UNIVERSIDADE DO VALE DO RIO DOS SINOS - UNISINOS
UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA
NÍVEL MESTRADO PROFISSIONAL

VINÍCIUS GABRIEL LINDEN

ANALISADOR DE NON-VOLATILE MEMORY EXPRESS BASEADO EM FIELD-PROGRAMMABLE GATE ARRAY

São Leopoldo
2020

Vinícius Gabriel Linden

Analisador de Non-Volatile Memory Express Baseado em Field-Programmable Gate Array

Dissertação apresentada como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica, pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade do Vale do Rio dos Sinos - UNISINOS.

Orientador: Prof. Dr. Rodrigo Marques de Figueiredo
Coorientador: Eng. Cassiano Silva de Campes

São Leopoldo
2020

L744a Linden, Vinícius Gabriel.
Analisador de Non-Volatile Memory Express baseado em Field-Programmabre Gate Array / Vinícius Gabriel Linden. – 2020.
99 f. : il. ; 30 cm.

Dissertação (mestrado) – Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Engenharia Elétrica, 2020.
"Orientador: Prof. Dr. Rodrigo Marques de Figueiredo
Coorientador: Eng. Cassiano Silva de Campes."

1. Tecnologia flash. 2. Solid-State Drive. 3. Field-Programmable Gate Array. 4. Edge computing. I. Título.

CDU 621.3

Dados Internacionais de Catalogação na Publicação (CIP)
(Bibliotecária: Amanda Schuster – CRB 10/2517)

RESUMO

A tendência de intensificação do uso e desenvolvimento de SSDs para aplicações diversificadas abre oportunidades de otimizar ou substituir por completo as técnicas limitantes herdadas dos ultrapassados discos rígidos. Desta forma, um comitê de indústrias fabricantes da tecnologia *flash* propôs um novo protocolo aberto chamado *Non-Volatile Memory Express* (NVMe). Este protocolo foi criado com o intuito de reduzir as penalidades de desempenho impostas pelas camadas de *software*, deixando assim o *hardware* mais padronizado, almejando alto grau de paralelismo e desempenho para o acesso às memórias *flash*. Por razão de segredo industrial, os fabricantes da tecnologia não compartilham os detalhes da implementação deste protocolo. Isto acarreta na indisponibilidade de recursos literários recentes neste tópico para a academia, sendo os trabalhos de pesquisa mais relevantes são publicados pela própria indústria. Isto faz com que tanto o SSD quanto o NVMe sejam de complicado alcance aos pesquisadores fora do círculo destas empresas, limitando a contribuição da academia à área. Por este motivo, o projeto apresentado desenvolveu uma solução para habilitar a pesquisa do protocolo NVMe e das características de SSDs, propondo uma plataforma flexível para possibilitar suporte a este desenvolvimento, sendo benéfico tanto para a indústria quanto para a academia. Nesta implementação, lógica programável foi empregada, propondo-se uma plataforma versátil e reconfigurável. Visto a atual situação de estado da arte, não há trabalho no sentido de propor uma plataforma de pesquisa especificamente para o desenvolvimento genérico de SSDs. Este trabalho foi focado na parte de implementação e de teste da Lógica de Tempo Real (LTR) para *hardwares* comerciais. Foram apresentadas soluções para obtenção de dados do protocolo NVMe e habilitação de acesso à memória ao propor-se três sistemas diferentes: captura e armazenamento de campos NVMe; contagem de comandos NVMe por função, com possibilidade para restrições de campos numéricos; e *switch* NVMe, para decisão automática do destino de comandos NVMe — baseados em seu próprio conteúdo. Campos armazenados e número de comandos são analisados através de uma interface externa e todas as configurações são introduzidas em tempo real. Estes sistemas são capazes de reduzir a lacuna semântica entre as novas diretrizes e a disponibilidade de acesso à pesquisa. Esta plataforma não se limita apenas aos grandes fabricantes da tecnologia, mas também permitirá assim que o estado da arte seja tangível nas novas linhas de pesquisa almejadas pela comunidade científica.

Palavras-chaves: Tecnologia *flash*. *Solid-State Drive*. *Field-Programmable Gate Array*. *Edge computing*.

ABSTRACT

The usage and development intensification of Solid-State Drive (SSD), in varying applications, open up the optimization possibilities for the legacy techniques acquired from the outdated Hard Disk Drives (HDD). For this reason, a committee made of the leading flash memory technology manufacturers has put forward a new standard for the communication protocol, named Non-Volatile Memory Express (NVMe). This protocol was created with the end goal of suppressing the performance penalties imposed by the software layer, leaving the hardware more standardized with a high degree of parallelism and performance. Due to industrial secret, manufacturers do not share the protocol implementation's details. This makes recent literary resources scarce to the academia, with the most relevant papers published by the industry itself. All of this contributes to turn the SSD development out of reach for the third-party researchers, limiting the academic contribution. For this reason, this project presented a solution for enabling the research of both the NVMe protocol and SSD's characteristics, making a flexible platform for this development: a benefit for both the industry and academia. By employing programmable logic in the project's implementation, greater flexibility and versatility was allowed. The current State of the Art presents one with no research that put forward such a platform exclusively aimed at SSD development and protocol analysis. This work focuses in the implementation and testing of real time logic in commercial hardware. Solutions for acquiring NVMe protocol data and enabling to memory access optimization were presented, in form of three different systems: a field capture and storage; a NVMe command counter by function, capable of restricting its count by numeric fields; and a NVMe switch, which decides where to send the commands based on their own content. Command count and captured fields are visible through an external interface and configurations are introduced at runtime, through the same interface. It is believed that these systems fill the gap between the new guidelines and access availability to research — not only for the main manufacturers, but most importantly for academia.

Key-words: Flash technology. Solid-State Drive. Field-Programmable Gate Array. Edge computing.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

AES	*Advanced Encryption Standard* (Padrão de Criptografia Avançada)
ASIC	*Application Specific Integrated Circuit* (Circuito Integrado para Aplicação Específica)
BER	*Bit-Error Rate* (Taxa de Erro de Transmissão de *Bits*)
BLAST	*Basic Local Alignment Search Tool* (Ferramenta de Procura de Alinhamento Básico Local)
BRAM	*Block* RAM (RAM interna ao FPGA)
BlueDBM	*Blue Database Machine* (Arquitetura de Armazenamento para Bases de Dados)
CFS	*Cluster File System* (Sistema de Arquivos Resistente a Falhas)
CISC	*Coordinating Intelligent SSD and CPU* (Coordenação para SSD Inteligente e CPU)
CPU	*Central Processing Unit* (Unidade de Processamento Genérico)
DDR	*Double Data Rate* (Técnica de Duplicação de Transmissão)
DRAM	*Dynamic-RAM* (Memória RAM Volátil)
DSM	*DataSet Management* (Informações Pertinentes ao Gerenciamento)
EXT4	*Fourth Extended File System* (Quarto Sistema de Arquivos Estendido)
FIFO	*First-In First-Out* (Fila de Prioridade de Chegada)
FMC	*FPGA Mezzanine Connector* (Conector Mezanino de FPGA)
FPGA	*Field-Programmable Gate Array* (Dispositivo de Lógica Programável)
FSM	*Finite State Machine* (Máquina de Estados Finita)
FS	*File System* (Sistema de Arquivos)
FTL	*Flash Translation Layer* (Camada de Tradução de Memória *Flash*)
GC	*Garbage Collection* (Algoritmo de Apagamento em Memória)
GPIO	*General Purpose IO* (IO de Uso Genérico)

HDD	*Hard Disk Drive* (Unidade de Disco Rígido)
HLS	*High-Level Synthesis* (Síntese de Abstração de Alto Nível)
HPC	*High Pin Count* (Alta Densidade de Pinos)
IEEE	*Institute of Electrical and Electronics Engineers* (Instituto de Engenheiros Eletricistas e Eletrônicos)
IOSR	*International Organization of Scientific Research* (Organização Internacional de Pesquisa Científica)
IO	*Input-Output* (Entrada-Saída)
IP	*Intellectual Propriety* (Propriedade Intelectual)
JEEE	*Journal of Electrical and Electronics Engineering* (Jornal de Engenharia Elétrica e Eletrônica)
LBA	*Logical Block Address* (Endereço do Bloco Lógico)
LPC	*Low Pin Count* (Baixa Densidade de Pinos)
LTR	Lógica de Tempo Real
MLC	*Multi Level Cell* (Célula a Dois Estágios)
MRAM	*Magnetoresistive RAM* (RAM Magnetoresistiva)
MST	*Minimum Spanning Tree* (Árvore de Extensão Mínima)
NVMe	*Non-Volatile Memory Express* (Protocolo de Comunicação para Memórias Não-Voláteis)
NVRAM	*Non-Voltiale RAM* (Memória RAM Não-Volátil)
OMP	*Orthogonal Matching Pursuit* (Técnica de Decomposição em Múltiplas Funções)
P/E	*Programming Erasure* (Ciclo de Programação/Apagamento)
PASBPI	*Page Allocation Scheme-Based Program Interference* (Interferências de Programação na Alocação de Página Baseado em Esquema)
PBA	*Physical Block Address* (Endereço do Bloco Físico)
PCI	*Peripheral Component Interconnect* (Interface Paralela de Média Velocidade)

PCIe	*PCI Express* (Interface Serial de Alta Velocidade)
PM	*Persistent Memory* (Memória Persistente)
QAM	*Quadrature Amplitude Modulation* (Modulação de Amplitude em Quadratura)
QLC	*Quad Level Cell* (Célula a Quatro Estágios)
RAID	*Redundant Array of Independent Disks* (Sistema de Gerenciamento de Múltiplas Unidades de Armazenamento)
RAM	*Random Access Memory* (Memória de Acesso Aleatório)
RBER	*Raw Bit-Error Rate* (Taxa de Erro de Transmissão de *Bits* Antes da Correção)
RFR	*Retention Failure Recovery* (Recuperação de Falha de Retenção)
ROR	*Retention Optimized Reading* (Leitura para Retenção Otimizada)
RRAM	*Resistive RAM* (RAM Resistiva)
RTR	*Run-Time Reconfiguration* (Método de Reconfiguração de FPGA Enquanto em Tempo de Execução)
SATA	Serial AT Attachment (Especificação de Comunicação Serial com Placa-mãe AT)
SDF	*Software-Defined Flash* (Unidade de Armazenamento *Flash* Definida por *Flash*)
SHA	*Secure Hash Algorithm* (Função de Dispersão Criptográfica Baseado em *Hash*)
SLBA	*Start of Logical Block Address* (Começo do Endereço do Bloco Lógico)
SLC	*Single Level Cell* (Célula a Único Estágio)
SMA	*SubMiniature version A* (Conector Miniatura Coaxial)
SQL	*Structured Query Language* (Linguagem de Consulta Estruturada)
SSD	*Solid-State Drive* (Unidade de Estado Sólido)
ST-RAM	*Spin-Transfer Torque MRAM* (MRAM a Transferência de *Spin*)
SoC	*System-on-Chip* (Sistema em Chip)
TLC	*Triple Level Cell* (Célula a Três Estágios)

