To appear in ACM TOG 35(6).

Supplemental Material for VizGen: Accelerating Visual Computing Prototypes in Dynamic Languages

Yuting Yang[1] Sam Prestwood[1] Connelly Barnes[1]
[1]University of Virginia

Abstract

This is a supplemental document for the paper "VizGen: Accelerating Visual Computing Prototypes in Dynamic Languages." In Section 11, we explain in detail when compiled versus interpreted code is generated. Next, in Section 12, we present results from the VizGen compiler for applications that use 32-bit floating-point precision.

11 Compiled versus interpreted code

In this section, we explain in detail when native C versus interpreted Python code is generated. When reading this paragraph, please refer to the main document for the VizGen compiler, Section 5, "Program transformations," and Section 9.1, "Assessing the effect of each transformation."

Frequently, significant performance gains are due to rewriting code that calls the Python interpreter to native C code. Four of our transformations help transform interpreted code to native C: *type specialization*, *API call rewriting*, *loop over implicit variables*, and *vectorize innermost*. We now briefly discuss how each of these four transformations converts code to native C. *Type specialization* can substitute in native C types such as scalars, fixed-length arrays, or variable-length arrays. *API call rewriting* can rewrite API calls within the Python interpreter to corresponding functions that have been implemented in C. *Loop over implicit variables* generates fused native C array operations from Python array operations with known dimensionality or sizes. *Vectorize innermost* is an alternative method of generating native C SIMD array code that applies only for special cases where the last dimension of an array is known to be 2, 3, or 4.

12 Evaluation: 32-bit floating-point results

In this section, we present results for applications that use 32-bit floating-point precision. When looking at these results, please compare them with the main document for the VizGen compiler, Section 9, "Evaluation," and the corresponding Table 1, which presents run-times for each application. In the main document for VizGen, results are presented for both 64-bit floating-point mode, and an "approximating" mode which selects either 64-bit or 32-bit precision based on whichever is faster. In this section, we present in Table 4 results where all applications and compilers are using 32-bit floating-point mode.

Application	**Ours Time [ms]**	**Ours Speedup vs**						**Ours Lines**	**Ours Shorter vs**	
		Python	**Numba**	**PyPy**	**Pythran**	**unPython***	**C code**		**vs C**	**vs Cython**
Bilateral grid	86.4	1284×	1334×	3755×	Error	1146×	0.8×	101	2.6×	3.0×
Camera pipeline	0.9	2120×	2074×	2889×	Error	707×	1.2×	173	1.6×	3.0×
Composite (gray)	0.3	2283×	0.7×	42×	2.4×	1.0×	1.1×	6	2.3×	3.0×
Composite (RGB)	0.4	2859×	75×	1603×	12×	1864×	1.2×	6	2.7×	3.0×
Harris corner	11.5	4308×	3.7×	4590×	Error	1.4×	1.3×	92	1.2×	3.2×
Interpolate	6.7	451×	Error	338×	Error	402×	8.8×	39	4.8×	4.8×
Local Laplacian	3.1	775×	Error	820×	Error	423×	1.9×	76	4.8×	3.7×
Mandelbrot	20.0	279×	201×	3.7×	10×	1.0×	3.2×	29	1.6×	2.8×
One stage blur (gray)	0.5	1793×	2.6×	Error	3.2×	1.1×	1.4×	31	1.8×	9.5×
One stage blur (RGB)	0.9	3761×	325×	12994×	60×	3464×	1.4×	31	2.5×	2.3×
Optical flow	9.5	2793×	Error	Error	Error	2344×	0.9×	232	1.6×	3.1×
Pac-Man	0.1	274×	0.8×	4.0×	1.9×	2.9×	10×	111	1.2×	2.0×
Raytracer	1.7	1545×	1564×	1058×	Error	1380×	0.8×	49	3.6×	3.1×
Two stage blur (gray)	0.5	692×	0.8×	141×	4.4×	1.3×	1.5×	10	2.1×	4.3×
Two stage blur (RGB)	0.8	2118×	82×	1474×	25×	2010×	3.7×	10	2.5×	4.3×
Median		2118×	79×	1058×	7.2×	423×	1.4×		2.3×	3.1×

Table 4: *Comparison of the speedups and lines of code for our compiler versus alternatives, with applications in 32-bit mode. Please consult Section 9 and Table 1 of the main paper for a full description of the applications and the different measurements in these columns.*