AUTOMATICALLY BATCHING CONTROL-INTENSIVE PROGRAMS FOR MODERN ACCELERATORS

Alexey Radul [1] **Brian Patton** [1] **Dougal Maclaurin** [1] **Matthew D. Hoffman** [2] **Rif A. Saurous** [3]

ABSTRACT

We present a general approach to batching arbitrary computations for accelerators such as GPUs. We show orders-of-magnitude speedups using our method on the No U-Turn Sampler (NUTS), a workhorse algorithm in Bayesian statistics. The central challenge of batching NUTS and other Markov chain Monte Carlo algorithms is data-dependent control flow and recursion. We overcome this by mechanically transforming a single-example implementation into a form that explicitly tracks the current program point for each batch member, and only steps forward those in the same place. We present two different batching algorithms: a simpler, previously published one that inherits recursion from the host Python, and a more complex, novel one that implemenents recursion directly and can batch across it. We implement these batching methods as a general program transformation on Python source. Both the batching system and the NUTS implementation presented here are available as part of the popular TensorFlow Probability software package.

1 INTRODUCTION

Modern machine learning accelerators such as GPUs are oriented around Single Instruction Multiple Data (SIMD) parallelism—doing the same thing to each item of a big array of data at once. Machine learning programs optimized for such accelerators generally consist of invoking *kernels*, where each kernel is a separately hand-tuned accelerator program for a specific function. Good utilization of the accelerator comes of making relatively few kernel calls, with each kernel processing a relatively large amount of data. In the case of a typical neural network workload, the kernels would be "matrix multiplication" or "convolution", and the call sequence would encode the architecture of the neural network.

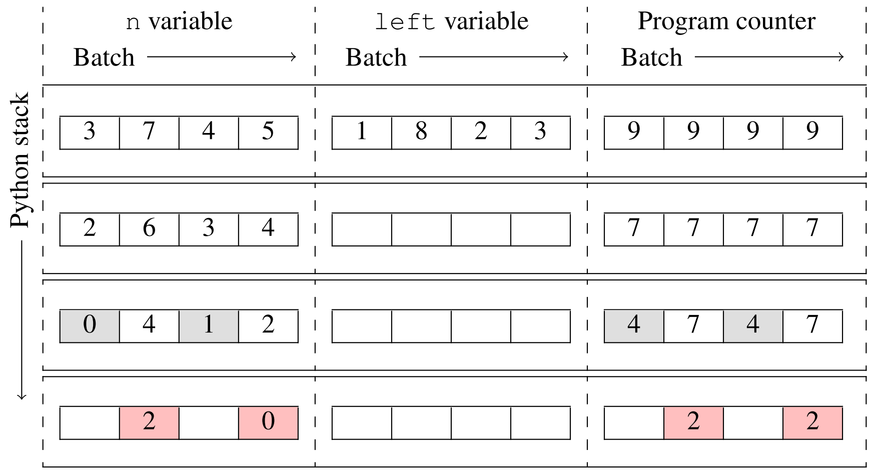
Let's briefly look at the resulting programming model. This review is worded in the TensorFlow (Abadi et al., 2015) ecosystem, since that's the setting for our work, but other machine learning frameworks are broadly similar. The top-level program is generally written in Python, calling TensorFlow API functions that correspond to kernels such as matrix multiplication. These functions can be executed immediately, in the so-called TensorFlow *Eager mode*. In this case they can be arbitrarily interleaved with the host Python, including control flow; but suffer corresponding dispatch and communication overhead. Alternately, the same API functions can be used to construct an operation graph to be executed all at once. This is the so-called TensorFlow *graph mode*. The advantage is that graphs can be saved, loaded, and optimized before being run, and suffer less dispatch overhead. The disadvantage is that graph computations cannot be interleaved with the host Python, and in particular graph mode cannot represent recursive computations. A third option is to further compile the graph with XLA (The XLA Team, 2017). XLA imposes even more restrictions, such as statically resolving the shapes of all intermediate arrays, but offers the additional benefit of fusing kernels together, which reduces dispatch overhead even more.

Good performance in this programming style depends heavily on vectorization, both within the kernels and at the level of kernel inputs. One very common strategy for vectorizing machine learning programs is so-called *batching*: processing a batch of independent inputs in lock-step in order to get more play for vectorization. Batching can also reduce per-input memory pressure: in the case of a neural network with N features, each input has size $O(N)$, whereas the weight matrices can easily have size $O(N^2)$. Running multiple inputs through the layers of the network in lock-step can re-use each weight matrix for many examples before having to evict it from memory caches in order to load the next one.

It is standard practice in machine learning frameworks such as TensorFlow or PyTorch (Paszke et al., 2017) to code the kernels to accept extra input dimensions and operate elementwise across them. Consequently, coding a batched version

[1]Google, Cambridge, Massachusetts, USA [2]Google, New York, New York, USA [3]Google, Mountain View, California, USA. Correspondence to: Alexey Radul <axch@google.com>.

