb_y1 = 0; tb_y1 < 14; ++tb_y1) {
 for (int32_t tb_x1 = 0; tb_x1 < 14; ++tb_x1) {
   container::NDArray<qVar_t<int8_t>, 27> ocm_tensor_0_rf;
   ocm_tensor_0_flow_1.read(ocm_tensor_0_rf);

   container::NDArray<qVar_t<int8_t>, 32> T_qlinear_conv2d_rf;
   BroadcastFlow<TensorAccessor<MinRoiDescriptor<OcmTensor<std::int8_t, 1, 1, 1, 1280>,... >

   for (int32_t ch1 = 0; ch1 < 32; ++ch1) {
    qVar_t<int32_t> _0 = nn::convTileBlockInt8<std::int32_t, 27, 1, 0, false>(ocm_tensor_0_rf);
    qVar_t<int32_t> _11 = qBroadcast<0, std::int32_t, BroadcastAction::POP>;
    T_qlinear_conv2d_rf[(ch1)] =
math::min(math::max(cgc::fxRoundPosInf<23>(math::fxMul<31>(math::min(math::max(math::fxMul<2>((math::min(math::max(cgc::fxRoundP
osInf<2>(math::fxMul<29>((_0 + _11), 4679030)), -128), 127)), 58507024), 0, 3.2212255e+09f, 1231605867)), -128), 127));
   }

   container::NDArray<qVar_t<int8_t>, 32> T_qlinear_conv2d_rf1;
   for (int32_t ch2 = 0; ch2 < 32; ++ch2) {
    qVar_t<int32_t> _2 = nn::groupwiseConvTileBlockInt8<std::int32_t, 1, 3, 1, 0, false>(T_qlinear_conv2d_rf,ch2);
    qVar_t<int32_t> _3 = qBroadcast<0, std::int32_t, BroadcastAction::POP>;
    T_qlinear_conv2d_rf1[(ch2)] = math::min(math::max(cgc::fxRoundPosInf<22>.....
```

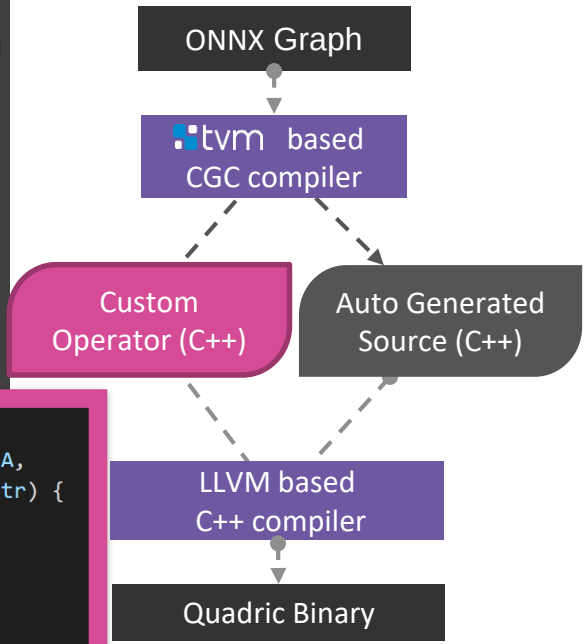
Example of auto generated Convolution code running on Chimera

Custom Operator Support

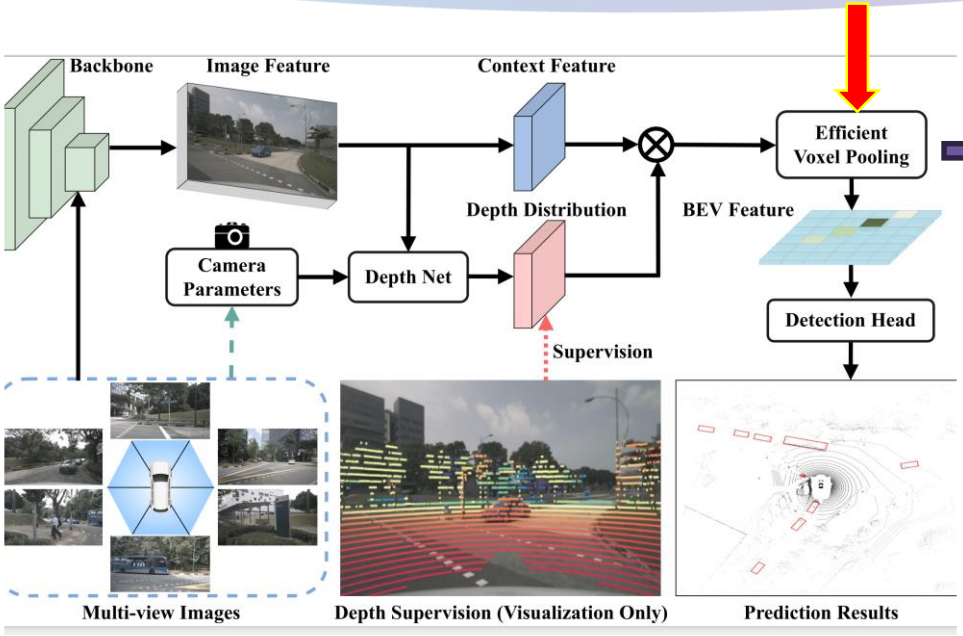
- **CGC Compiler Supports Hybrid Lowering**
  - Automatic code generation for whole/partial graph
- **Custom Implementation of nodes/subgraphs**
  - e.g., NMS, proprietary layers, custom operators