UFIA	*Updating Frequency based Inode Aggregation* (Agregação de Inode Baseada em Frequência de Atualização)
VHDL	*Very High Speed Integrated Circuit Hardware Description Language* (Linguagem de Descrição de *Hardware* a Alta Frequência)
WAF	*Write Amplification Factor* (Fator de Amplificação de Escrita)

LISTA DE SÍMBOLOS

T	Transferências
RPM	Rotações Por Minuto

SUMÁRIO

1 INTRODUÇÃO

Memórias digitais nos últimos anos vêm se desenvolvendo em três aspectos principais: menor custo; maior velocidade de banda; e maior capacidade de armazenamento. A principal tecnologia atualmente em uso, a Unidade de Disco Rígido, ou *Hard Disk Drive* (HDD) em inglês, consiste em um disco magnético rotacionando em alta velocidade, provendo armazenamento a baixo custo e alta capacidade. Porém, dentre os efeitos indesejados causados pela parte mecânica, está a limitação da velocidade de banda. Uma das alternativas à tecnologia citada é a Unidade de Estado Sólido, *Solid State Drive* (SSD). Pela natureza da tecnologia, a SSD supera a HDD em velocidade de banda e, com o avanço das pesquisas na área, tende a ultrapassá-la em custo e capacidade de armazenamento, viabilizando seu uso comercial disseminado. Para aproveitar ao máximo a tecnologia SSD, criou-se o Protocolo de Comunicação para Memórias Não-Voláteis, *Non-Volatile Memory Express* (NVMe), que desbloqueou parcialmente as limitações de banda previamente impostas em *software*. Não obstante, a tecnologia *flash*, empregada em SSDs, pode ser ainda mais aperfeiçoada. Para tanto, faz-se necessário um dispositivo que implemente uma plataforma para o estudo do tráfego de informações e possível otimização de acesso.

Quando se deseja otimizar o processamento de algum algoritmo específico, uma boa solução é a implementação deste algoritmo em *hardware*, pois tais implementações apresentam uma banda superior ou igual às implementações equivalentes em *software*. O *hardware* é planejado especificamente para a tarefa, significando que este pode executar múltiplos passos em um ciclo de *clock*, em um grau de paralelismo adequado — além da possibilidade de implementação de lógica assíncrona. Por conseguinte, é dito que o *hardware* provê a chamada "velocidade de fio" (*wirespeed*), significando uma velocidade de banda próxima à capacidade teórica. Outrossim, para execução em *hardware* é comum utilizar Dispositivos de Lógica Programável, ou *Field-Programmable Gate Array* (FPGA) em inglês. FPGAs permitem a implementação reprogramável de um *hardware* digital — diferentemente do Circuito Integrado para Aplicação Específica, *Application Specific Integrated Circuit* (ASIC), incapaz de reprogramação. Portanto, FPGAs estão sendo cada vez mais utilizados para acelerações. Um primeiro caso de uso de aceleração de algoritmos específicos por *hardware* são os *racks* da Microsoft™, que empregam FPGAs para a aceleração da sua ferramenta de busca na Internet (MCMILLAN, 2014). Outros exemplos serão providos no capítulo de Estado da Arte.

O presente projeto de pesquisa propõe uma combinação da tecnologia *flash* emergente, em formato SSD, com lógica programável. Este projeto deverá ser implementado futuramente em FPGA. Para isto, será proposta uma *bridge* de comunicação entre *host* e SSD. Uma *bridge* é qualquer dispositivo eletrônico que governa o fluxo de dados entre os pontos que se conecta, sendo capaz de observar o tráfego, identificando informações relevantes. Já um *host* define-se por qualquer servidor, computador ou qualquer outro dispositivo eletrônico capaz de se comunicar

com a *bridge* através de sua interface respectiva, implementando as funções necessárias de emissão e recepção de dados, através de seu sistema operacional. O *host* perceberá a *bridge* conectada como uma SSD. A implementação desta *bridge* em *hardware* dar-se-á pelo uso da placa de avaliação FPGA. Uma placa de avaliação FPGA define-se por ser pré-projetada e fornecida por terceiros, normalmente sendo os próprios fabricantes do FPGA, com o objetivo de desenvolvimento de projetos em plataforma FPGA, tendo como principal aspecto o seu reuso e versatilidade. Inicialmente, a função principal desta *bridge*, será de permitir a comunicação direta entre *host* e SSD, sem alterações dos dados, com a leitura de certos campos relevantes para diferentes características de cargas de trabalhos (*workload*) para as memórias. Tal *bridge* poderá coletar informações em tempo real sobre o protocolo NVMe, podendo gerar futuramente estatísticas relevantes a outros pesquisadores. Uma comunicação externa com um microcontrolador deverá existir, sendo que este irá capturar as informações geradas pelo FPGA. Chama-se esta *bridge* de habilitadora, pois ela habilitará: pesquisas sobre tráfego entre ambos os pontos, podendo gerados trabalhos científicos sobre os padrões de acesso do SSD, sobre a estrutura dos dados que transitam, entre outros; e a futura otimização do acesso. Estão previstas como contribuições científicas deste trabalho:

- Planejamento de uma arquitetura flexível de *bridge* para SSDs;
- O desenvolvimento desta *bridge*;
- Habilitar a pesquisa sobre o tráfego de dados entre *host* e SSD;
- Possibilitar a otimização de acesso à memória;
- Desenvolvimento de um ambiente de simulação de partes internas da *bridge*;
- Especificação do *hardware* futuramente empregado;
- Disponibilização dos códigos em uma plataforma aberta à pesquisadores;
- Habilitar projetos futuros, que poderão basear neste.

Definições adicionais serão necessárias para o entendimento do texto a vir, são elas: metadado, aplicativo, aplicação e módulos. Primeiramente, um metadado significa qualquer descrição sobre características de um dado, por exemplo: tamanho, tipo de arquivo, data de criação, proprietário, entre outros (DICIONÁRIO PRIBERAM, 2019b); a segunda definição é a de aplicativo, sendo um programa informático que visa facilitar a realização de uma tarefa em um dispositivo qualquer (DICIONÁRIO PRIBERAM, 2019a); A terceira definição trata-se do significado de aplicação, que equivale à camada de abstração — que permite transmitir informações de um nível menor adaptando-as para uma abstração de nível mais alto e vice-versa. Por último, a definição de módulos, que são “blocos” de funções únicas que constroem qualquer sistema de Lógica de Tempo Real (LTR). São exemplos destes módulos: instanciações

de primitivas, gerenciador de relógio (*clock*), *buffers* de dados, serializadores, entre outras implementações. Módulos podem ser feitos de outros módulos, não havendo qualquer restrição.

O texto está organizado por capítulos, sendo: Conceitos Básicos, que apresentará o conceitos para a boa compreensão do trabalho; após, o Estado da Arte, que dissertará sobre trabalhos relacionados; em seguida, na Metodologia, serão apresentados, em ordem, o método empregado, os requisitos impostos, o sistema proposto e os critérios de avaliação; em seguida, na Análise de Resultados, serão descritos conceitos desenvolvidos para a implementação do sistema proposto, após, serão apresentadas as três soluções aos objetivos descritos, comprovando o funcionamento destas; por último, o capítulo de Conclusão tratá uma reflexão final, analisando a observância aos objetivos conforme as métricas impostas, terminando por sugestões de trabalhos futuros. Apêndices de auxílio ao texto serão apresentados. Terminado este capítulo de Introdução, iniciar-se-á pelo capítulo de Conceitos Básicos, que trará fundamentos para a boa compreensão do projeto.

2 CONCEITOS BÁSICOS

Este capítulo desenvolve conceitos fundamentais para a compreensão da problemática resolvida com o projeto proposto. São revisados as interfaces de portas legado e atuais, em seguida são tratadas a camadas de comunicação entre a memória *flash* e um *host*, após será descrita a memória *flash* em si, continuada por sua aplicação para armazenamento genérico. Também é revisto o protocolo NVMe utilizado junto com o conceito básico de *Field-Programmable Gate Array*. Inicia-se pelo conceito de Interfaces Seriais e Paralelas, buscando entender suas limitações e o motivo de sua evolução.

2.1 INTERFACES SERIAIS E PARALELAS

Interfaces digitais seriais possuem uma única via para comunicação, assim enviando em série, uma sequência de *bits*. Entretanto, as interfaces paralelas são diferentes, pois enviam uma palavra inteira em paralelo para cada ciclo de *clock* (no caso de taxa de transferência única). Isto significa que cada *bit* da palavra necessita de uma via própria, trazendo vantagens e desvantagens. No Quadro 1 são apresentadas algumas das vantagens de cada tipo de interface. É possível observar que, a paralela possui uma maior taxa de transmissão de dados, pois envia uma palavra ao invés de *bits*. No entanto, este paralelismo apresenta um problema de sincronia para frequências mais elevadas, limitando a interface paralela a baixas frequências.

Quadro 1 – Comparativo das interfaces serial e paralelo

Interface	**Paralela**	**Serial**
Vantagens	Maior taxa de transmissão de dados para a mesma frequência	Custo reduzido, simplicidade técnica, menor quantidade de pinos, interligação de múltiplos circuitos na mesma interface
Exemplos	ATA, SPI, PCI, PCI-X ...	SAS, SATA, I2C, CAN, PCIe, ...

Fonte: Autor.

Alguns dos motivos para os quais portas seriais tradicionais evoluíram de antigos padrões de portas paralelas são apresentados na lista:

- Portas paralelas precisam sincronizar a transmissão entre *bits* individuais. Para elevadas frequências, isto se torna um problema de sincronia;
- *Crosstalk* mais relevante para interfaces paralelas, dado que há um número maior de trilhas;

- A corrupção em um único *bit* significa a perda da palavra por completo.

Em geral isto significa que, quanto maior a frequência, mais vantajoso é o uso de interfaces seriais. Como o esforço nos últimos anos foi de elevar a frequência, ao mesmo tempo que se diminui os custos, este fato teve como consequência inevitável uma transição de comunicação paralela para serial. Dentre as interfaces seriais que se destacam na transmissão de dados para meios de armazenamento, estão elas: a SAS e a SATA. Ambas são interfaces seriais de comunicação ponto a ponto (*point-to-point*), que cumprem um mesmo objetivo tecnológico, começando pela SATA.

