https://doi.org/10.2352/ISSN.2470-1173.2017.14.HVEI-136

GPU-accelerated vision modeling with the HPE cognitive computing toolkit

Benjamin Chandler; Los Altos, CA

Abstract

The HPE Cognitive Computing Toolkit (CCT) is an open-source modeling platform backed by Hewlett Packard Enterprise. CCT provides a domain-specific language designed for problems like vision modeling and deep learning. The CCT platform compiles programs written in this language to native graphics processor (GPU) code. Developing vision models in CCT is far simpler and more productive than writing GPU code directly, but without sacrificing the performance gains of GPU acceleration. This programming model scales to interesting problems like dense optic flow, anisotropic diffusion, and deep learning. CCT is particularly powerful when combining multiple state-of-the-art techniques in a single algorithm.

Introduction

Graphics processor unit (GPU) acceleration is one of the driving forces behind the rapid growth of deep learning [1]. For the right workloads, a GPU implementation can be radically faster and more efficient than an equivalently optimized CPU implementation. Dense matrix-matrix multiplication is one such workload. Google's TensorFlow platform leverages optimized parallel CPU and GPU operators for matrix multiplication [2]. TensorFlow 0.12.1 with a single consumer-grade NVIDIA GTX 1080 GPU can multiply two 16,384 by 16,384 single-precision matrices in approximately 1.85 seconds, including the overhead of transferring data back to main memory. Running the same TensorFlow graph on an Intel i7-6700K CPU at 4 GHz requires approximately 82 seconds per multiplication. The GPU implementation requires 44X less time to complete.

GPUs, however, use a different programming model than conventional CPUs and can be much harder to program. Two key productivity hurdles when working with GPUs are the need for *kernel fusion* and the lack of *performance portability*. Kernel fusion in an optimization in which two or more GPU kernel functions are merged into a single kernel function. Merging functions in this way requires more development time and produces code that consumes more register space on the GPU, but saves GPU memory bandwidth and GPU global memory space.

When implementing a simple computation like $(a+b)/2$, for instance, a more modular approach would be to build one kernel for addition and one kernel for division by a constant. This approach offers maximum flexibility and makes code changes easy. It requires two kernel invocations to execute the function, however, and memory bandwidth utilization is poor. An optimal implementation would merge those two kernels into a single, specialized kernel that adds two variables and divides by a constant. The developer would need to write a new specialized kernel for every function. In many real-world applications, however, minimizing GPU memory bandwidth consumption is critical for achieving good performance and kernel fusion is therefore an unavoidable optimization step.

The work required to write good specialized kernels would be less of a productivity concern if not for the rapid evolution and heterogeneity of GPUs. Each vendor balances parallelism, clock frequency, local memory, and the memory hierarchy differently. Even from a single vendor, the balance points from generation to generation can change significantly. In practice, this often means that some amount of re-tuning is required every time a GPU-accelerated program needs to run on new hardware. Vendors like NVIDIA have smoothed away much of the pain for common application domains by providing libraries of accelerated primitives. The cuDNN and cuBLAS libraries are two examples from NVIDIA [3]. The vendor updates these libraries as they release new hardware to ensure the accelerated primitives are always efficient. For less common application domains, users are left to handle per-device performance tuning on their own.

The cognitive computing toolkit

The HPE cognitive computing toolkit (CCT), formerly published under the name ”Cog ex Machina” [4], is an open-source software platform designed for GPU-accelerated modeling. It works well for workloads like deep learning and many classes of vision problems. CCT is available under an Apache 2.0 license at `https://github.com/hpe-cct`.

Like TensorFlow, CCT is an *embedded domain-specific language*. This means it provides programming primitives aimed a specific problem domain, rather than trying to target general programs. The *embedded* attribute means that it is hosted inside another language and uses the tooling from that host language. For TensorFlow, the host language is Python. CCT uses the Scala language. Developing CCT programs works much like developing regular Scala programs. All of the same integrated development environments, build tools, and dependency managers work exactly as they do for ordinary Scala. No CCT code is invoked until the program is run.

At run time, the CCT compiler and execution manager build an internal representation of the computation the user wants to execute, optimizes that representation to a minimum set of GPU kernels, emits appropriate GPU code, and manages the scheduling of the necessary GPU kernels on the available devices.