```
template <typename OcmWeightsShape>
void CustomOperator(DdrInTensorShape::ptrType ddrInPtrA,
                    DdrOutTensorShape::ptrType ddrOutPtr) {
  /* Operator implementation*/
  .
  .
  .
  .
  .
  .
  .

}
```



BEVDepth – The Key Component Is in CUDA



Mixed ONNX + CUDA Code

BEVDepth is a hybrid algorithm that includes common neural network graph constructs as well as DSP algorithms for voxel-pooling, implemented in CUDA.

Voxel-pooling is ~60% of the algorithmic compute cost

→ How & where you run Voxel Pooling will determine overall BEVdepth performance efficiency

Arxiv paper	BEVDepth on Dual QC-Ultra
Original Repo	GitHub Repo
Network Config	NuScene Configurations
Total Ops (G)	700
Parameters (M)	76.25
Input Size	6 x 3 x 256 x 704
Inference Rate (FPS) (with Voxel Pooling)	26.30

Dual Chimera Ultra Config – 64 TOPS →

BEVDepth – Simple Port from CUDA

```
__syncthreads();

for (int i = tidy;
     i < THREADS_PER_BLOCK && block_sample_idx + i < total_samples;
     i += THREADS_BLOCK_Y) {
  const int sample_x = geom_xyz_shared[i * 3 + 0];
  const int sample_y = geom_xyz_shared[i * 3 + 1];
  const int sample_z = geom_xyz_shared[i * 3 + 2];
  if (sample_x < 0 || sample_x >= num_voxel_x || sample_y < 0 ||
      sample_y >= num_voxel_y || sample_z < 0 || sample_z >= num_voxel_z) {
    continue;
  }
  const int batch_idx = (block_sample_idx + i) / num_points;
  for (int j = tidx; j < num_channels; j += THREADS_BLOCK_X) {
    atomicAdd(&output_features[(batch_idx * num_voxel_y * num_voxel_x +
                               sample_y * num_voxel_x + sample_x) *
                                  num_channels +
                              j],
              input_features[(block_sample_idx + i) * num_channels + j]);
  }
}
```

FETCHING GEOM_XYZ

CUDA Implementation

```
for(std::uint32_t mroiIdx = 0; mroiIdx < featuresReadFlow.numMROIs(); mroiIdx++) {
  auto                                        qFeatures = featuresReadFlow.read();
  auto                                        geomBuf   = geomInElsFlow.template read<false>();
  container::NDArray<qVar_t<std::int16_t>, 3>  qGeomXYZ;
  container::NDArray<qVar_t<std::uint32_t>, 3> qAddrs;
  rau::config(geomBuf);
  for(std::int32_t dim = 0; dim < 3; dim++) {
    qVar_t<std::int32_t> qGeomIdx = qRow<>;
    qAddrs[dim]                   = rau::computeAddr<decltype(geomBuf)>(0, 0, qGeomIdx, dim);
  }
  rau::load::tiles(qAddrs, qGeomXYZ, geomBuf);

