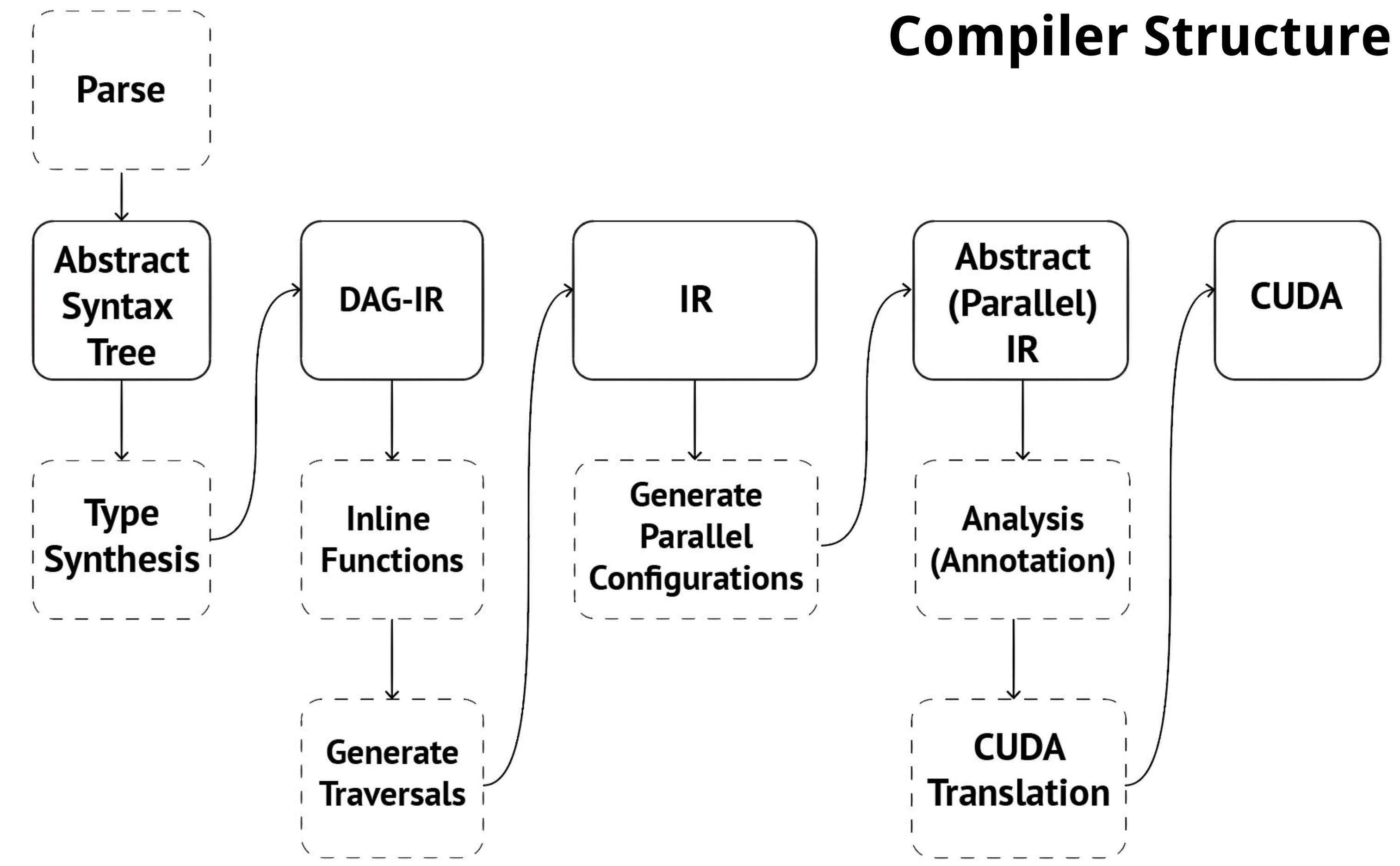
Compiling a high-level language to CUDA

GPUs make parallelism available; CUDA is a popular programming language for harnessing it. Structuring a CUDA program requires a decision: Which parts of a program should be a kernel launch, and which should run on the host? Changing this decision requires re-implementing the partitioning of data among CUDA threads and the copying of data between host and device. Our project automates this process, lowering the barrier for considering many different strategies of parallelization.