Proceedings of the 3^{rd} MLSys Conference, Austin, TX, USA, 2020. Copyright 2020 by the author(s).



```
1  def fibonacci(n):
2    cond = n <= 1
3    if cond:
4      return 1
5    else:
6      n2 = n - 2
7      left = fibonacci(n2)
8      n1 = n - 1
9      right = fibonacci(n1)
10     return left + right
```

Figure 1. Runtime operation of a locally, statically auto-batched recursive Fibonacci program. This snapshot occurs on the batch of inputs $3, 7, 4, 5$. The batching transformation adds storage for all the batch members and handles divergent control flow by masking. The recursion is handled in Python. In this example, there are two "active" logical threads about to execute lines 2-3 of the Fibonacci program, highlighted in red. There are also two logical threads suspended one Python stack frame earlier, waiting for the active threads to re-converge with them so they can all return from that frame. The runtime cannot batch together logical threads with different call stacks, because those stacks are embedded in the runtime's Python-level call stack. The `left` variable has no value in most of the shown stack frames because the program hasn't assigned it yet.

of a straightline program is relatively straightforward, if somewhat tedious and error-prone. Simple neural networks being straightline, batch training is the norm. Obstacles arise, however, when one wishes to batch a program with control flow, such as conditionals or variable-length loops. Then it becomes necessary to keep track of which batch member takes which branch of each conditional, and avoid or ignore computations on batch members at the wrong program point. The difficulty of doing this by hand impedes using sophisticated classical algorithms in machine learning. Despite the impedance, people have used tree searches (Silver et al., 2016), optimization routines (Amos & Kolter, 2017) and ordinary differential equations solvers (Chen et al., 2018) in machine learning work; what else could we accomplish if it were easier?

Additional obstacles arise when trying to run a recursive program on a modern machine learning framework, in batch or otherwise, because the dataflow graph representation cannot execute recursion natively. This is as true in XLA or TensorFlow graph mode as it is in other graph-oriented machine learning frameworks like Caffe (Jia et al., 2014). The user is therefore forced to fall back to eager-style execution, paying more communication overhead. If machine learning is to benefit fully from the last 60 years of computer algorithm development, we must be able to run recursive algorithms reasonably efficiently.

Our goal in this paper is to push the boundary of what classical algorithms can efficiently execute on accelerators, in the context of modern machine learning frameworks. In particular, we

- Introduce *program-counter autobatching* (Section 3), a global, static program transformation for batching programs with arbitrary control flow, and materializing recursion into an underlying dataflow system.

- Demonstrate that program-counter autobatching can successfully accelerate the No U-Turn Sampler, a classic algorithm from Bayesian statistics, by compiling its recursion into explicit stack management, and by statically constructing a schedule for running it on batches of inputs.

- Provide, using the same vocabulary, a formal description of *local static autobatching* (Section 2). This is a simpler and lower-overhead batching transformation with less batching power in the recursive case.

- Survey (Section 5) the local static autobatching systems (Agarwal, 2019; Bradbury & Fu, 2018; Bradbury et al., 2017–2019) that have been implemented for several machine learning frameworks.

- Directly compare these two autobatching strategies on a test problem from Bayesian statistics (Section 4).

Program-counter autobatching is available as a module in the popular TensorFlow Probability (The TFP Team, 2018–2019; Dillon et al., 2017) software package. That module also implements a local static autobatching variant for comparison.

$$
\begin{array}{rcll}
\text{Program} & P & ::= & [F] \\
\text{Function} & F & ::= & \text{input } x, \text{body } [B], \text{output } y \\
\text{Block} & B & ::= & [op], t \\
\text{Operation} & op & ::= & \text{Primitive } y = f(x) \\
 & & & \mid \text{Call } y = F(x) \\
\text{Terminator} & t & ::= & \text{Jump } i \mid \text{Branch } x\ i\ j \mid \text{Return} \\
 & f & ::= & \sin \mid \cos \mid \ldots
\end{array}
$$

Figure 2. Syntax of locally batchable programs. We use $[\cdot]$ to denote ordered lists. The symbols x, y range over variable names, and i, j index blocks within the same function. We present a unary syntax for succinctness; the n-ary generalization is standard.

2 LOCAL STATIC AUTOBATCHING

The simplest batching strategy (whether automated or hand-coded) is to retain the graph of the computation as-is and just transform every operation into a batched equivalent. We call this *local static autobatching*. Intuitively, it's "local" because the pattern of composition of operations doesn't change, and every operation can be transformed on its own; and it's "static" because the batching schedule doesn't depend on the input data, and can thus be computed before starting execution.

When extending this idea to programs with control flow, it is necessary to at least introduce a mask of which batch members are "currently active". One then arranges to execute every control path that at least one batch member follows, and avoid or ignore each computation for each batch member that did not take that path. If the program being batched is recursive, the recursion still has to be carried out by the control language, i.e., Python. The runtime operation thus looks like Figure 1.

Local static autobatching can be implemented in many styles. For the sake of clarity, we will describe it as a nonstandard interpretation of a simple control flow graph language, given in Figure 2. In addition to eliminating many incidental considerations, this presentation aligns with the presentation of program-counter autobatching in Section 3, which will be a (different) nonstandard interpretation of a very similar language. Going through this presentation first also allows us to compare to other local static autobatching systems more precisely, in Section 5.

The nonstandard interpretation itself is given in Algorithm 1. In addition to storage for all the batch member inputs, we maintain an *active set* (initially the whole batch) and a *program counter* (initially the start of the entry point). The active set is a mask—all inactive batch members are ignored and never modified until they become active. The program counter gives the program point (as a basic block index) each active batch member is waiting to execute. The execution model is simple: at each step, we select some basic block that has at least one active batch member and execute it in batch. We then update the data storage and program counters of just those *locally active* batch members. Repeat until all active batch members have exited the function, then return.

