1. Resumen Ejecutivo

Durante el Sprint 0 (20 h) nos enfocamos en definir cómo debería verse FeCUDA antes de que existan archivos o código. Partimos desde la referencia funcional de `forgeffects` en Python y esbozamos la traducción al ecosistema CUDA, pensando en futuros componentes como `CMakeLists.txt`, `src/main.cu` o una API REST. El resultado es un conjunto de lineamientos sobre qué debe entregar el sistema, cómo se relacionarán los módulos y qué riesgos debemos considerar incluso antes de escribir la primera línea.

2. Arquitectura y Flujo de Datos

El flujo propuesto contempla tres tramos:

a) **Carga y preparación**: se planifica un ejecutable principal que lea matrices tridimensionales (derivadas de los datos `CC/CE/EE`) y las convierta en una estructura común tipo `TensorResult`. Este código viviría en un futuro `src/main.cu` y utilizaría utilidades declaradas dentro de `include/utils.cuh` una vez se creen.

b) **Procesamiento GPU**: la función central `iterative_maxmin_cuadrado` se implementará en `src/algorithms/` y orquestará kernels para bootstrap, maxmin, prima e índices. Todos compartirán el modelo `TensorResult`, diseñado en `include/core/types.cuh` para manejar memoria host/GPU.

c) **Exposición de resultados**: un binding C++ entregará JSON a una API FastAPI y a un frontend estático. Estos futuros archivos residirán en `services/effects_api.py`, `services/effects_api_binding.cpp` y `services/web/`.

En paralelo se proyecta un sistema de benchmarking con documentos ADR y experimentos que describan decisiones como un kernel fusionado.

3. Requerimientos y Supuestos

- **Funcionales**: reproducir la lógica de `FE()` y `directEffects()` (según `forgeffects/README.md`) con parámetros `THR`, `maxorder` y réplicas, y entregar caminos y métricas listos para visualizar.
- **Técnicos**: se asumirá un entorno CUDA 13 con GCC 12, bibliotecas mathdx, cuDNN, cuTensor, cuBLAS y NVML, lo que se documentará en el futuro `CMakeLists.txt`. La API necesitará Python 3, FastAPI y NumPy.
- **Supuestos**: los datasets `CC/CE/EE` estarán disponibles en formato compatible (por ejemplo, `.npy` o `.txt`); se podrán generar validadores que comparen JSON estructurados; los equipos contarán con herramientas `ncu/nsys` para medir rendimiento.

4. Hallazgos Clave por Módulo

- **Core/Tipos**: necesitamos una estructura `TensorResult` que encapsule dimensiones y ownership para evitar fugas al mover datos entre host y GPU.
- **Utilidades**: las funciones de IO y chequeo de CUDA deberán ofrecer mensajes claros y opciones “seguras” para pruebas sin detener el proceso completo para asi detectar errores fácilmente.

- **Algoritmos**: `iterative_maxmin_cuadrado` será el cerebro; deberá validar thresholds y órdenes antes de lanzar kernels, y mantener copias host para serializar resultados.
- **Servicios**: la API REST cargará dinámicamente la librería compartida y expondrá un endpoint /effectsçon métricas simples (tiempo total, memoria GPU, réplicas). El frontend mostrará chips y tablas por orden.
- **Validación**: se planifica un script en `validation/validation.py` que compare archivos JSON contra referencias, realizando comparativas para asegurar el rendimiento del algoritmo al haber cambios.
- **Benchmarking**: se establecerá una metodología de benchmarking robusta que mida el rendimiento del algoritmo.

5. Riesgos y Plan de Mitigación

Riesgo	Por qué importa	Mitigación sugerida
Requerimientos difusos	Sin acuerdos tempranos se implementará algo distinto a `forgeffects`	Completar entrevistas y documentos de alcance antes del desarrollo
Entorno CUDA complejo	Configurar CUDA 13+mathdx+NVML puede bloquear nuevas personas	Documentar un paso a paso y evaluar contenedores/AMIs listos
Falta de datasets validados	Sin datos reales no se puede comparar contra Python	Solicitar muestras anonimizadas y crear datasets sintéticos controlados
Validador rígido	Si las rutas quedan fijas será difícil automatizar QA	Diseñar desde el inicio parámetros para carpetas y tolerancias
Benchmarking sin KPIs	No se sabrá si el kernel fusionado aporta valor	Definir KPIs (tiempo, GB/s, ocupancia) y plantillas de captura antes de programar

6. Trabajo Futuro

1. Ejecutar el backlog investigativo (4 semanas × 5 h) para cerrar requerimientos, flujo arquitectónico, plan de datos y plan de mediciones.
2. Redactar los primeros archivos del repositorio (`CMakeLists.txt`, `src/main.cu`, etc.) siguiendo las decisiones documentadas.
3. Implementar el validador y scripts de benchmark desde las plantillas definidas, asegurando rutas configurables.
4. Revisar periódicamente los riesgos para actualizar mitigaciones antes de pasar a Sprint 1.

Enlace al repositorio: `https://github.com/criquelme2003/feCUDA`