  qVar_t<std::int16_t> qX = qGeomXYZ[0];
  qVar_t<std::int16_t> qY = qGeomXYZ[1];
  qVar_t<std::int16_t> qZ = qGeomXYZ[2];
  // make sure that if qZ is out of bounds, we do not write.
  qVar_t<std::int16_t> qBaseChnIdx = qZ * (OutShape::NUM_CHN + core_array::coreDim) + qCol<>;

  voxelAtomicAdd(qFeatures, qBaseChnIdx, qY, qX, outGrid);
}
```

FETCHING GEOM_XYZ

Chimera Implementation

Custom Operator: Voxel Pooling

- **Program at PE-level, awareness of entire PE-array**
  - Similar to CUDA's thread/warp
- **Data access:**
  - Flows, with automatic double-buffering
  - Random-access

```
void voxelPooling(…) {
  …
  auto featuresReadFlow = createGeneralReadFlow (imgFeatures, ocmMemAlloc);
 ExtFlow<FlowType::Read, ..> geomInElsFlow{geomXYZ, ocmMemAlloc};
 for(std::uint32_t chnOffset = 0; chnOffset < numChannels; chnOffset += core_array::coreDim) {
  container::NDArray<qVar_t<std::int32_t>, ...> qOut = {0};
  for(std::uint32_t row = 0; row < numPoints; row += mroiHeight) {
   auto qFeatures = featuresReadFlow.read();
   auto geomBuf   = geomInElsFlow.template read<false>();

   rau::config(geomBuf);
   for(std::int32_t mroiRowIdx = 0; mroiRowIdx < numTilesPerMroi; mroiRowIdx++) {
    container::NDArray<qVar_t<GeomT>, 3>       qGeomXYZ;
    container::NDArray<qVar_t<std::uint32_t>, 3> qAddrs;
      …
     rau::load::tiles(qAddrs, qGeomXYZ, geomBuf);
     qVar_t<std::int16_t> qX = qGeomXYZ[0];
     qVar_t<std::int16_t> qY = qGeomXYZ[1];
     qVar_t<std::int16_t> qZ = qGeomXYZ[2];
     qVar_t<std::int16_t> qBaseChnIdx = qZ * (OutShape::NUM_CHN + core_array::coreDim) + chnOffset + qCol<>;
     voxelAtomicAdd<OutShape>(qFeatures[mroiRowIdx], qBaseChnIdx, qY, qX,
qOut);
    }
   }
   rau::config(outGrid);
   qVar_t<std::uint32_t> qChnIdx = chnOffset + qCol<>;
   for(std::size_t i = 0; i < qOut.size(); i++) {
    qVar_t<std::uint32_t> qRowIdx = i / OutShape::NUM_TILES_PER_COL;
    qVar_t<std::uint32_t> qColIdx = (i % OutShape::NUM_TILES_PER_COL) * core_array::coreDim + qRow<>;
     rau::store::oneTile(0, qChnIdx, qRowIdx, qColIdx, qOut[i], outGrid);
    }
   }
  }
```

Key Lines of Code in Large Font

Voxel Pooling: Atomic Add

- **Atomic add implemented *without* atomic primitives**
- **All data exchanged within PE-array**
- **Huge performance gain relative to CUDA**

```
void voxelAtomicAdd(...) {
  static_assert(sizeof(InT) == 1, "Input type must be 1 byte type.");
  qVar_t<std::int8_t> qValid = qArrayCore & (qColIdx >= 0) & (qColIdx < OcmShape::NUM_COLS) & (qRowIdx >= 0) &
                               (qRowIdx < OcmShape::NUM_ROWS) & (qBaseChnIdx >= 0) &
                               (qBaseChnIdx < OcmShape::NUM_CHN);
  qVar_t<std::int8_t>   qDist = std::numeric_limits<std::int8_t>::min();
  qVar_t<std::uint16_t> qChn  = 0;
  if(qValid) {
   qVar_t<std::int16_t> qLinearizedIdx = qRowIdx * 128 + qColIdx;
   qDist                = qLinearizedIdx % core_array::coreDim - qRow<>;
   qChn                 = qLinearizedIdx / core_array::coreDim;
  }