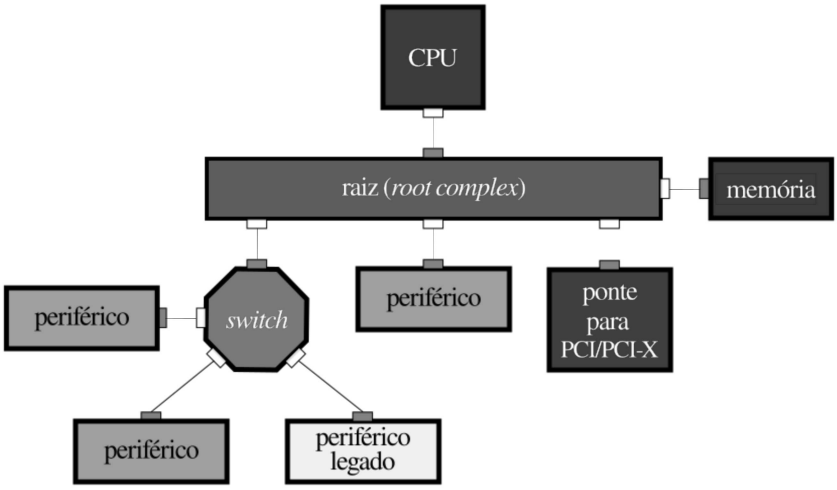
Serial AT Attachment (SATA) é uma interface tradicional para comunicação entre *host* e periféricos. Anunciada no ano 2000 como uma interface padrão, superando a prévia *Parallel AT Attachment*, ou simplesmente *AT Attachment*, em custo e capacidade de transmissão. O grupo que coordena a especificação da SATA é *Serial ATA International Organization* SATA-IO (2019b), que é composta por vários membros, incluindo Hewlett Packard, Intel, SanDisk, Seagate Technology dentre outros. Já a *Serial Attached Small Computer System interface* (SAS) é uma competidora da SATA, ocupando o mesmo nicho tecnológico. SAS foi anunciada no ano 2004, sendo sua antecessora a *Parallel Small Computer System interface*, também uma interface paralela. O grupo que rege a especificação da SAS é o Incits (2019), composto por mais de 1700 empresas, consórcios e universidades. Ambas as interfaces SATA e SAS citadas possuem capacidade para troca a quente (*hot swapping*), o que suas antecessoras paralelas não possuíam.

As especificações no ano regente de 2019 estabelecem uma transferência de dados de 6 Gbps para SATA (SATA-IO, 2019a) e 22,5 Gbps para SAS (KEYSIGHT TECHNOLOGIES, 2019). Para HDDs, estas taxas são suficientes devido a restrições físicas do *hardware* que o impossibilitam de saturar a interface. Já para aplicações rápidas, como SSD, estas interfaces são facilmente saturadas, limitando a capacidade do *hardware*. Devido sua saturação de transferência de dados, elas estão sendo substituídas pelo padrão *Peripheral Component Interconnect Express* — de banda superior.

2.2 *PERIPHERAL COMPONENT INTERCONNECT EXPRESS*

A *Peripheral Component Interconnect Express* (PCIe) é uma interface de comunicação de múltiplas vias seriais, em paralelo - combinado assim o melhor de cada tipo de interface em uma só. Esta combinação traz benefícios de ambas, além de prover um benefício adicional: a flexibilidade de seleção de capacidade de transmissão de dados. Como cada via opera de maneira independente, quanto mais vias utilizadas, maior é a taxa de transmissão. A Figura 1 apresenta um exemplo de topologia para uma rede PCIe. Este exemplo apresenta todos os elementos de uma rede PCIe, sendo: raiz, *switch* e periféricos, além das pontes para outros protocolos.

Figura 1 – Exemplo de topologia PCIe



Fonte: Adaptado de Solomon (2019).

O grupo PCI-SIG (2019b) é responsável pela especificação, sendo coordenado por diretores de empresas como AMD ®, Dell®, NVIDIA®, Keysight®, Hawlett Packard®, entre outras. Suas antecessoras paralelas, a PCI e a PCI-X também foram especificadas por PCI-SIG. PCIe é uma especificação em desenvolvimento ativo, cuja última atualização foi lançada em 2017 (4ª geração). A quinta geração já foi liberada para acesso aos membros a partir de 2019 (PCI-SIG, 2019a). Cada nova especificação dobra a capacidade de transmissão de dados de sua antecessora, desde seu lançamento. Confira a evolução da PCIe no Quadro 2.

Quadro 2 – Dados sobre PCIe

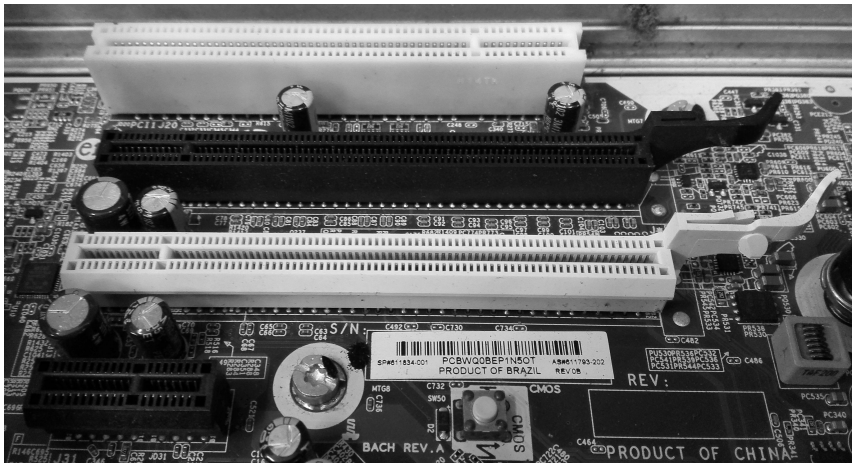
Geração	**Lançada**	**Taxa de transmissão por quantidade de vias**				
		x1	**x2**	**x4**	**x8**	**x16**
1.0	2003	250 MBps	0,5 GBps	1 GBps	2 GBps	4 GBps
2.0	2007	500 MBps	1 GBps	2 GBps	4 GBps	8 GBps
3.0	2010	984,6 MBps	1,97 GBps	3,94 GBps	7,88 GBps	15,8 GBps
4.0	2017	1969 MBps	3,94 GBps	7,88 GBps	15,75 GBps	31,5 GBps
5.0	2019*	3938 MBps	7,88 GBps	15,75 GBps	31,51 GBps	63 GBps

* Anunciada no final de 2017, liberada para membros em 2019.

Fonte: Autor, informação retirada de PCI-SIG (2019b).

Conectores padrão fêmeas são chamados de *slots*; enquanto que machos, de *edge connectors*. É possível utilizar um número de vias inferior ao disponível - por exemplo, um periférico de quatro vias em um soquete de oito. Alguns *slots* menores que *x16* possuem um lado aberto para o encaixe de *edge connectors* maiores (Anexo A). A Figura 2 apresenta conectores PCI, PCIe fêmeas em uma placa mãe.

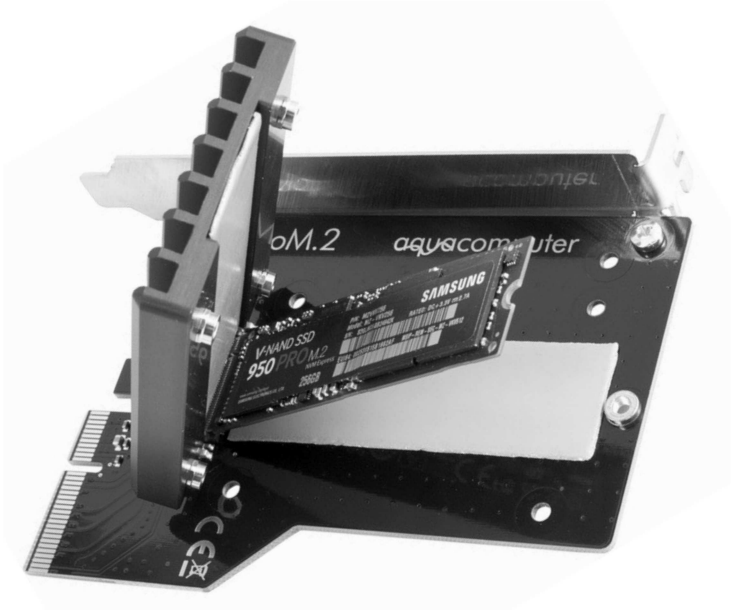
Figura 2 – Conectores padrão (fêmeas): PCI, dois PCIe (x16) e PCIe (x1)



Fonte: Autor.

A especificação PCIe também permite o uso de outros conectores, como o M.2 de menor tamanho, usado em computadores fixos e portáteis (*laptops*). A Figura 3 mostra uma placa adaptadora de conector M.2 para PCIe.

Figura 3 – Adaptador conectores de M.2 para PCIe macho, contendo um SSD



Fonte: Adaptado de Aqua Computer (2019).

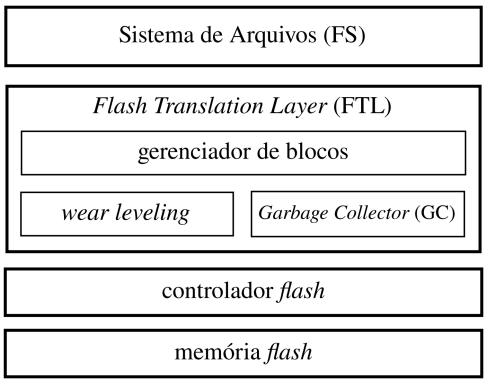
A especificação PCIe é compatível com suas versões anteriores (*backward compatible*) - por exemplo, em um computador que suporte PCIe Gen 5, também é possível utilizar um periférico PCIe Gen 1, sem nenhuma restrição de uso. Sua arquitetura é especificada como uma camada física, seguida de camadas de Enlace de Dados e de Transação. Dada um uso qualquer, deverão haver mais camadas específicas à implementação. Esta padronização PCIe é utilizada

para SSDs NVMe, já que permite uma transmissão de dados superior aos protocolos anteriores (SAS e SATA). Para concluir sobre a necessidade de uma alta taxa de transmissão de dados, é necessário entender a memória *flash* em si, começando por suas camadas de comunicação.

2.3 CAMADAS DE COMUNICAÇÃO COM A MEMÓRIA

Para um *host* se comunicar com uma memória, é necessário atravessar uma sequência de camadas, cada qual com uma tarefa específica. Estas servem para simplificar a implementação, pois caso não existissem, cada aplicativo teria de implementar-se em conjunto o *driver* de comunicação com o periférico. A construção por camadas simplifica em muito a implementação de aplicações, pois cria uma interface comum para tudo que interaja com a memória - ou qualquer outro periférico. Esta arquitetura por camadas não é exceção para memórias, pois outros protocolos, como o de internet, também são projetados da mesma forma. Na Figura 4 é representado o esquema simplificado de comunicação entre um *host* e *flash*.