CCT holds all program state in an abstraction called a *compute graph*. Inside the compute graph, a *field* is the abstraction for data, an *operator* is the abstraction for computation, *sensors and actuators* allow developers to get state into or out of a compute graph, and the *feedback operator* allows learning and adaptation. The core data type, a field, is an N-dimensional array of tensors. N may be zero, one, two, or three. Each tensor in the field may be be of order zero through order three. All tensors in a given field

```
object BackgroundSubtraction extends CogDebuggerApp (
  new ComputeGraph {
    val movieFile = "resources/courtyard.mp4"
    // Build an asynchronous sensor bound to a movie file on disk. The sensor
    // will read frames as quickly as possible.
    val movie = ColorMovie(movieFile, synchronous = false).toVectorField()
    // Build an array of vectors matching the dimensions of the input movie
    val background = VectorField(movie.fieldShape, Shape(3))
    // Compute a low-pass filter over the video
    background <== 0.999f * background + 0.001f * movie
    // Compute an error norm between the background state and current frame
    val suspicious = reduceSum(abs(background - movie))
    // Probes - prevent the optimizer from removing these marked fields
    probe(movie)
    probe(background)
    probe(suspicious)
  }
)
```

Figure 1. *Source code for the BackgroundSubtraction application. The program lives inside a ComputeGraph, which is owned by the wrapping debugger application instance. The debugger allows a developer to step, run, or reset the program at will while visualizing the program state. The "movie" sensor binds to a video file on disk. In asynchronous mode, it will read frames as quickly as the compute graph can step. Synchronous mode locks the step rate of the compute graph to the native frame rate of the video file. The "background" field is a variable of the same dimensions as the input video. On each step of the compute graph, the feedback operator "<==" indicates that the state of the background field should be updated with 0.999 times the previous background state and 0.001 times the current frame. This is a low-pass filter. The suspicious field is an error metric computed over the current background state and the current frame. Probe annotations mark fields that the user may want to look at. The optimizer is free to merge away any non-probed fields.*

must have the same shape. All fields have an implicit time dimension. For example, a 1080p color video would be represented in CCT with a 1920x1080 field containing vectors of length three. Each vector plane holds a single color channel. The time dimension indicates which frame of the video is loaded.

Operators allow developers to construct new fields as a function of existing fields. Operators in CCT range from simple functions like addition up to complex primitives like frequency-domain convolution. Users can define their own operators by writing functions that use existing CCT primitives and functions. To define new primitives, CCT also provides a low-level GPU operator API. This low-level API is suitable for portions of an application that are too difficult for CCT to automatically optimize.

Sensors and actuators are the primitives by which users can move data into or out of a compute graph. On each step of the compute graph, the CCT framework will execute an arbitrary user-defined CPU function for each sensor or actuator. For a sensor, the framework will pass the user function a standard CPU buffer to fill. For an actuator, the framework will provide the user function a copy of the fields driving that actuator.

The special feedback operator "<==" defines temporal relationships in the compute graph. In an application without feedback, there is no persistent state and the compute graph contains a directed acyclic graph of fields and operators. On each step of the compute graph, CCT will read data from the sensors, execute any intermediate operators, and write to any bound actuators. The feedback operator allows persistent state by allowing a field to update its state as a function of fields from the *previous* step of the compute graph.

The source code for a simple example containing feedback is shown in Figure 1. In this example, the "background" field is defined by a feedback loop. On each step of the compute graph, the background state updates using its own previous state and the current state of the input to the application. This example is a simple low-pass filter, but the feedback primitive is powerful enough to represent much more sophisticated types of computation. The weights of a deep learning system implemented in CCT use the feedback primitive in exactly the same manner.

CCT includes a visual debugger to ease development of vision algorithms. This debugger automatically extracts the structure of the compute graph and allows users to visualize any part of the model state. Model stepping is controlled by the debugger, so users are free to step, run, stop, or reset the model at will. The sample application from Figure 1 running in the debugger is shown in Figure 2.

CCT optimization

The CCT programming model is designed for ease-of-use, but also to preserve critical information necessary to automatically emit efficient GPU code. Kernel fusion and kernel tuning are the two primary objectives of the CCT optimizer.

To enable kernel fusion, CCT adds a layer of abstraction on top of a conventional low-level GPU kernel. This abstraction, a *kernel fragment*, captures the inputs, outputs, and threading structure of the kernel code. A kernel fragment can contain an arbitrary computation, but must use the threading structure provided by the CCT kernel fragment API, read its inputs from the CCT API, and write its outputs to the CCT API. These hooks allow CCT to automatically wire together kernel fragments to produce merged kernels.