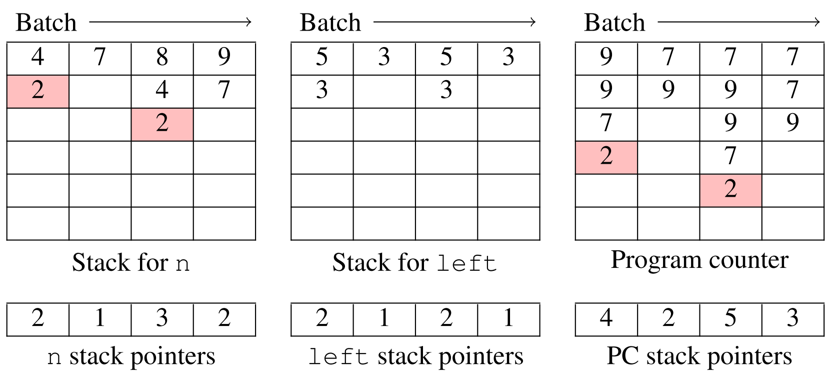
Algorithm 1 Local static autobatching

```
Input: Function $F$ with $I$ basic blocks $B_i$, input variable $x$, and output variable $y$;
Input: Batch size $Z$;
Input: Data array $T$ with leading dimension $Z$;
Input: Active set $A \subseteq \{0, 1, \ldots, Z-1\}$.
Initialize length $Z$ program counter $pc = [0, 0, \ldots, 0]$
Initialize $x = T$
while (for any $b \in A$, $pc_b < I$) do
  Set block index $i = \min_{b \in A} pc_b$
  Compute locally active set $A' = \{b \in A | pc_b = i\}$
  for $op \in B_i$ do
    if $op$ is (Primitive $y = f(x)$) then
      Compute outputs $o = f(x)$
      Set $y_{A'} = o_{A'}$
    else if $op$ is (Call $y = G(x)$) then
      Recursively compute outputs:
      $o = \text{Local-static}(G, Z, x, A')$
      Set $y_{A'} = o_{A'}$
    end if
  end for
  if $t_i$ is Jump $j$ then
    Set $pc_{A'} = j$
  else if $t_i$ is Branch $x\ j\ k$ then
    for $b \in A'$ do
      Set $pc_b = j$ if $x_b$ otherwise $pc_b = k$
    end for
  else if $t_i$ is Return then
    Set $pc_{A'} = I$
  end if
end while
return Current value of $y$
```

If the block we are executing ends in a branch (i.e., the prelude of a source language `if` statement), the locally active batch members may *diverge*, in that some may move to the true branch and some to the false. They will *converge* again when both of those branches complete, and we continue after the end of the `if`.

If the block we are executing contains a (potentially recursive) call to a function the user asked us to auto-batch, we appeal to the host language's function call facility. The only trick is to update the active set in the recursive autobatching invocation to include only the locally active batch members (i.e., those whose program counter was at that call).

Why does this work? Consider this runtime from the point of view of one batch member. It wants to execute some



Stack for n

4	7	8	9
2		4	7
		2	

Stack for left

5	3	5	3
3		3	

Program counter

9	7	7	7
9	9	9	7
7		9	9
2		7	
		2	

n stack pointers

2	1	3	2

left stack pointers

2	1	2	1

PC stack pointers

4	2	5	3

```
1  def fibonacci(n):
2    cond = n <= 1
3    if cond:
4      return 1
5    else:
6      n2 = n - 2
7      left = fibonacci(n2)
8      n1 = n - 1
9      right = fibonacci(n1)
10     return left + right
```

Figure 3. Runtime operation of a program counter auto-batched recursive Fibonacci program. This snapshot occurs on the batch of inputs $6, 7, 8, 9$. In addition to the batch dimension (across), the batching transformation also augments every non-temporary variable from the program with a stack dimension (down), and an array of stack pointers. Additionally, the runtime maintains a `program counter` variable that records which block each logical thread is waiting to execute. At each time step, the runtime selects a basic block to run (lines 2-3 in this example) and updates the state and program counter of the logical threads executing that block (highlighted in red). Because recursive state is captured explicitly in the arrays storing the data, the runtime doesn't need to itself rely on recursion in Python (the host language). This means both that it can be executed in TensorFlow's graph mode, and that it can let logical threads re-converge on function calls, even at different stack depths. Note that the stack for the `n` variable will only hold values in frames where the program counter hasn't moved past line 8, where `n` is last used. Conversely, `left` is only pushed in frames where the program counter is past line 7.

sequence of basic blocks, as given by the edits to its program counter. Every time the runtime runs one of those basic blocks, it updates the state of that batch member the same way it would if the batch had size 1. And every time the runtime runs some other block, it doesn't update the batch member at all. The only way this can fail is if some underlying batch operation in the platform doesn't treat batch members independently (e.g., if an error in one batch member causes an exception which aborts execution of all of them) or if some batch member doesn't terminate and starves the others.

There are two significant free choices in this runtime. The first is how to execute a primitive operation on some batch members but not others. Algorithm 1 is written in *masking* style: we run the primitive on all the batch members, and just ignore the results of the ones that were at different points in the program. This is simple and has very low in-system overhead, because masking is a cheap operation. The down side is that it wastes computation on the batch members that are going to be masked out, which can be significant if batch utilization is low. There is also the subtlely that this extra computation happens with junk data, which may trigger spurious failures in the underlying platform.