Figura 4 – Camadas tradicionais de comunicação entre *host* e *flash*



Fonte: Adaptado de IBM Developper (2019).

Funções do sistema (*system calls*) permitem a interação entre aplicativo e Sistema de Arquivo.

2.3.1 SISTEMAS DE ARQUIVOS

O Sistema de Arquivos, ou *File System* (FS), é essencialmente encarregado de controlar o fluxo de dados, sendo responsável por recolher e armazenar os dados no dispositivo de armazenamento não-volátil, dividindo o espaço físico do dispositivo de armazenamento em meios lógicos comumente descritos como arquivos, atribuindo-lhes nomes. Pode-se concluir que FS é uma parte essencial do sistema operacional. Sem FS não há como o sistema operacional diferenciar entre arquivos, uma vez que não haveria separação do espaço na memória, logo cada partição necessariamente deve ser organizada por um FS. Alguns sistemas operacionais dispõem de múltiplos FS - tal como os sistemas operacionais baseados em Linux. Pode-se imaginar o FS

como um mapeador entre espaço de acesso do usuário e espaço lógico. Alguns tipos em especial de FS são listados:

- Sistema de Arquivos sensível à *Flash*: não possui acesso direto às memórias *flash* - pois ainda interage com a Camada de Tradução de Memória *Flash* (*Flash Translation Layer*, FTL)[1]. No entanto este FS é projetado considerando as principais características internas da memória *flash* a fim de não desarranjar a organização interna desta, porém não é de uso exclusivo da *flash*, podendo ser utilizado em outros tipos de memória com desempenho menos otimizado.
- Sistema de Arquivos *Flash*: similar ao anterior, porém exclusivamente projetado para memórias *flash*, operando somente para esta tecnologia de armazenamento - incorporando todas as camadas de FTL.
- Sistema de Arquivos com *Journaling*: é um FS que visa garantir a recuperação dos dados em caso de falhas. Isto é realizado com a utilização de um dos dois possíveis métodos: o *full-journaling* ou *ordered journaling*, respectivamente armazenando dados e metadados em uma região reservada da mídia de armazenamento, enquanto que o segundo encarrega-se de armazenar apenas os metadados na região reservada. Uma vez que estes dados são escritos, eles são posteriormente reescritos em sua posição final, permitindo assim que, em caso de falhas, o FS possa recuperar os arquivos na região reservada.
- Sistema de Arquivos com versionamento: permite opcionalmente a capacidade de, ou incorpora nativamente o versionamento de arquivos, significando que o FS mantém diferentes versões do mesmo arquivo simultaneamente.

Estes tipos de FS não são mutualmente excludentes, ou seja, um Sistema de Arquivos *Flash* pode ter *Journaling* e versionamento simultâneo. Alguns dos principais FS estão listados no Quadro 3.

Quadro 3 – Principais Sistemas de Arquivos

Nome	Desenvolvedor	Enfoque
Ext	Rémy Card	uso geral em Linux
Ext2		
Ext3	Stephen Tweedie	
Ext4	comunidade open-source	
JFS	IBM™	uso geral
XFS	Silicon Graphics™ Red Hat™	alto grau de paralelismo
Btrfs	Oracle™	*Copy-on-Write* uso geral em Linux
FAT32	Microsoft™	compatibilidade entre sistemas operacionais

Fonte: Autor.

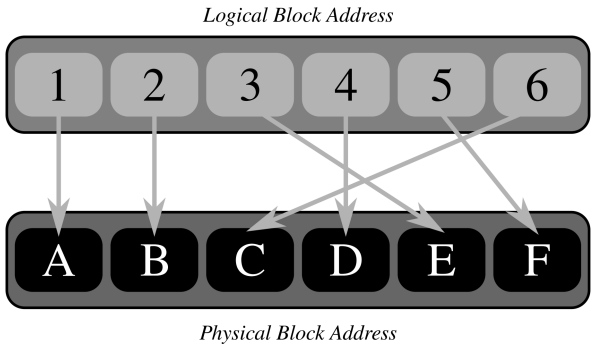
[1] FTL será explicado no texto

Faz-se menção especial ao Ext4 (*fourth extended file system*), que é rápido, moderno, maduro (mais de 10 anos) e completo - utilizado em Android (KANG et al., 2013) e prioritariamente em sistemas operacionais baseados em Linux. Ext4 é aberto (*open-source*) (GITHUB, 2019) (EXT4 WIKI, 2019) e por este motivo é frequentemente modificado para testar alterações decorrentes de pesquisa (ver capítulo de Estado da Arte). Além do FS, há outra camada relevante encarregada de gerenciar o acesso às memórias: a FTL.

2.3.2 *FLASH TRANSLATION LAYER*

A *Flash Translation Layer*, bem como o nome indica, é uma camada específica para memórias *flash*, que traduz o espaço lógico (*Logic Block Address*) (LBA) em espaço físico (*Physical Block Address*) (PBA). Isto é feito através de uma tabela chamada *Logic to Physical page*, que traduz os nomes lógicos aos endereços físicos. Basicamente, ela compatibiliza a tecnologia *flash* com outras tecnologias de armazenamento, como os discos rígidos (ZEEIS, 2019). A Figura 5 ilustra a tabela mencionada.

Figura 5 – Ilustração da tradução do espaço lógico ao espaço físico



Fonte: Adaptado de flashdba (2019).

Em outras palavras, a FTL compatibiliza para o FS a memória *flash* com outros meios de armazenamento legados (HDD), servindo como uma máscara genérica. Do ponto de vista do FS, a FTL tira a totalmente a dependência da tecnologia de armazenamento. A FTL pode ser implementada totalmente no *host*, técnica conhecida como *open-channel*, ao contrário do caso em que o controlador SSD implemente o FTL. A *flash* deve conter no mínimo um sistema de *Garbage Collection*, *wear leveling* e *Error Correction Code* específicos.

2.3.3 *GARBAGE COLLECTION*

Conforme será explicado na seção de Memória *Flash*, é intrínseco da tecnologia NAND a programação por páginas e apagamento por blocos. Entretanto, a operação de apagar por blocos é lenta e caso seja executada em sequência a um outro comando, adicionar-se-ia à latência resultante. Quando se modifica um arquivo, salvando-se o na *flash*, este é primeiramente

armazenado em uma outra página livre, deixando o arquivo original intacto. Sendo assim, páginas contendo versões anteriores de arquivos são desmarcadas como contendo informação, para após ser recolhido o "lixo", este sistema conhecido como "Coleta de Lixo", ou *Garbage Collection* (GC). A operação de apagar é idealmente executada no tempo livre da memória (*off-line*), tempo o qual a memória não está executando nenhuma outra ação. No entanto, esta situação ideal frequentemente não ocorre, precisando ser executada concorrentemente com outras operações de escrita ou de leitura, incrementando-se a latência. Outrossim, a técnica de *over-provisioning* é frequentemente utilizada para ajudar com o GC, deixando reservado blocos de *flash* não acessíveis ao usuário. Contudo, é necessário um sistema que controle o desgastes proporcional das células, implementado com o *Wear Leveling*.

2.3.4 *WEAR LEVELING*

Também, na mesma seção de Memória *Flash* será explicado que a *flash*, por ser baseada em transistores de porta flutuante, possui uma vida útil menor em comparação aos outros meios de armazenamento. Imaginando que existem vários blocos em uma memória *flash*, cada bloco desgastar-se-á em tempos diferentes, proporcionalmente ao seu uso. Por isto, é importante que blocos sejam utilizados em um ritmo similar, a fim de desgastá-los uniformemente. Este sistema é conhecido como *wear leveling*. A técnica de *wear leveling* é necessária para prolongar a vida útil do componente, não utilizá-la é indesejável, já que o componente inteiro iria perder espaço físico de maneira acelerada. Com a utilização desta técnica, esta característica é amenizada. A vida útil é determinada pelo tipo de célula e pela capacidade de correção do ECC.

2.3.5 *ERROR CORRECTION CODE*

O algoritmo de correção de erro (ECC) é empregado em diversos campos da área de tecnologia, principalmente em sistemas de telecomunicação, sempre que a informação é transmitida por um meio ruidoso ou com perdas - todos os meios. Existem inúmeros algoritmos com capacidades de correção diferentes, cada qual para um uso ideal. O Quadro 4 apresenta alguns algoritmos utilizados em *flash* com informações relevantes como o *overhead*:

Quadro 4 – Número de *bits* necessários para ECC comuns em tecnologias *flash*

***Bits* em erro corrigíveis**	***Bits* adicionais necessários em uma NAND *flash* (*overhead*)**		
	Hamming	**Reed-Solomon**	**BCH**
1	13	18	13
2	não disponível	36	23
3		54	39
4		72	52
5		90	65
6		108	78
7		126	91
8		144	104
9		162	117
10		180	130

Fonte: Adaptado de Novotný, Kadlec e Kuchta (2015).

Um erro é considerado quando o *bit* recebido não equivale ao *bit* transmitido. Erros podem ser contabilizados com o *Bit-Error Rate* (BER). Este é definido como a taxa de *bits* com erro após a correção do ECC, dividido pelo total de *bits* transmitidos. Já o *Raw Bit-Error Rate* (RBER) é definido como a taxa de *bits* com erro "crus", ou seja antes de qualquer correção, dividido pelo total de *bits* transmitidos. Logo, é função do ECC diminuir o RBER em BER. São definidas as equações do RBER e BER:

$$\text{RBER} = \frac{\textit{bits} \text{ em erro}}{\text{total de } \textit{bits} \text{ transmitidos}} \tag{1}$$

$$\text{BER} = \frac{\textit{bits} \text{ em erro após correção com ECC}}{\text{total de } \textit{bits} \text{ transmitidos}} \tag{2}$$

O emprego do algoritmo possui um custo adicional em *bits* para serem utilizados para o cálculo da correção. Estes *bits* são armazenados em espaço extra dedicado a este propósito na página da memória *flash*. São fatores que afetam a escolha de ECC: espaço adicional de memória (*overhead*), implementação, complexidade, atraso, capacidade de detecção de erro e capacidade de correção de erro.