  container::NDArray<qVar_t<std::int8_t>, 2> qUnpacked1 = {qDist, qVal};
  auto qPacked1 = qUnpacked1.template asReinterpretCast<qVar_t<std::uint16_t>, 1>();
  container::NDArray<qVar_t<std::uint16_t>, 2> qUnpacked = {qPacked1[0], qChn};
  auto qPacked = qUnpacked.template asReinterpretCast<qVar_t<std::uint32_t>, 1>();

  qOut[qChn] += qVal * (qDist == 0);
  setCleanStateForNeighborMovement<InT>(0);
  qNorthSouthBcast<std::uint32_t> = qPacked[0];
#pragma unroll
    for(std::int32_t move = 1; move < core_array::coreDim; move++, rot180()) {
      container::NDArray<qVar_t<std::uint32_t>, 1> qPackedSouth = {qSouth<std::uint32_t>};
      auto qUnpacked1South = qPackedSouth.template asReinterpretCast<qVar_t<std::uint16_t>, 2>();
      qVar_t<std::uint16_t> qChnSouth  = qUnpacked1South[1];
      auto qUnpackedSouth =
       (qUnpacked1South.template slice<1>(0)).template asReinterpretCast<qVar_t<std::int8_t>, 2>();
      qOut[qChnSouth] += qUnpackedSouth[1] * (qUnpackedSouth[0] == -move);

      container::NDArray<qVar_t<std::uint32_t>, 1> qPackedNorth = {qNorth<std::uint32_t>};
      auto qUnpacked1North = qPackedNorth.template asReinterpretCast<qVar_t<std::uint16_t>, 2>();
      qVar_t<std::uint16_t> qChnNorth      = qUnpacked1North[1];
      auto qUnpackedNorth = (qUnpacked1North.template slice<1>(0)).template
asReinterpretCast<qVar_t<std::int8_t>, 2>();
      qOut[qChnNorth] += qUnpackedNorth[1] * (qUnpackedNorth[0] == move);
    }
  }
```

Combine Model + Custom Operator w/ Python

```
@chipy.ccl_custom_op(ccl_func_name="voxelPooling")
def voxel_pooling(geom_xyz, img_feats):
  # Add python implementation to check CCL implementation against pass