The other option for batching the primitive operations is to use the indices of the locally active batch members to gather the inputs into a smaller array, perform just the live computation, and then scatter the results back into the full runtime state. This avoids wasting computation and avoids computing on junk data, but gathering and scattering are more expensive than masking. Furthermore, the intermediate arrays will have statically indeterminate size, making the gather-scatter approach less effective on platforms like the XLA compiler for Google's TPUs (The XLA Team, 2017) that statically infer array shapes.

The second significant free choice in this runtime is the heuristic for selecting which basic block to run next. As long as we don't starve any blocks, any selection criterion will lead to a correct end result. Algorithm 1 encodes a surprisingly effective choice: always run the earliest available block in program order. This has the merit of being (relatively) predictable by the user; but more refined heuristics are definitely possible.

In our implementation, the frontend for this is a Python-embedded compiler. That is, it's a user-invoked AST transformation based on AutoGraph (Moldovan et al., 2018) that converts the user program into the form given in Figure 2. All the user's actual computations become $\mathrm{Primitive}$ operations, and the control and recursion constructs are encoded in a standard way in Jump, Branch, Call, and Return instructions.

3 Program Counter Autobatching

The local static autobatching discussed in Section 2 has an interesting limitation. Because it relies on the Python stack to implement recursion, it cannot batch operations across different (recursive) calls to the same user function. So two batch members could be trying to execute the same code and not be batchable because the system doesn't have a long-enough sight line to connect them. And, of course, relying

$$
\begin{array}{rlll}
\text{Program} & P & ::= & \text{input } x, \text{code } [B], \text{output } y \\
\text{Block} & B & ::= & [op], t \\
\text{Operation} & op & ::= & \text{Push } y = f(x) \mid \text{Pop } x \\
\text{Terminator} & t & ::= & \text{Jump } i \mid \text{Branch } x\ i\ j \\
 & & & \mid \text{PushJump } i\ j \mid \text{Return} \\
 & f & ::= & \sin \mid \cos \mid \dots
\end{array}
$$

Figure 4. Syntax of program counter batchable programs. We use $[\cdot]$ to denote ordered lists. The symbols x, y range over variable names, and i, j index blocks of the program. This syntax is also unary for succinctness. The difference from locally autobatched programs (Figure 2) is that all control flow graphs are merged, and Call operations are replaced with explicit stack manipulation operations. Push and Pop save and load data; PushJump i j jumps to block i after setting up a return to block j; and Return returns by popping the program counter stack.

on Python to manage the recursion imposes communication costs and limits the optimizations the underlying machine learning framework can do.

We can serve two purposes with one intervention by implementing the stack within the autobatching system. We choose to give each program variable its own stack (by extending the relevant array with another dimension), getting a runtime state that looks like Figure 3. The layout of the figure is intentionally the same as Figure 1 to emphasize that we are representing all the same stuff, just in a different way.

To implement this, we want a slightly different control flow graph language, shown in Figure 4. Since the runtime is now managing the stacks itself, we replace the Call instruction with explicit (per-variable) Push and Pop instructions, as well as PushJump for entering function bodies. The Push also computes the value to write to the top of the variable's stack. The language is otherwise the same as Figure 2, and indeed our implementation compiles to the latter first and then lowers from there to the former.

3.1 Runtime

The runtime is spelled out in Algorithm 2. As compared with local static autobatching (Algorithm 1), the active set no longer persists across steps, while the explicit program counter takes on a more central role (hence the name). The program counter now has a stack dimension of its own. The locally active set can now include batch members at different stack depths, giving the runtime a chance to batch their computations together. Consequently, computations can converge by calling into the same subroutine from different code locations, and conversely diverge by returning thereto.

The major advantage of managing variable stacks and the program counter is that the runtime is no longer itself recursive, so can be coded completely in a system like (graph-

Algorithm 2 Program counter autobatching

```
Input: Program $P$ with $I$ basic blocks $B_i$, input variable $x$, and output variable $y$;
Input: Batch size $Z$; Stack depth limit $D$;
Input: Data array $T$ with leading dimension $Z$.
Initialize $D$-by-$Z$ program counter $pc = [0, 0, \dots, 0]$
Initialize length $Z$ stack indexes $pc_{stack} = [0, 0, \dots, 0]$
for variable $v$ do
  Initialize $v$ to zeros with leading dimensions $D, Z$
  Initialize length $Z$ indexes $v_{stack} = [0, 0, \dots, 0]$
end for
PUSH $T$ onto $x$
Initialize length $Z$ $pc^{top} = pc[pc_{stack}]$
while any $pc^{top} < I$ do
  Set next block index $i = \min pc^{top}$
  Compute locally active set $A = \{b | pc_b^{top} = i\}$
  for $op \in B_i$ do
    if $op$ is Push $y = f(x)$ then
      Compute $x^{top} = x[x_{stack}]$
      Compute outputs $o = f(x^{top})$
      PUSH $o_A$ onto $y_A$
    else if $op$ is Pop $x$ then
      POP $x_A$
    end if
  end for
  if $t_i$ is Jump $j$ then
    Set $pc_A^{top} = j$
  else if $t_i$ is Branch $x$ $j$ $k$ then
    Compute $x^{top} = x[x_{stack}]$
    for $b \in A$ do
      Set $pc_b^{top} = j$ if $x_b^{top}$ otherwise $pc_b^{top} = k$
    end for
  else if $t_i$ is PushJump $j$ $k$ then
    Set $pc_A^{top} = k$
    PUSH $j$ onto $pc_A$
  else if $t_i$ is Return then
    POP $pc_A$
  end if
end while
return Current value of $y[y_{stack}]$
```