2.3.6 *WRITE AMPLIFICATION FACTOR*

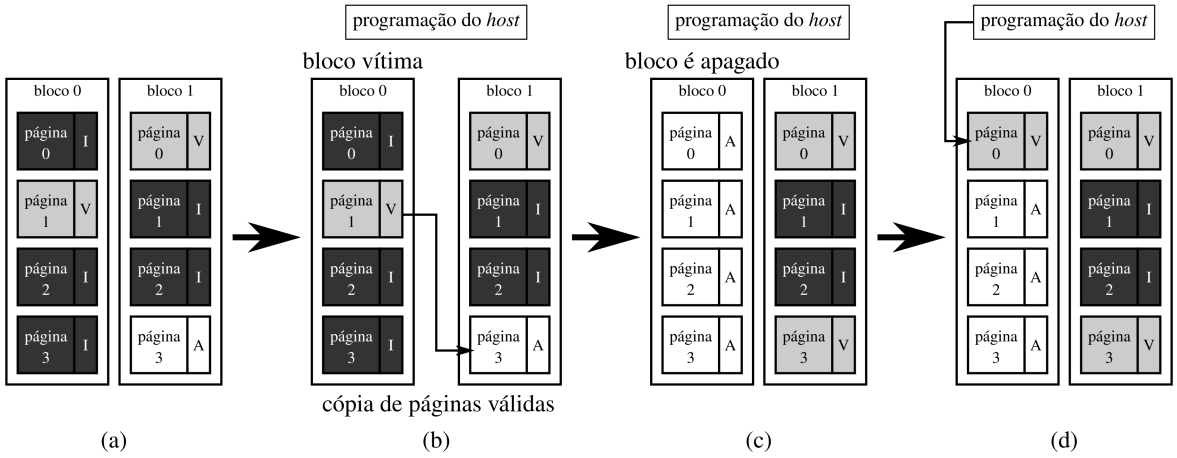
Todas as funções mencionadas anteriormente acrescentam o consumo do espaço de memória e de operação, tanto o ECC, quanto a GC e *wear leveling*. Tal acréscimo é contabilizado em um fator chamado de *Write Amplification Factor* (WAF), que pode ser calculado como:

$$\text{WAF} = \frac{\text{Dados escritos na memória}}{\text{Dados escritos pelo } \textit{host}} \tag{3}$$

Contextualizando, as três operações em *flash*, feitas pelo FS são: programar, invalidar e apagar, sendo esta última somente efetuada por bloco (mais detalhes serão dados na parte de

operação da memória *flash*). Por consequência, uma página pode estar em somente um dos três possíveis estados: válido, significando uma página gravada com dados válidos; inválido, uma página gravada, cujos dados não são mais válidos; e apagado, uma página limpa, desprogramada e disponível. O significado de "válido" ou "inválido" é ditado pelo *host*, sendo: um dado "válido" ainda relevante ao *host*; e "inválido", irrelevante ao *host*. Página invalidadas poderão ser apagadas pelo GC - conforme explicado na parte de *Garbage Collection*. Na Figura 6 estão ilustradas as três operações e estados mencionadas, significando V o estado válido; I, inválido; e A, apagado.

Figura 6 – Exemplo ilustrando operações em *flash*



Fonte: Autor.

Na Figura 6, é possível compreender melhor o conceito de WAF. Inicialmente na Figura 6 (a), as páginas inválidas ocupam somente espaço físico (pois necessitam ser apagadas) e não espaço lógico. Sendo assim, o *host* percebe seis páginas (cinco páginas inválidas mais uma página apagada) disponíveis para serem escritas, embora para o *hardware* elas estejam inválidas - cabendo ao *hardware* realizar o GC como demonstrado na Figura 6 (b). Quando o *host* enviar um novo comando de escrita (sinalizado pelo retângulo "programação do *host*"), o GC terá de apagar o bloco 0 para que mais páginas físicas possam ser escritas. Este processo irá retardar a escrita solicitada pelo *host* pois o GC acaba de entrar em operação. Sendo assim, as páginas válidas do bloco 0 são copiadas para o bloco 1 (apenas uma página neste exemplo), e então a operação apagamento é realizada no bloco 0, ilustrado na Figura 6 (c). Por conseguinte, a escrita solicitada pelo *host* pode então ser realizada e a página é programada, ilustrado na Figura 6 (d). Como se pode perceber, a escrita solicitada pelo *host* acabou impreterivelmente por gerar a operação de GC, o que causou um WAF = 2. Percebe-se que os dados escritos na *flash* nunca serão menores que os dados escritos pelo *host*. Logo o WAF será sempre superior a 1 (100 %) quando a GC deverá acontecer. É interessante manter o WAF menor possível para reduzir a latência total para o *host* ao máximo. Outrossim, o aplicativo desconhece os detalhes do *hardware* devido as camadas de abstração já discutidas. Por consequência, os aplicativos acabam não se preocupando com

características deste, deixando para o *hardware* a execução das tarefas laboriosas responsáveis pelo gerenciamento e acesso às memórias.

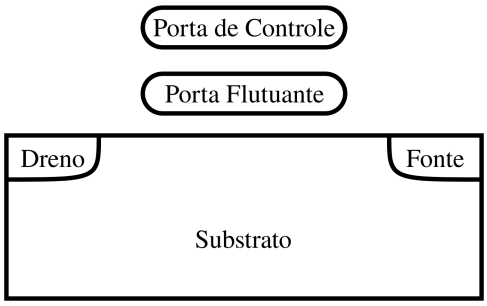
2.4 MEMÓRIA *FLASH*

Memória *flash* é um tipo de memória não-volátil inventada em 1980 pelo japonês Fujio Masuoka, enquanto trabalhava como engenheiro pela Toshiba (MASUOKA; IIZUKA, 1980). O nome *flash* se refere ao *flash* de uma câmera fotográfica, sugerido como nome de batismo a tecnologia de memória, pois a célula executa ações de apagar, ler e programar mais rápido em comparação às tecnologias tradicionais. A primeira memória a utilizar células *flash* foi para implementar uma EEPROM, vindo ao mercado em 1984 (quatro anos após a patente). A fim de simplificar a leitura deste documento, sempre que se mencionar *flash* sem nenhuma palavra qualificadora antecedente a esta (memória, topologia, célula, entre outros), interpreta-se como memória *flash* tradicional. Para explicar como opera a *flash*, é necessário fazer menção à tecnologia de transistor a qual se baseia. Este transistor dá as características deste tipo de memória, podendo ser compreendidas através deste.

2.4.1 TRANSISTOR DE PORTA FLUTUANTE

O transistor de porta flutuante é um tipo de tecnologia de transistor de efeito de campo *Field-Effect Transistor* com duas portas, sendo uma delas isolada eletricamente por uma barreira de óxido de silício (SiO_2), ou qualquer outro material eletricamente isolante, conforme ilustra a Figura 7.

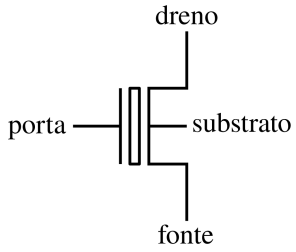
Figura 7 – Ilustração da seção de um transistor de porta flutuante convencional



Fonte: Adaptado de Meliolla et al. (2017).

Na Figura 8 consta o símbolo convencionalmente utilizado em esquemáticos para representar o transistor de porta flutuante.

Figura 8 – Símbolo convencional para a representação do transistor de porta flutuante



Fonte: Adaptado de Brewer e Gill (2007).

A função da porta flutuante é de gerar um campo elétrico próprio, de tal forma que a resultante será a soma dos campos das portas de controle e flutuante. Este *offset* faz com que o fluxo entre dreno e fonte esteja mais ou menos condutivo, sem a intervenção do controle. Uma vez que a carga elétrica está contida na porta flutuante, completamente isolada eletricamente (eis o nome de "flutuante"), esta não se perderá, caracterizando o transistor de porta flutuante como de tecnologia não-volátil. Existem duas técnicas tradicionalmente utilizadas para programação/desprogramação de um transistor de porta flutuante: canal de elétron quente e tunelamento de *Fowler-Nordheim*. Nos próximos dois parágrafos serão explicadas ambas as técnicas mencionadas de injeção de elétrons, também conhecidas como tunelamento - pois criam um "túnel" dentro do isolante para o elétron atravessar.

A técnica de canal de elétron quente (*channel hot electron*, ou *hot-carrier injection*) consiste em impor ao dreno uma tensão relativamente alta (tipicamente entre 4 a 6 V), impondo também uma tensão ainda mais alta à porta de controle (tipicamente entre 8 a 11 V), enquanto que a fonte é aterrada. Isto gera uma corrente significativa (tipicamente entre 0,3 a 1 mA) que acelera os elétrons a uma determinada energia cinética. O campo elétrico é suficientemente forte para atrair elétrons "quentes" (com energia cinética) através da barreira do isolante, sendo presos na porta flutuante - um condutor (BREWER; GILL, 2007).

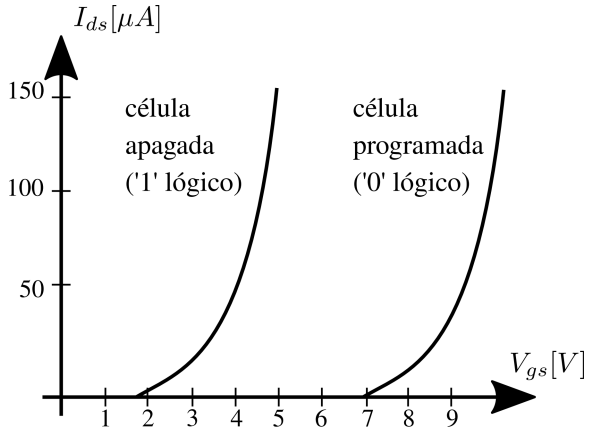
Já a técnica de tunelamento de *Fowler-Nordheim* consiste em aterrar a porta de controle, enquanto que o dreno, fonte e substrato são ligados em alta tensão - ou aterrar o dreno e a fonte enquanto que a porta flutuante é imposta uma tensão negativa. Ou seja, qualquer configuração em que a porta de controle esteja em uma tensão substancialmente menor que a fonte e o dreno. Assim, elétrons no semicondutor serão atraídos pelo campo elétrico em direção à porta flutuante e sendo capturados dentro do isolante. Esta técnica é mais lenta em comparação à anterior.

O isolante é danificado sempre quando elétrons atravessam o isolante. Por este motivo, transistores a porta flutuante possuem uma vida útil mais curta em comparação a outros dispositivos eletrônicos de armazenamento (HDD incluso). Este encurtamento da vida útil é conhecido como desgaste (*wear*) e será uma das consequências da tecnologia *flash*. Outra causa deste encurtamento da vida útil é conforme vários ciclos de Programação/Apagamento (*Program-*

ming/Erasure, P/E) são feitos, elétrons ficam presos permanentemente dentro do óxido, deixando a célula inutilizada (Cai et al., 2017a).

Para a leitura da programação, aplica-se uma tensão quiescente (*bias*) entre porta e fonte, gerando corrente ou não, de acordo com a carga contida na porta flutuante. O valor da corrente é interpretado digitalmente conforme observado no exemplo da Figura 9. Neste exemplo, uma tensão de 4 V deverá resultar em aproximadamente 100 µA se a célula estiver apagada, caso contrário a célula é considerada como programada. Nota-se que há uma tensão entre porta e fonte que o transistor é condutivo independentemente da programação, neste exemplo é de aproximadamente 10 V. É convenção para memórias *flash*, que o estado programado seja o estado lógico '0'.