@chipy.func()
def full_model(
    model_path,
    imgs,
    points,
    ida_mats
):
    geom_xyz, feats = chipy.infer_onnx(model_path,
                                       ida_mats,
                                       imgs,
                                       points)
    return voxel_pooling(geom_xyz, feats)
```

Voxel Pooling: Chimera Performance

Twice the Performance
1/8th of the Bandwidth Required
10 - 80x Lower Power*

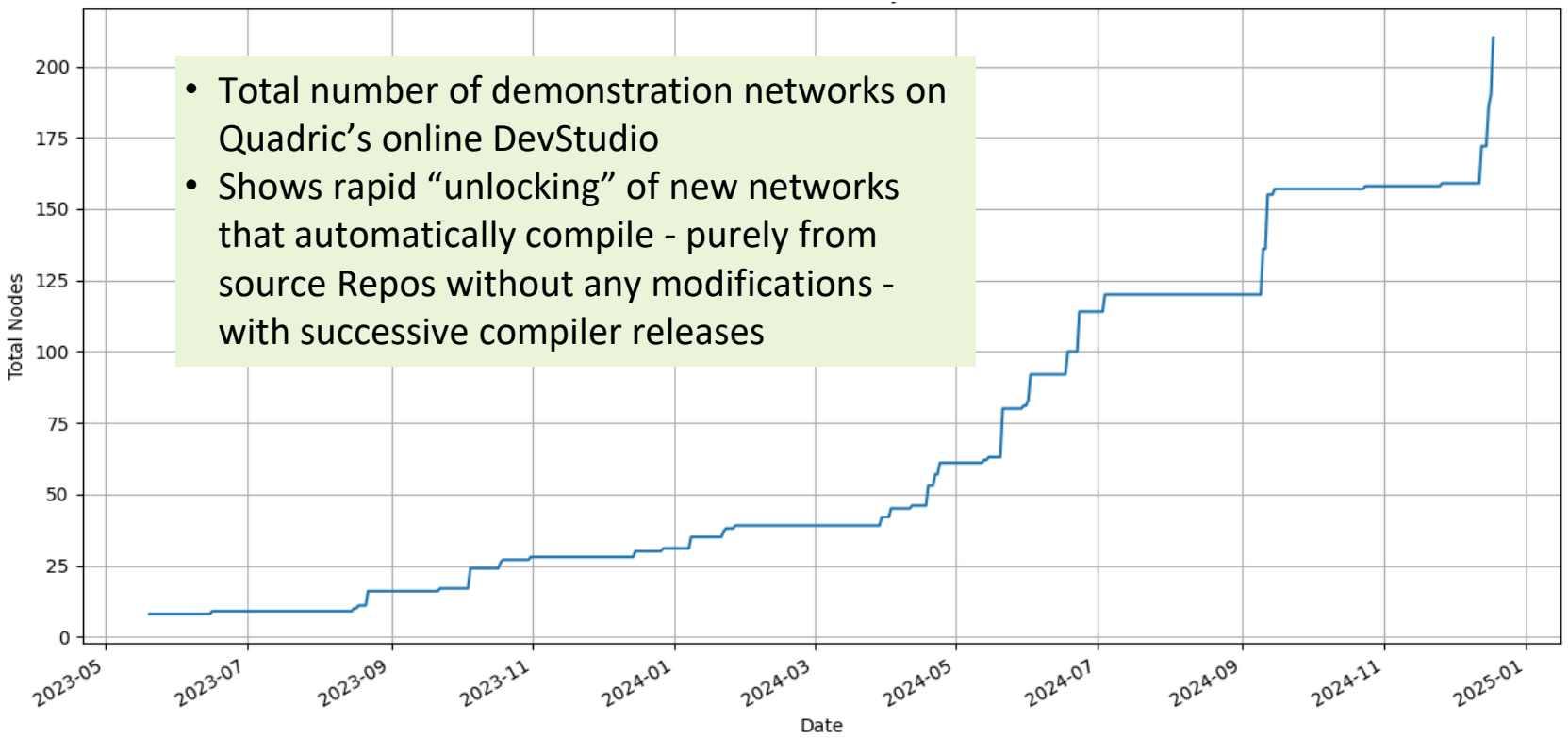
	Nvidia RTX 3090 (Full Chip)	Quadric QC-Ultra (quad-core IP)
Coding Language	CUDA C++	CCL C++
Clock Freq	1.4 GHz	1.4 GHz
ALU Capacity (ALU Count)	41,984 (from 10496 shader cores)	4096
Memory Bandwidth	936 GB/s	128 GB/s (32GB per core)
Voxel Pooling Performance (1 iteration)	23 ms	11 ms

Full Detailed Tutorial on DevStudio with complete Voxel Pooling source code

Voxel Pooling using data sizes of: [geom_xyz 6, 112, 16, 44, 3]; [img_feat 6, 112, 16, 44, 80]; [out_grid 80, 128, 128]
* RTX3090 card hosts a GA102 GPU – 8 nm process, 628 mm^2, 450 Watts (max). Voxel Pooling power difficult to measure directly on the RTX3090. Quadric power estimates for cores only – not including system memory I/F.

Benefit of Pure Compiler Approach

Unlocking New Networks Far Faster Than Manual Porting



- Total number of demonstration networks on Quadric's online DevStudio