mode) TensorFlow or XLA that does not support recursion natively. In TensorFlow, the code corresponding to the main loop in Algorithm 2 looks like Figure 5. The loop body is a dispatch to the correct block based on the dynamic value of the next program counter. The main trick, common in partial evaluation (Jones et al., 1993), is that the block index `pc` on line 3 is static at TF graph build time. The `block` function (not shown) is hence just a direct transcription of the loop body in Algorithm 2. Picking out the `pc`th program block and interpreting that block's operations will happen once when constructing the TensorFlow graph, leaving just the Tensor operations needed to manipulate the autobatched variables. In effect, lines 3–4 are the final stage of our compiler, lowering the entire autobatched program from our representation of Figure 4 to raw TensorFlow operations.

```
1 while next_pc < halt_pc:
2   all_vars = tf.switch_case(next_pc,
3      [block(pc, all_vars)
4       for pc in valid_pcs])
5   next_pc = tf.reduce_min(
6      get_top(get_pc_var(all_vars)))
```

Figure 5. Pseudocode for implementing the main loop of Algorithm 2 in TensorFlow. The main idea is to dispatch at runtime on the dynamic `next_pc` (line 2), to the block compiled with the correct static `pc` (line 3). See text.

3.2 Optimizations

The price we pay for implementing our own stack is that reads from the stack structure must gather according to the stack depths (which may vary across batch members), and writes must correspondingly scatter. We implement five compiler optimizations to reduce this cost:

1. Each variable in the program being auto-batched gets its own stack, so we can optimize those stacks independently. In particular, we can arrange the stack operations in a per-variable caller-saves stack discipline to set up for later pop-push elimination.

2. The compiler statically detects whether each variable is live across an iteration of the runtime loop. Those that are not are temporary and don't need to be touched by the autobatching system at all: they exist only inside basic block executions.

3. The compiler also statically detects whether each variable is live across a recursive function call that might need to reuse it (at a different stack depth). Those that are not do not need a stack or a stack pointer, and autobatching only amounts to updating their top value with a mask to select only the active batch members.

4. For those variables that do require stacks, the runtime caches the top of each stack variable, so that repeated reads do not require large gather operations.

5. Finally, the compiler also statically cancels pairs of Pop followed by Push that have no intervening reads, and converts the latter into an in-place Update instruction (not shown) that only interacts with the cached top of each stack.

An important consequence of the above optimizations is that program counter autobatching will run a non-recursive program entirely without variable stacks (except for the program counter itself). It will thus replicate the performance of local static autobatching without needing a host-language stack for (non-recursive) function calls, while also being able to batch across them. Unlike a tracing-based system such as JAX (Bradbury et al., 2017–2019), this compiled approach also doesn't amount to inlining all function calls, so can autobatch a program with significant subroutine reuse without combinatorial explosion in code (or traced graph) size.

4 Experiments

We evaluate autobatching on the No U-Turn Sampler (NUTS), a workhorse gradient-based Markov-chain Monte Carlo method widely used in Bayesian statistics. We chose NUTS as a test target because (i) the standard presentation is a complex recursive function, prohibitively difficult to batch by hand, and (ii) batching across multiple independent chains can give substantial speedups.

Our results show that batching on GPUs enables NUTS to efficiently scale to thousands of parallel chains, and get inferential throughput well beyond existing CPU-based systems such as Stan. By demonstrating performance gains from batching NUTS, we hope to contribute to a broader practice of running large numbers of independent Markov chains, for more precise convergence diagnostics and uncertainty estimates.

We test autobatched NUTS on two test problems:

- Exploring a 100-dimensional correlated Gaussian distribution.
- Inference in a Bayesian logistic regression problem with 10,000 synthetic data points and 100 regressors.

We test three forms of autobatching NUTS:

- Program counter autobatching, compiled entirely with XLA;
- Local static autobatching, executed entirely with TensorFlow Eager; and
- A hybrid: Running the control operations of local static autobatching in TensorFlow Eager, but compiling the straight-line components (basic blocks) with XLA.

The purpose of the latter is to try and tease apart the benefits of compilation specifically for control flow versus compiling (and fusing together) straightline sequences of kernel invocations. It should be noted that identifying the basic blocks to compile separately is a nontrivial program transformation in its own right. It fits conveniently into our software framework, but represents considerable work to do by hand.