Figura 9 – Leitura da programação



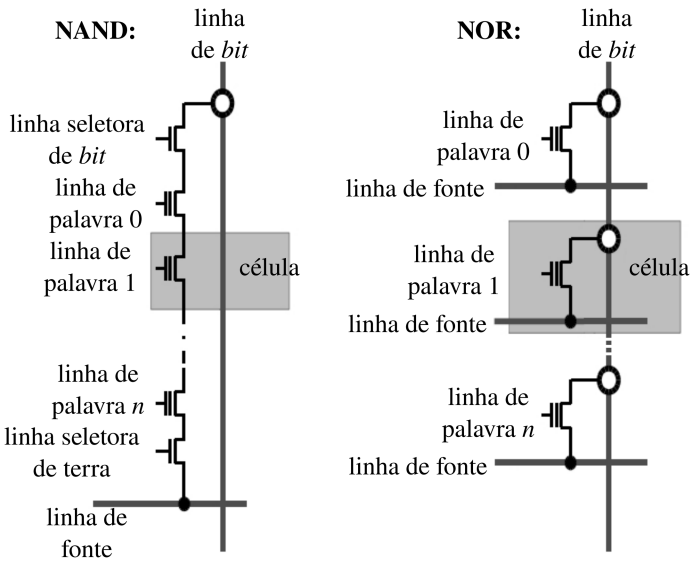
Fonte: Adaptado de Brewer e Gill (2007).

A *flash* é construída por transistores de porta flutuante. Entretanto, é necessário saber como esta memória é estruturada - observando primariamente as diferentes topologias.

2.4.2 TOPOLOGIAS

Existem duas topologias de maior relevância para construir uma memória *flash*: NAND e NOR - nomes que tem origem nas respectivas portas lógicas homônimas. Na Figura 10 são apresentados os esquemáticos de ambas.

Figura 10 – Topologias *flash* NAND e NOR



Fonte: Adaptado de Novotný, Kadlec e Kuchta (2015).

Se faz importante definir alguns termos da terminologia em questão:

- Pacote: Invólucro, também conhecido como "*chip*", é o componente final, contendo um ou mais *dies*;
- *Die*: Semicondutor final, cortado a partir do *wafer*;
- Plano: O *die* é separado no mínimo em dois planos idênticos, contendo múltiplos blocos contidos, bem como a área de controle aos blocos e páginas;
- Bloco: Unidade mínima para apagar, tipicamente contendo entre 512 a 768 páginas; e
- Página: Mínima unidade de programação e leitura, tipicamente contém entre 4 kB a 16 kB.

No Quadro 5 é apresentada uma comparação extensiva entre ambas as topologias. A vida útil e os ciclos de escrita são medidos em ciclos de programação e apagamento (P/E).

Quadro 5 – Comparação entre as topologias *flash*

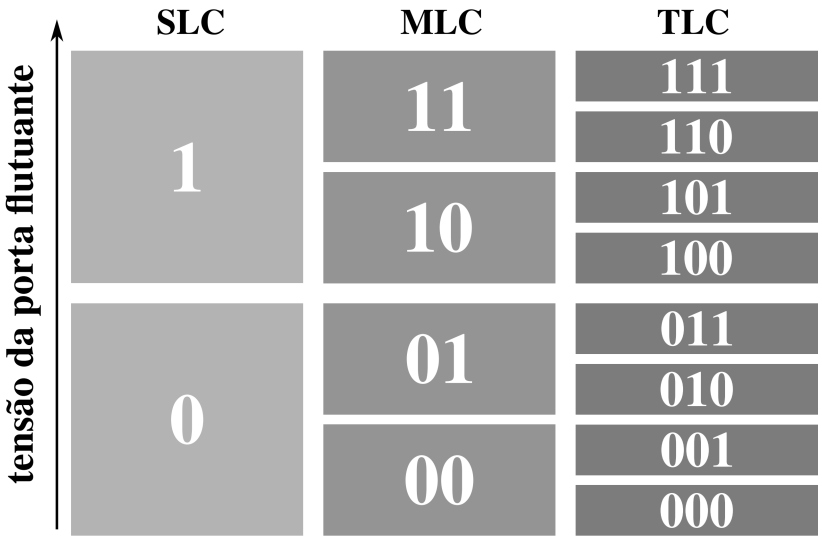
	NAND	**NOR**
Arranjo das células	Série, células adjacentes compartilham a fonte e o dreno	Paralelo, células adjacentes compartilham a fonte
Capacidade	Dezenas de Gb, armazenamento de uso geral	Alguns Gb, armazenamento apropriados para código
Não-volatilidade	Sim, maior densidade de células em comparação à NOR	Sim, maior área por célula
Interface	Interface IO	Interface completa de memória
Acesso de alta velocidade	Sim, acesso aleatório	Sim, acesso serial
Método de acesso	Sequencial	Acesso aleatório de *bytes*
Organização	Por página, apagando por bloco	Por *byte* ou palavra
Desempenho	Leitura sequencial, programação e apagamento rápidos	Leitura aleatória rápida, programação e apagamento lentos
Preço	Menor custo por *bit*	Maior custo por *bit*
Vida útil	10^5-10^6 P/E	10^4-10^5 P/E
Ciclos de escrita	10^6	10^6
Vantagens	Programação e apagamentos rápidos	Acesso aleatório e possibilidade de programação por *byte*
Desvantagens	Acesso aleatório lento, dificuldade de programação por *byte*	Programação e apagamento lentos
Uso típico e aplicações	Armazenamento, voz, dado, gravações de vídeo ou qualquer uso que grave por dados sequenciais	Memória em rede, substituição para EPROM, aplicações com execução diretamente de memória não-volátil

Fonte: Adaptado de Novotný, Kadlec e Kuchta (2015).

Visto que é mais adotada a memória *flash* do tipo NAND para aplicações usuais, este trabalho consagra-se a esta topologia. O texto que segue trata da arquitetura NAND, considerando a NOR somente quando especificada.

Memórias *flash* são notoriamente de baixa densidade (*bit* por área). Na técnica tradicional, a Célula a Único Estágio (*Single Level Cell, SLC*), toda a faixa útil é dividida em dois, representando assim todos os estados binários. Por este motivo, há um esforço para aumentar a capacidade da tecnologia *flash*, evitando adicionar ao custo. Cada célula pode ser discretizada a faixa útil com a finalidade de aumentar a capacidade, inserindo mais *bits*. Assim sendo, foram desenvolvidas as técnicas de Célula a Dois Estágios (*Multi Level Cell*, MLC), Célula a Três Estágios (*Triple Level Cell*, TLC) e Célula a Quatro Estágios (*Quad Level Cell*, QLC), que aumentam a capacidade a um custo de menor confiabilidade e susceptividade à interferências. Sendo TLC, QLC contendo respectivamente três e quatro *bits* por célula. A Figura 11 representa os valores lógicos em respeito aos diferentes níveis de tensão da porta flutuante. Nota-se que QLC não está nesta figura, considerada desnecessária a representação.

Figura 11 – Representação das diferenças entre SLC, MLC e TLC



Fonte: Adaptado de Silicon Power Blog (2019).

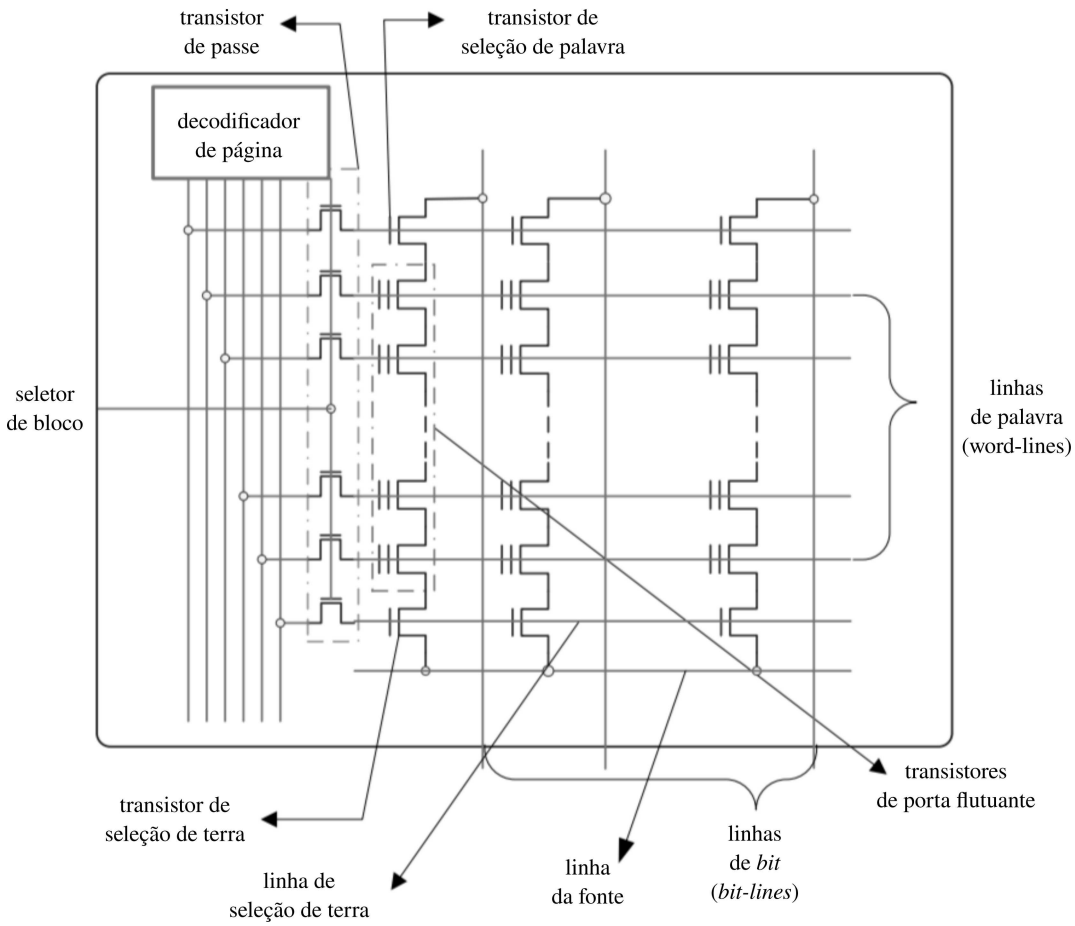
Os valores binários associados a estas faixas de tensão da porta de controle são convenção dos fabricantes. Percebe-se que com a diminuição desta faixa dos estados individuais, diminui também a margem de erro, pois a margem dos níveis de tensão para as células SLC permitem uma maior faixa de erro de leitura. Por isto, para aplicações de segurança (*safety*) ou qualquer outro uso que requer alto grau de confiabilidade, sempre é usado *flash* SLC. A diminuição do custo para uma capacidade maior também acarreta na vida útil menor, pela mesma justificativa.