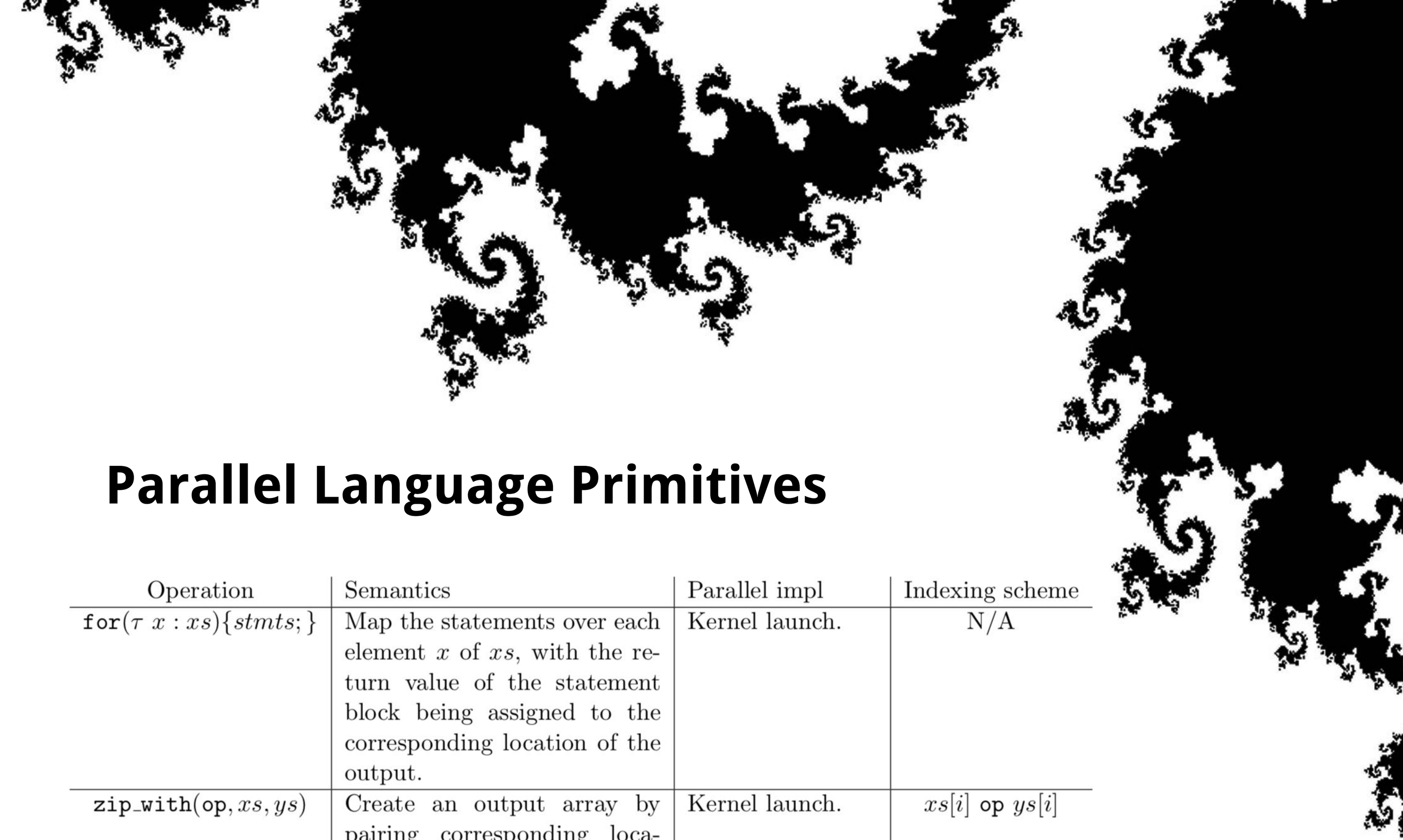
Our Contributions:

- A high-level, data-parallel programming language, **Dag**, for expressing massively parallel computation.
- A compiler, **dagc**, which generates multiple candidate CUDA programs for a Dag input program.
- **Performance heuristics** to select the best CUDA program from the candidates.
- A **benchmark suite of Dag programs** that we compare against reference C implementations.

Compiler Structure



1. An explicit control flow graph of the input program. The "traversals" (topological sorts) generated from this form license optimizations across kernel boundaries.
2. Generates configurations of structuring the CUDA output, designating different parts of the program as the kernel.
3. Performs static analysis to determine which arrays in the source program do not need to be allocated in the target.



Parallel Language Primitives

Operation	Semantics	Parallel impl	Indexing scheme
`for`$(\tau\ x : xs)\{stmts;\}$	Map the statements over each element x of xs, with the return value of the statement block being assigned to the corresponding location of the output.	Kernel launch.	N/A
`zip_with`(`op`, xs, ys)	Create an output array by pairing corresponding locations in xs and ys and applying op.	Kernel launch.	$xs[i]$ `op` $ys[i]$
`filter_with`(xs, bs)	Create an output array consisting of the members of xs where the corresponding member of bs is set to `true`.	N/A.	N/A
`reduce`(`op`, id, xs)	Combine all elements of xs with `op`, where id is the identity of the operation.	Thrust call.	N/A
`scan`(`op`, id, xs)	Create an output array consisting of the result of reducing every prefix of xs with id and `op`.	Thrust call.	N/A.
`range`(n)	Create an array with elements from 0 to $n-1$.	Kernel launch.	i
`transpose`(xss)	Create the transposition of xss.	Kernel launch.	$xss[j][i]$

Dag Input

```
int[][] matrix_multiply(int[][] A, int[][] B) {
   return for (int[] row : A) {
      return for (int[] col : transpose(B)) {
         return reduce(+, 0, zip_with(*, row, col));
      };
  };
}
```

CUDA Output

```
__global__ void kernel(int *output, int *A, int *B) {
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   int i = (idx / J_N) % A_M;
   int j = idx % J_N;
   int acc = 0;
   for (int k = 0; k < A_N; k++)
      acc += A[i * A_N + k] * B[k * B_N + j];
   output[i * A_N + j] = acc;
}

void matrix_multiply(int *output, int *A, int *B) {
   // Malloc kA, kB, and kOutput on device.
   cudaMemcpy(kA, A, A_M * A_N * sizeof(int));
   cudaMemcpy(kB, B, B_M * B_N * sizeof(int));
   kernel<<<dimGrid, dimBlock>>>(kOutput, kA, kB);
   cudaMemcpy(output, kOutput, A_M * B_N * sizeof(int));
}
```

CUDA Output (Sequential)

```
void matrix_multiply(int *output, int *A, int *B) {
for (int i = 0; i < A_M; i++) {
   for (int j = 0; j < B_N; j++) {
      int acc = 0;
      for (int k = 0; k < A_N; k++)
         acc += A[i * A_N + k] * B[k * B_N + j];
      output[i * A_N + j] = acc;
     }
   }
}
```

Heuristics

- Penalize sequential for loops on the host
- Penalize memory transfers
- Penalize allocation on the device

Conclusions

Strengths of the compiler:
- Compiles regular, embarrassingly parallel workload into efficient CUDA programs.
- Small, succinct syntax, and requires low programmer effort.
- Suits some complex workloads.

Areas for improvement:
- Use additional target knowledge to perform program-specific optimizations.
- Memory effects and irregular workload will still require knowledge to optimize.

Benchmarks & Speedups