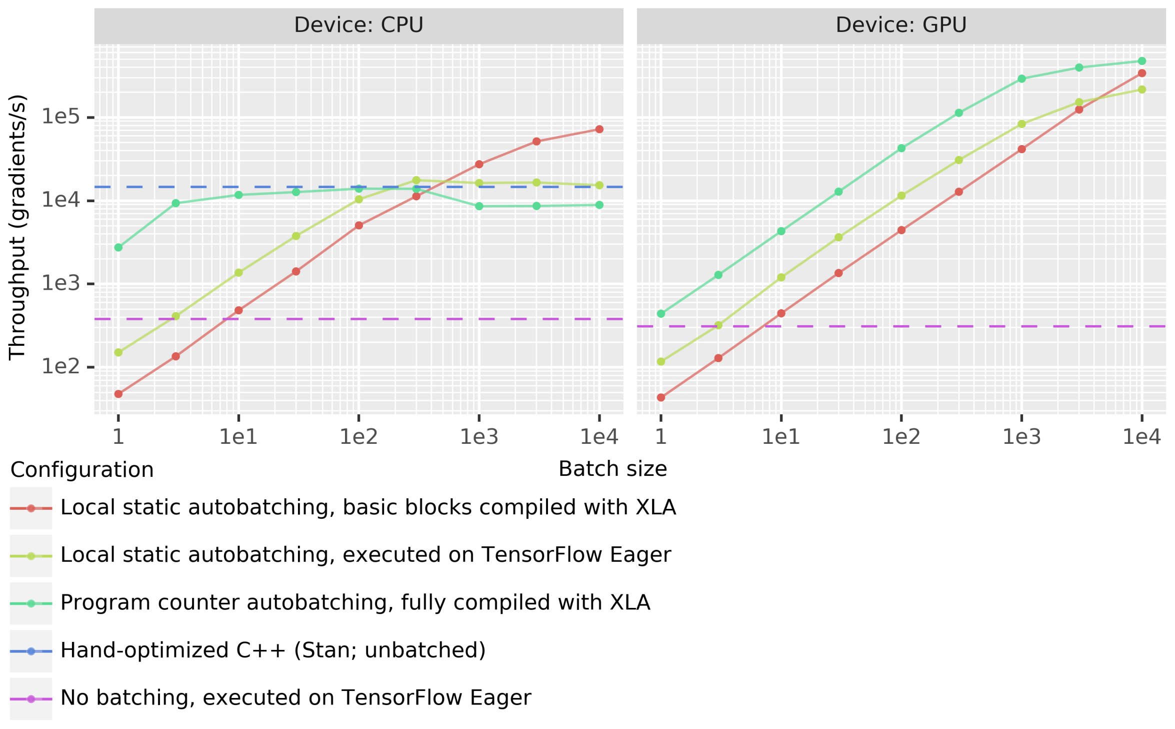


Figure 6. Performance of auto-batched No U-Turn Sampler on the Bayesian logistic regression problem (100 latent dimensions, 10,000 data points). The batch size refers to the number of chains running in tandem. The reported gradients are the total across all chains, excluding waste due to synchronization. We compare the performance of program counter autobatching compiled with XLA to our local static autobatching executed in TensorFlow's Eager mode. We also include two baselines. One is the same program executed directly in Eager mode without autobatching (perforce running one batch member at a time). The other is the widely used and well-optimized Stan implementation of (a variant of) the same NUTS algorithm. Batching provides linear scaling on all tested platforms, until the underlying hardware saturates. See text for details of the experimental setup.

4.1 Runtime on logistic regression

In Figure 6, we measure the number of model gradient evaluations per second we can get out of batching NUTS in different ways. The main effect we see is GPU throughput scaling linearly with batch size. We also see the speedup from avoiding cycling to Python (on the host CPU!) to implement the recursion.

The behavior when running entirely on CPU is more nuanced. CPUs are superb at control flow and recursion as it is, so the main effect of batching seems to be to amortize away platform overhead, until we match the performance of the Stan system's long-optimized custom C++ at a batch size of a few hundred—or just ten for compiling fully with XLA, whose per-leaf-kernel overhead is much smaller.

At very large batch sizes, however, the hybrid strategy of running local static autobatching in TensorFlow Eager but compiling the basic blocks with XLA outperforms all other NUTS implementations we tested. We are not quite sure why this happens, but we hypothesize that (1) it beats Stan because of better memory locality in batch evaluation of the target density; (2) it beats fully compiled autobatching by avoiding the overhead of gathering from and scattering to per-variable stack arrays; (3) it beats fully Eager local autobatching by avoiding per-leaf-kernel dispatch overheads; but (4) it's slower at low batch sizes because of per-fused-kernel invocation costs. We leave a complete investigation of this phenomenon to future work.

A few details of the experimental setup. These measurements are for the synthetic Bayesian logistic regression problem. The measured time counts only a warm run, excluding compilation, the one-time TensorFlow graph construction, etc. This allows measurements for small batch sizes not to be swamped by O(1) overhead. The CPU computations were run on a shared 88-core machine with 100 GiB of RAM allocated to the benchmark, in 32-bit floating-point precision. The GPU computations were run on a dedicated Tesla P100 GPU, also in 32-bit precision. In all cases, we slightly modified the published NUTS algorithm to take 4 steps of the leapfrog integrator at each leaf of the NUTS tree, to better amortize the control overhead. This has no effect on the soundness of the algorithm. The timings are