Às vezes na literatura, MLC tem um significado de qualquer número de *bits* por célula superior a um (SLC). Sendo assim, tanto TLC quanto QLC podem ser chamadas de MLC a três e quatro *bits*, respectivamente. Porém, por fins de evitar confusão neste documento MLC foi estipulada para significar somente dois *bits*, que é o mais usual. Neste caso, são chamadas de não-SLC todas as células que contenham mais de um *bit* - atualmente MLC, TLC e QLC. A distinção entre SLC e não-SLC ficará mais clara uma vez que seja compreendida as operações de uma célula *flash*.

2.4.3 OPERAÇÕES

Em todas as operações, o bloco deve ser primeiramente selecionado através do seletor de bloco. O decodificador de página é o mecanismo responsável por todas as operações nos transistores das linhas de *bit*, também chamados de “braços”. Existem diversos paradigmas de alocação para escrita (*allocation scheme*), dentre eles ordem aleatória, ordem por página e ordem por linha de palavra. A Figura 12 apresenta a estrutura simplificada de uma memória *flash*. Nesta figura, cada página é uma linha de palavra, já o bloco em si é somente o conjunto das páginas apresentadas.

Figura 12 – Esquemático simplificado de um plano NAND *flash*



Fonte: Adaptado de Mohan et al. (2013).

Confere-se no Quadro 6 que técnica é empregada para apagar e programar nas topologias NAND e NOR, utilizado-se sempre a técnica mais rápida e eficiente.

Quadro 6 – Técnicas de tunelamento utilizadas para as topologias NAND e NOR

	NAND	**NOR**
Apagar	*Fowler-Nordheim*	
Programar	*Fowler-Nordheim*	Elétron quente

Fonte: Autor, informação retirada de Mohan et al. (2013).

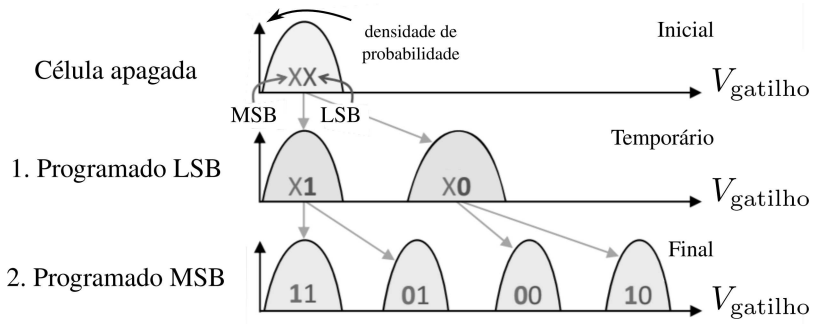
A operação de leitura de um valor de uma *flash* feita em cada braço exige que se aplique o princípio apresentado da porta flutuante. Isto é, aplicar uma tensão quiescente no transistor da linha de palavra e ler o valor da corrente. Porém, a corrente é incapaz de circular sem que todos os outros transistores em série estejam em estado de condução. Com este objetivo, todo outro transistor (de porta flutuante ou não) é aplicada uma tensão de tal forma que se possa conduzir. Desta maneira tanto a leitura quanto a escrita são efetuadas, conforme explicado anteriormente.

Evidentemente, ambas são feitas por páginas, visto que existe uma estrutura de decodificação, que seleciona os *bits*, que formam as palavras.

Conforme comentado na parte de transistor de porta flutuante, para apagar é necessário remover a carga aprisionada dentro do isolante, na porta flutuante. Para isto, operações de apagar são feitas em blocos. Apagar em blocos é mais vantajoso, visto que há um câmbio entre tamanhos de blocos grandes e pequenos. Quanto menor o tamanho do bloco mais fácil e rápido para apagar, entretanto blocos pequenos são mais custosos em termos de área do *die* (BREWER; GILL, 2007). Logo, deve ser encontrado um equilíbrio entre os tamanhos de blocos.

As memórias *flash* tem a tendência de acumular carga na porta flutuante, tendo impacto relevante nas técnicas não-SLC. Logo identificam frequentemente estados erroneamente, tendenciosamente para um estado com maior referência de corrente. Diante disto, fabricantes de *flash* para as tecnologias não-SLC frequentemente utilizam técnicas chamadas programação incremental por pulso (*incremental step pulse programming*), sendo esta técnica para MLC conhecida como programação a dois passos (*two-step programming*). A programação a múltiplos passos cria estágios intermediários de programação, ao contrário da programação a um único passo que leva do estado apagado diretamente ao final. A técnica incremental por pulso é mais controlável que a programação a um passo (*one-shot programming*), pois cada estado possui uma pequena margem de tensão - cada pulso possui uma duração e energia adequada. Na Figura 13 é ilustrada a programação para uma célula MLC. Por convenção, a primeira programação se trata da LSB, seguida da MSB.

Figura 13 – Passos para a programação a dois passos



Fonte: Adaptado de Cai et al. (2017b).

Adiciona-se a estas observações que as células *flash* possuem a particularidade de se auto-interferirem, causando potenciais erros.

2.4.4 ERROS E INTERFERÊNCIAS

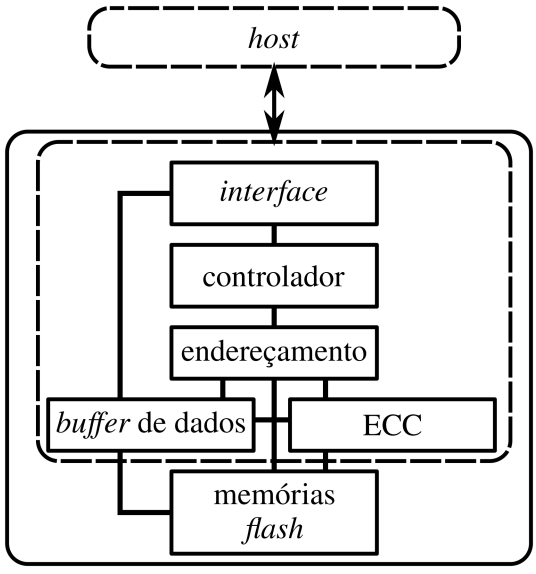
Erros e interferências são lugares comuns para memórias *flash*. Existem majoritariamente dois tipos de auto-interferências: de leitura e de proximidade entre células. A interferência de leitura é um fenômeno intrínseco da arquitetura *flash*, no qual uma leitura efetuada em uma

página dentro de um mesmo bloco pode afetar a densidade de probabilidade de cada estado do mesmo bloco (Cai et al., 2015a). Eventualmente, esta alteração acumula, transformando o estado atual em outro estado errado. Já a interferência de proximidade entre células é causado quando células programadas estão perto da célula "vítima", essencialmente agindo como uma porta flutuante (Lee; Hur; Choi, 2002). Quanto menor a tecnologia de transistor empregada, mais severas são as interferências. Outrossim, células podem ser classificadas de acordo com a capacidade de manter cargas na porta flutuante, como perda rápida ou perda vagarosa (Cai et al., 2015b). Regiões próximas terão características similares (Cai et al., 2017b), logo regiões contendo células de perda rápida terão poucas células de perda vagarosa. Memórias *flash* possuem um grande mercado - mencionado anteriormente. Dentre eles são alguns: *pendrives*, cartões de memória portáteis, memórias embarcadas e SSDs.

2.5 *SOLID-STATE DRIVE*

O *Solid-State Drive* (SSD) é um dos dispositivos de armazenamento baseado em *flash*. Essencialmente ele emprega um conjunto de invólucros de memória *flash*, acoplando-os a um controlador e dando-lhes uma interface para comunicação com o *host*. A interface com o *host* proposta é a mesma que um HDD, mantendo a compatibilidade. Confira na Figura 14 um esquema simplificado de um dispositivo SSD.

Figura 14 – Arquitetura simplificada de um SSD



Fonte: Adaptado de Storage Review (2019a).

O controlador SSD é responsável por:

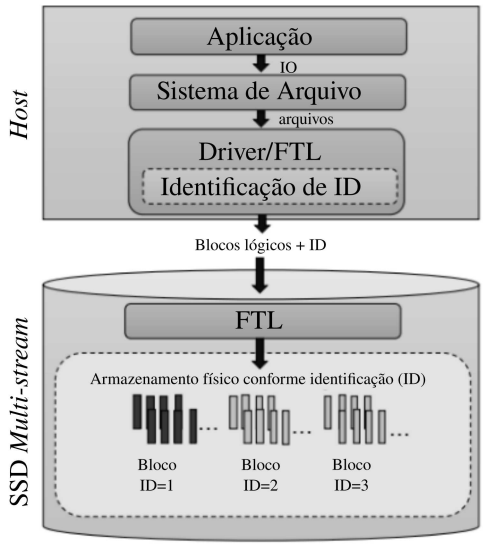
- Realizar operações de leitura, escrita e apagamento nas células;

- Gerir e distribuir a carga aos invólucros individuais (chipes), a fim de não sobrecarregar um invólucro em desfavor de outro, técnica comparável ao *wear leveling* em escala maior;
- Mapear células em falha e substituí-las por células em reserva (comentado no próximo parágrafo);
- Implementar parte ou completamente a FTL;
- GC;
- Algum outro sistema customizado e implementado direto no SSD (ver capítulo de Estado da Arte).

Devido ao super provisionamento de memória (*over-provisioning*) em SSDs, a capacidade total real será sempre maior do que a capacidade oferecida ao usuário. Por exemplo, um dispositivo que ofereça 100 GB de armazenamento poderia ter 10 GB de capacidade extra para operações citadas anteriormente, como GC e *wear leveling*. Outra parte deste super provisionamento é a necessidade de substituição de células falhadas, conforme citado na listagem anterior.

Diferentes dados possuem tempos de vida diferentes. Os dados com tempo de vida mais longo espera-se que se mantenham em sua posição de escrita original, enquanto que dados com tempo de vida mais curtos são frequentemente apagados. Embora essa distinção dos tempos de vida de dados seja bastante útil, os atuais SSDs não podem tirar proveito dessa informação, o que resulta na gravação de dados com tempos de vida distintos nos mesmos blocos da *flash*. Isto resulta em um aumento do GC por haver dados com tempo de vida misturados. A segregação de dados por seus diferentes tempos de vida é chamado de *multi-stream*, sendo cada categoria de dado de *stream*. Existindo-se essa separação por *streams*, o *host* poderá passar esta informação via um identificador (ID) e o FTL então poderá separar os dados em diferentes blocos, pois ele terá o conhecimento que dados em um único *stream* têm características similares. Alguns dos esforços atuais estão trabalhando nesta direção de como elaborar critérios para uma boa categorização (KANG et al., 2014), como será visto no capítulo de Estado da Arte. Esta proposta configura-se como mais um esforço para reduzir o trabalho pesado do SSD por conta das diferentes frequências de acessos aos dados, conferindo-lhe mais autonomia. A Figura 15 ilustra o conceito do sistema *multi-stream*.