The kernel merger proceeds in two phases. In the first, the merger attempts to combine fields and operators that contribute

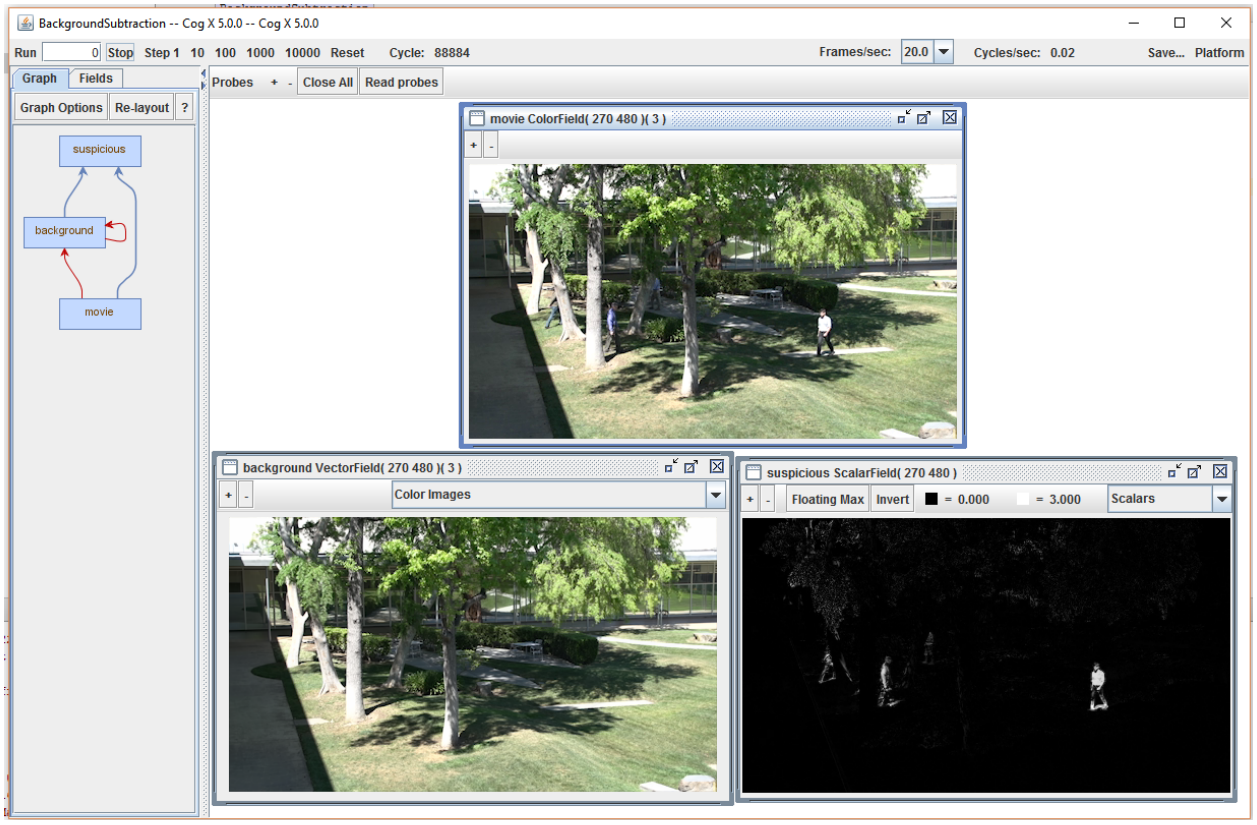
BackgroundSubtraction -- Cog X 5.0.0 -- Cog X 5.0.0
Run 0 Stop Step 1 10 100 1000 10000 Reset Cycle: 88884 Frames/sec: 20.0 Cycles/sec: 0.02 Save... Platform
Graph Fields
Graph Options Re-layout ?
Probes + - Close All Read probes
suspicious
background
movie
movie ColorField(270 480)(3)
background VectorField(270 480)(3)
Color Images
suspicious ScalarField(270 480)
Floating Max Invert = 0.000 = 3.000 Scalars

Figure 2. *The BackgroundSubtraction application running in the CCT visual debugger. The application ingests a video stream from a fixed camera (top panel), performs a low-pass filter to build a background model (bottom left panel), and computes an error metric to identify pixels in the current frame that differ significantly from that background model (bottom right panel). This program runs at approximately 1000 frames per second on an NVIDIA GTX 1080 GPU.*

towards producing a single output. After this phase, a single-output expression like $(a+b)/2$ would consist of only one merged kernel. The second phase attempts to combine fields and operators with shared inputs. This phase can produce merged kernels with multiple outputs. The sample code in Figure 1 is a case where multi-output merging is important. Both the background and suspicious fields depend on the previous state of the background field and the current movie frame. The multi-output merging phase is able to produce a single, merged kernel that produces a value for both the next background state and the suspicious field.