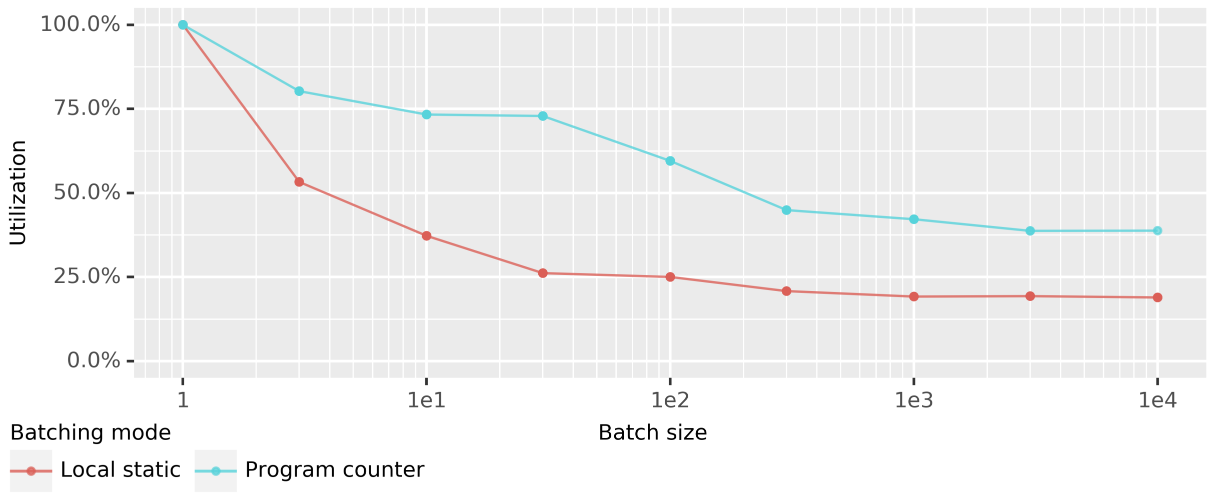


Figure 7. Utilization of batch gradient computation on the correlated Gaussian test problem. Utilization is less than 100% above 1 batch member because different batch members choose to use different numbers of gradients at each trajectory. We can see from the local-static line that on this problem, the longest trajectory that NUTS chooses at any iteration tends to be about four times longer than the average. Program counter autobatching recovers more utilization by batching gradients across 10 consecutive NUTS trajectories, instead of having to synchronize on trajectory boundaries.

best of five independent runs. Due to technical limitations, the Stan baseline was run on different hardware. We scaled its throughput against a calibration run of program counter autobatching on the same machine and precision.

4.2 Batch utilization on correlated Gaussian

We also measure the specific effect of batching across recursion depths. The NUTS algorithm dynamically chooses how many gradient steps to take in each trajectory. When running a multi-step Markov chain, one therefore has a choice of whether to synchronize on trajectory boundaries or on individual gradient steps. Local autobatching systems can only implement the former, because the control structure of the whole batch computation necessarily follows the control structure of the user program. Program counter autobatching, however, can synchronize on the gradients, for example evaluating the 5th gradient of the 3rd trajectory of one batch member in tandem with the 8th gradient of the 2nd trajectory of another. In the regime where the model gradients are expensive relative to the inter-trajectory book keeping, the latter should be preferable.

We find in Figure 7 that on a synthetic correlated Gaussian, synchronizing on trajectory boundaries leaves as much as a factor of 4 on the table even at a batch size as low as 30. Program counter autobatching is able to recover a factor of about 2 in utilization in as few as 10 NUTS trajectories. We expect gradient utilization to approach 1 in the limit of long chains as the per-chain distribution of total gradients taken approaches a Gaussian.

5 Related Work

The machine learning community has seen several systems arise to address the difficulty of hand-batching by batching user programs automatically. The extant ones have used one of two major architectures. The first is the local static autobatching we described in Section 2. The second, called *dynamic batching*, is exemplified by (Neubig et al., 2017) and (Looks et al., 2017). In this architecture, the runtime performs batching dynamically, by running parallel evaluations of the user program against a scheduler that manages the execution and batches opportunistically. From the lens of control flow, the advantage of dynamic batching is its ability to recover more batching (including within a single execution, if there is no data dependence). On the other hand, both local and program counter autobatching have less runtime overhead, because all the decisions about batch scheduling are done statically (at batch-program extraction time). For the same reason, the latter two architectures are more amenable to running on top of an existing machine learning framework, whereas dynamic batching requires support from the underlying graph executor.

The presentation in Section 2 gives one implementation style for the local static autobatching transformation. Other systems achieve the same effect in different ways.

Matchbox (Bradbury & Fu, 2018) relies on a relatively lightweight syntax transformation to intercept `if` statements and `while` loops, and otherwise accomplishes batching by defining a “batched array” type that carries the mask. The batched array overloads all the methods for a standard array with appropriate additional masking. Recursion is left to the ambient Python stack. In our terms, the mask corresponds to the active set.

At `if` statements, Matchbox first executes the `then` arm (if any batch members need it) and then the `else`. The program counter of Algorithm 1 is thus encoded in the queue (also maintained on the Python stack) of mask-block pairs to be executed. The data structure is equivalent: one vector of indices encodes the same information as a list of pairs of index with exclusive mask of items having that index. Storing the queue of program resumption points on the Python stack makes it more difficult for Matchbox to use a different heuristic for the order in which to run blocks, but the extant behavior is equivalent to the "run the earliest block" heuristic presented in Section 2.

Jax (Bradbury et al., 2017–2019) relies on an explicit tracing pass to construct an internal representation, on which batching (invoked via `jax.vmap`) is an explicit static transformation (one of several Jax can perform). Control flow requires using the Jax-specific control operators: `lax.cond` instead of `if` and `lax.while_loop` instead of `while`.

Recursion is not supported in Jax at all, because it confuses the tracer. There is therefore no stack. The program counter is encoded in a mask and an execution sequence the same way it is in Matchbox, with the same effects.