- Shows rapid "unlocking" of new networks that automatically compile - purely from source Repos without any modifications - with successive compiler releases

Release Over Release Performance Upgrades

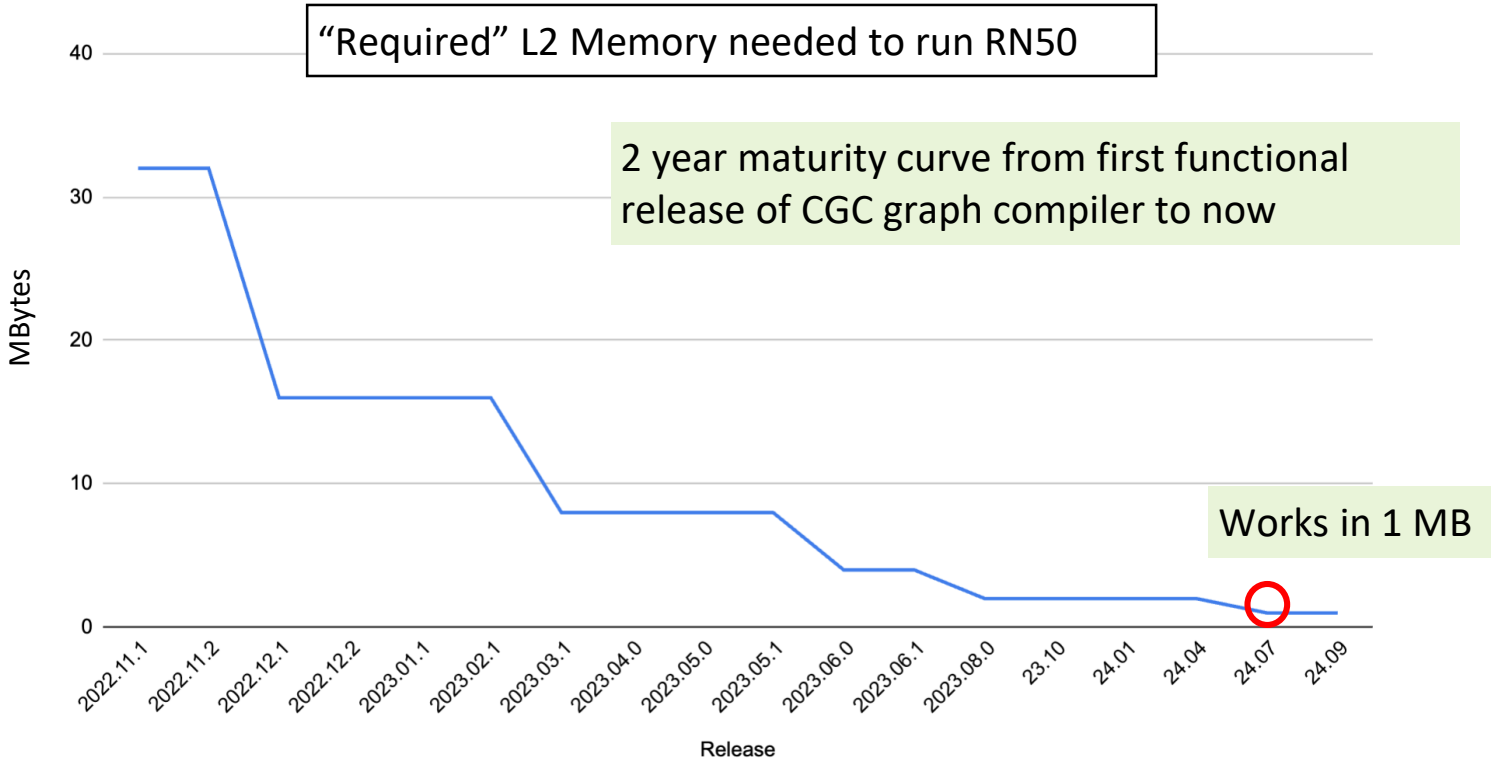
Code generation optimizations continue to unlock large performance gains, compounding over time

23.08	23.10	24.01	24.04	24.07	24.09	24.12
						
Up to 50%	Up to 33%	Up to 42%	Up to 22%	Up to 76%	Up to 106%	Up to 68%

- Percentages represent the largest release-over-release performance gains for existing networks (different networks each comparison period).
- All networks auto-compiled from original FP32 source repos – no network mods, no layer changes, no manual preparation.

CGC Compiler Maturity

Memory Management Refinement



Conclusions

- Pure Graph Compiler approach to AI / ML inference yields far better results than manual porting / optimization
  - Developer productivity → most networks “just compile” out of the box
  - Compiler maturity → “free” improvements each release into the future

- C++ programmability → runs more than just “graph code”

- Programming in C++, Graph and Python code makes embedded AI inference almost as easy as datacenter AI training & inference

Quadric

Resources @ EVS'25

Visit us on the web

https://www.quadric.io

Try Quadric DevStudio

https://studio.quadric.io/

2025 Embedded Vision Summit

Please visit us at Booth #821

Quadric

Thank You

Quadric