Merging scales to complex vision algorithms. Figure 3 shows the output of a dense optic flow algorithm running on CCT. The implementation runs approximately 5x faster than real time on a single NVIDIA GTX 1080 GPU.

To address the performance portability problem, CCT includes a simple kernel autotuner with online profiling. Rather than requiring developers to specify a single, concrete implementation of a kernel fragment, the autotuner allows developers to specify two or more candidate implementations. At runtime, the autotuner will benchmark each candidate implementation on the available hardware and choose the fastest option. The results of this benchmarking process are cached to eliminate redundant testing of a single candidate kernel fragment on the same GPU.

The CCT deep learning package leverages the autotuner to optimize frequency-domain convolution. Frequency-domain convolution requires a computation involving local shared memory, so efficiency depends on each GPU thread allocating a chunk of work sized to match the available local shared memory on the available GPU. The amount of local shared memory per execution unit is highly variable from GPU model to GPU model. The CCT kernel autotuner is able to find a suitable work size allocation at runtime, eliminating the need for manual GPU detection and parameter tuning in the deep learning package.

Conclusions

The HPE Cognitive Computing Toolkit (CCT) is an open-source modeling platform backed by Hewlett Packard Enterprise. CCT is available under an open-source Apache 2.0 license at `https://github.com/hpe-cct`.

CCT offers a number of benefits for accelerated vision modeling. CCT code is easy to write and easy to change, but preserves enough information that the CCT optimizer can emit efficient GPU code. GPU-accelerated applications can run significantly faster than an equivalently optimized CPU implementation on common commodity hardware. This ease-of-use and performance gain can significantly accelerate research in vision algorithms.

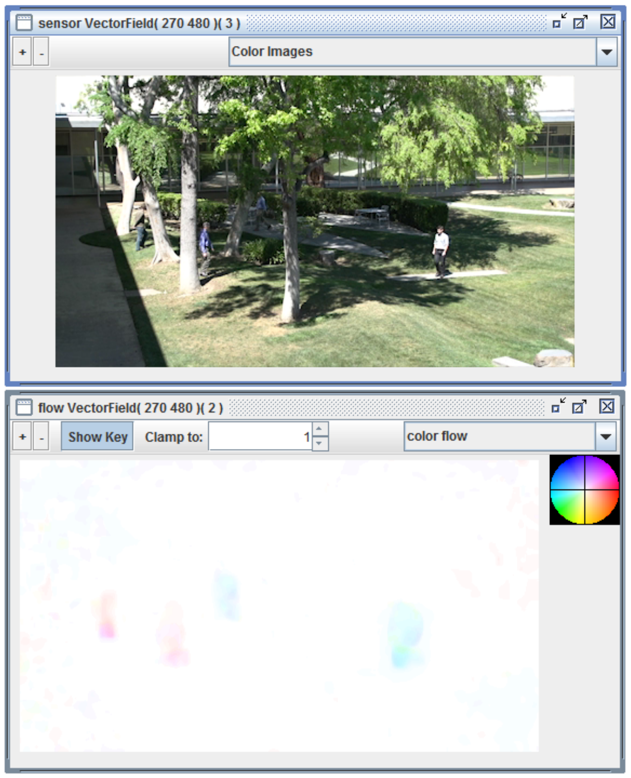
sensor VectorField(270 480)(3)
+
-
Color Images
flow VectorField(270 480)(2)
+
-
Show Key
Clamp to: 1
color flow

Figure 3. *Dense optic flow implementation in CCT. The raw video (top frame) shows pedestrians moving around a courtyard from a fixed vantage point. The optic flow output (bottom frame) indicates that the leftmost two people are moving right and rightmost two people are moving left. This program runs at approximately 350 frames per second on an NVIDIA GTX 1080 GPU.*

References

[1] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems (2012).

[2] Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016).

[3] Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014).

[4] Snider G, Amerson R, Carter D, Abdalla H, Qureshi MS, Lveill J, Versace M, Ames H, Patrick S, Chandler B, Gorchetchnikov A. From synapses to circuitry: Using memristive memory to explore the electronic brain. Computer (2011).

Author Biography

Ben Chandler received his BS in cognitive science from Carnegie Mellon University (2007) and his PhD in cognitive and neural systems from Boston University (2014). He joined HP Labs in 2011, moved to Hewlett Packard Labs when Hewlett Packard Enterprise split from HP in 2015, and then left the company for freelance work in December 2016. His research focuses on applying machine intelligence at scale to real-world problems.