Similarly, TensorFlow's `pfor` facility (Agarwal, 2019; Agarwal & Ganichev, 2019) operates on the TensorFlow graph, including its `tf.cond` and `tf.while_loop` control operators. The transformation in `pfor` is the same as Jax's `vmap`, up to implementation details. Recursion is similarly not supported, because the underlying TensorFlow graph doesn't support it.

This is the sense in which the transformation is "local": this autobatching style (at least with this basic block choice heuristic) perserves the nesting structure of the user's original control constructs. As such, it can be implemented by a local transformation that, e.g., turns a `while` loop into a similar loop with a transformed body.

Somewhat farther afield, GPU programming languages such as CUDA (Nickolls et al., 2008) are also automatically vectorized. The handling of control constructs in CUDA is identical with local static autobatching, but of course only applies to kernels written therein. An interesting potential direction for application-level automatic batching could be to forward surface language control constructs through a compiler targeting CUDA (such as the GPU backend of the XLA compiler) and rely on CUDA to batch them. The ISPC compiler (Pharr & Mark, 2012) performs the same automatic vectorization transform for vector units in CPUs.

Finally, the NUTS algorithm is central enough to have prompted two rewrites in non-recursive form (Phan & Pradhan, 2019; Lao & Dillon, 2019) for the express purpose of running it on accelerators more effectively. The non-recursive form is also amenable to batching either by hand or using an established autobatching system (whether local or dynamic). One would expect such a manual effort to obtain better performance, but its labor-intensiveness necessarily limits its scope.

6 Conclusion

We presented program counter autobatching, a novel method for automatically vectorizing batch computations at the machine learning framework level. Program counter autobatching handles arbitrary control flow in the source program, including batching operations across recursion depth. We demonstrated the efficacy of the method by mechanically batching a (recursive) implementation of the No U-Turn Sampler, obtaining speedups varying (with batch size) up to three orders of magnitude. An implementation of program counter autobatching is available in the TensorFlow Probability package.

Acknowledgments

The authors would like to thank Delesley Hutchins for invaluable early critique of the architecture of the compiler; Sean Talts for benchmarking against Stan; Pavel Sountsov, Alex Wiltschko, James Bradbury, and Dan Piponi for insightful discussions; and the anonymous reviewers for their feedback.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Agarwal, A. Static automatic batching in TensorFlow. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 92–101, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/agarwal19a.html.

Agarwal, A. and Ganichev, I. Auto-vectorizing TensorFlow graphs: Jacobians, auto-batching and beyond, 2019. URL https://arxiv.org/abs/1903.04243.

Amos, B. and Kolter, J. Z. Optnet: Differentiable optimization as a layer in neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 136–145. JMLR. org, 2017.

Bradbury, J. and Fu, C. Automatic batching as a compiler pass in PyTorch. In *Workshop on Systems for ML*, Dec 2018. URL http://learningsys.org/nips18/assets/papers/107CameraReadySubmissionMatchbox__LearningSys_Abstract_(2).pdf.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M., Leary, C., Maclaurin, D., and Wanderman-Milne, S. JAX, 2017–2019. URL https://github.com/google/jax. Specifically the vmap functionality.

Chen, T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. K. Neural ordinary differential equations. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31*, pp. 6571–6583. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/7892-neural-ordinary-differential-equations.pdf.

Dillon, J. V., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., Patton, B., Alemi, A., Hoffman, M., and Saurous, R. A. TensorFlow Distributions, 2017. URL https://arxiv.org/abs/1711.10604.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. URL https://arxiv.org/abs/1408.5093.

Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial evaluation and automatic program generation*. Prentice Hall International, 1993.

Lao, J. and Dillon, J. V. Unrolled implementation of no-u-turn sampler, August 2019. URL https://github.com/tensorflow/probability/blob/master/discussion/technical_note_on_unrolled_nuts.md. Software contributed to TensorFlow Probability as https://github.com/tensorflow/probability/blob/master/tensorflow_probability/python/mcmc/nuts.py.

Looks, M., Herreshoff, M., Hutchins, D., and Norvig, P. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017. URL https://arxiv.org/abs/1702.02181.

Moldovan, D., Decker, J. M., Wang, F., Johnson, A. A., Lee, B. K., Nado, Z., Sculley, D., Rompf, T., and Wiltschko, A. B. Autograph: Imperative-style coding with graph-based performance, 2018. URL https://arxiv.org/abs/1810.08061.

Neubig, G., Goldberg, Y., and Dyer, C. On-the-fly operation batching in dynamic computation graphs. In *Advances in Neural Information Processing Systems*, pp. 3971–3981, 2017.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL http://doi.acm.org/10.1145/1365490.1365500.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

Phan, D. and Pradhan, N. Iterative NUTS, May 2019. URL https://github.com/pyro-ppl/numpyro/wiki/Iterative-NUTS.

Pharr, M. and Mark, W. R. ispc: A spmd compiler for high-performance cpu programming. In *2012 Innovative Parallel Computing (InPar)*, pp. 1–13. IEEE, 2012. URL http://doi.acm.org/10.1145/1133255.1133997.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

The TFP Team. TensorFlow Probability, 2018–2019. URL https://github.com/tensorflow/probability.

The XLA Team. Xla—TensorFlow, compiled, 2017. URL https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html.