Figura 15 – SSD do tipo *multi-stream*



Fonte: Adaptado de Bhimani et al. (2018).

A parte do emprego de novas técnicas como o *multi-stream*, o SSD possui poucas desvantagens em comparação aos outros meios de armazenamento.

2.5.1 VANTAGENS E DESVANTAGENS

Atualmente existem diversos competidores modernos e tradicionais à tecnologia *flash*. Alguns deles são RAM Magnetoresistiva (*Magnetoresistive RAM*, MRAM), RAM Resistiva (*Resistive RAM*, RRAM) e MRAM a Transferência de *Spin* (*Spin-Transfer Torque MRAM*, ST-RAM), para citar os principais. Porém estes ainda são tipicamente menos adotados pela indústria, não sendo aplicados para armazenamento em massa. A tecnologia tradicional HDD é a principal tecnologia competidora à *flash* no sentido em que é utilizada na grande maioria para o armazenamento massivo, tanto em computadores pessoais como para servidores e bases de dado. Além de ser uma tecnologia ultrapassada, a tecnologia HDD foi desenvolvida em 1956 pela IBM™, contendo discos magnéticos que devem girar em altas velocidades - quanto mais rápido, melhor o desempenho. Visto que a agilidade das operações de escrita e leitura está relacionado à velocidade de giro. Tipicamente, este giro está entre 5400 RPM até 15 000 RPM para *laptops* e até 72 000 RPM em servidores (STORAGE REVIEW, 2019a).

O fato que HDD precisa de uma parte móvel em alta velocidade para operar é uma grande desvantagem em comparação à SSD, pois consome muita energia para manter o giro — reduzindo a eficiência energética das operações. Outrossim, o HDD possui um braço móvel que se move sobre o disco em alta rotação, fazendo a leitura com sua ponta sensível ao magnetismo. Já a SSD não necessita este gasto adicional de energia, por ser inteiramente eletrônico. Nesta análise, como a SSD não consome tanta energia quanto o HDD, consequentemente não produzindo tanto

calor - sendo desnecessário sistema de arrefecimento forçado, utilizado em centrais de dados (*data centers*). Outra grande vantagem do SSD sobre HDD é a velocidade das operações, tendo a mesma justificativa à anterior. Confira no Quadro 7 um comparativo entre ambas as tecnologias.

Quadro 7 – Comparativo entre SSD e HDD

	SSD	**HDD**
Consumo	Entre 2 a 3 W, resultando em uma bateria aproximadamente 30 minutos mais longa	Entre 6 a 7 W, consequentemente bateria mais curta
Custo	Cara, aproximadamente 0,20 US$/GB para 1 TB	Muito barata, somente 0,03 US$/GB para 4 TB
Capacidade	Tipicamente menor que 1 TB para *laptops*; no máximo 4 TB para computadores fixos	Aproximadamente entre 500 GB e 2 TB para *laptops*; no máximo 10 TB para computadores fixos
***Boot* do sistema operacional**	Entre 10 a 13 segundos	Entre 30 a 40 segundos
Ruído	Não há partes móveis, logo não há ruído	O disco rotante pode causar ruído
Vibração	Não há partes móveis, logo não há vibração	O disco rotante pode causar vibração
Calor produzido	Não há partes móveis, nem excesso de consumo, logo o consumo é baixo	HDD não produz tanto calor, mas terá um calor adicional significante comparado ao SSD, devido às partes móveis e o maior consumo
Taxa de falha	Tempo médio entre falhas de 2 milhões de horas	Tempo médio entre falhas de 1,5 milhões de horas
Velocidade de programação	Geralmente acima de 200 MBps até 550 MBps para tecnologia de ponta	Entre 50 a 120 MBps
Criptografia	*Full Disk Encryption* suportado em alguns modelos	*Full Disk Encryption* suportado em alguns modelos
Velocidade de leitura de arquivo	Até 30 % mais rápido que HDD	Mais lento que SSD
Afetado pelo magnetismo?	Seguro contra os efeitos do magnetismo	Ímãs podem apagar os dados

Fonte: Adaptado de Storage Review (2019b).

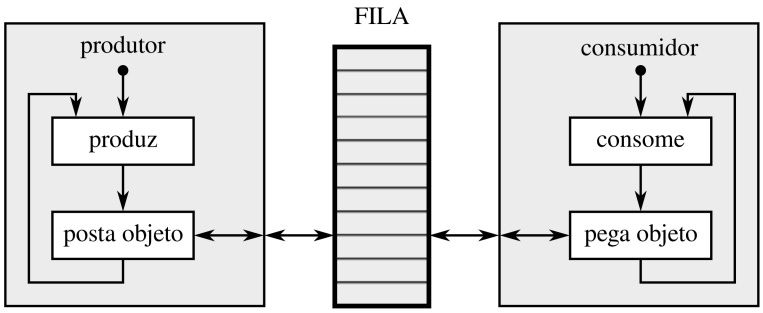
Este Quadro 7 desenvolve comparativamente as principais vantagens do SSD: seu baixo consumo, resistência ao magnetismo e alta velocidade de acesso aos arquivos. Percebe-se neste quadro que em prática, a única vantagem do HDD sobre o SSD é sua capacidade total de armazenamento e seu baixo custo. Todavia, esta diferença tende a diminuir com a evolução de memórias *flash*. A adoção de SSDs para armazenamento massivo em centrais de dados está ocorrendo gradativamente, como já é feito nos bancos de dado da Google® (SCHROEDER; LAGISETTY; MERCHANT, 2016), entre outras empresas (OUYANG et al., 2014). Todas aplicações que exigem uma velocidade de transferência e/ou eficiência energética superiores buscam a adotar a tecnologia SSD (SAMSUNG, 2019). A fim de maximizar a capacidade de transferência, uma especificação de protocolo chamada de NVMe foi desenvolvida, especificamente para SSDs.

2.6 *NON-VOLATILE MEMORY EXPRESS*

A especificação do protocolo *Non-Volatile Memory Express* (NVMe), também conhecida como *Non-Volatile Memory Host Controller Interface Specification* é uma especificação padrão e aberta - disponível sem custo ao público para a consulta e implementação no sítio oficial (NVM EXPRESS, 2019). A primeira especificação oficial foi lançada em Abril de 2008 pela Intel. Desde então, uma força tarefa de várias empresas foi formada para elaborar novas especificações. Dentre os membros, ainda existe um grupo de promoção, eleitos pelos próprios, já participaram/participam deste grupo as empresas como Intel, Micron, Samsung, entre outras. A especificação define uma interface de registradores para comunicação e comandos padrões.

A especificação original NVMe utiliza obrigatoriamente o protocolo PCIe, formando assim uma instrução NVMe encapsulada pelo protocolo PCIe, genericamente chamada de "comando de dado". Assim, o NVMe PCIe foi expressamente projetado para substituir a ultrapassada SATA, que limita a taxa de transferência, agindo efetivamente como um gargalo no sistema. A especificação NVMe funciona em conjunto com PCIe e aproveita o alto grau de paralelismo e baixa latência da memória *flash*, propondo um sistema do tipo produtor-consumidor, o qual está ilustrado na Figura 16.

Figura 16 – Sistema produtor-consumidor simplificado



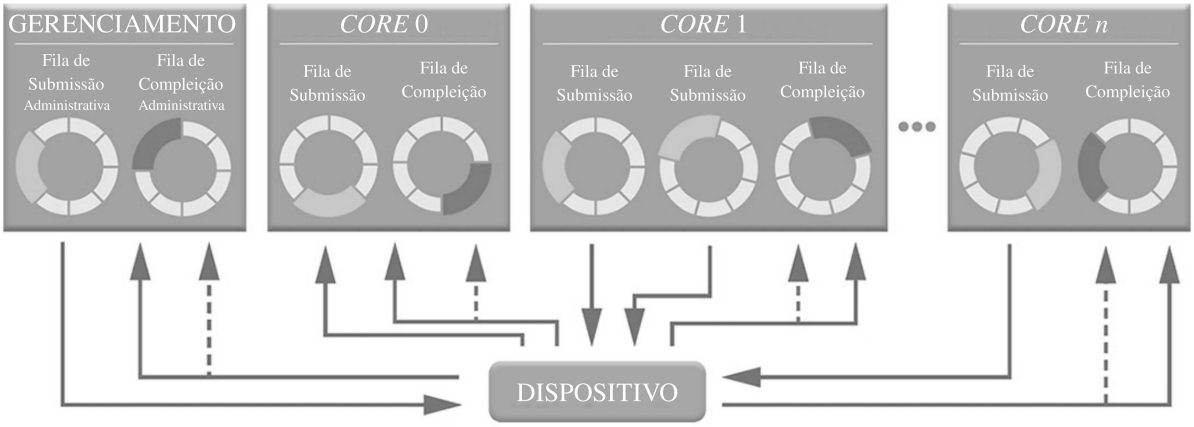
Fonte: Autor.

O sistema do tipo produtor-consumidor consiste em duas máquinas de estado: produtora e consumidora, além de uma fila compartilhada. A máquina de estados produtora produz dados e armazena na fila, já a máquina consumidora recolhe da fila e consome. Em um sistema deste tipo, cada dispositivo pode ter mais que uma máquina produtora e mais que uma máquina consumidora. Há também a possibilidade de mais que uma fila, sendo evidentemente contida ao menos dentro de um dos dispositivos. Esta flexibilidade no sistema permite um alto grau de paralelismo ao permitir que exista máquinas de estados rodando em paralelo. Outra vantagem deste sistema é que não é necessária a sincronia.

A proposta da especificação conta um sistema de diversos "*cores*", que são centrais de

fila. Cada *core* conta com ao menos duas filas: uma de submissão e outra de compleição de comandos. As filas de: submissão servem para armazenar comandos do *host* ao periférico; e a de compleição, comandos de resposta do periférico ao *host*. Existem ao mínimo dois *cores*: um para comandos de administrador e outro para comandos genéricos. Ambos os lados, tanto o dispositivo SSD quanto o *host* possuem ao mínimo uma de cada máquina de produtor e de consumidor. Todos os *cores* (junto com as filas) são armazenados no *host* (dentro do *driver*). Na Figura 17 é possível visualizar *cores* do sistema NVMe.

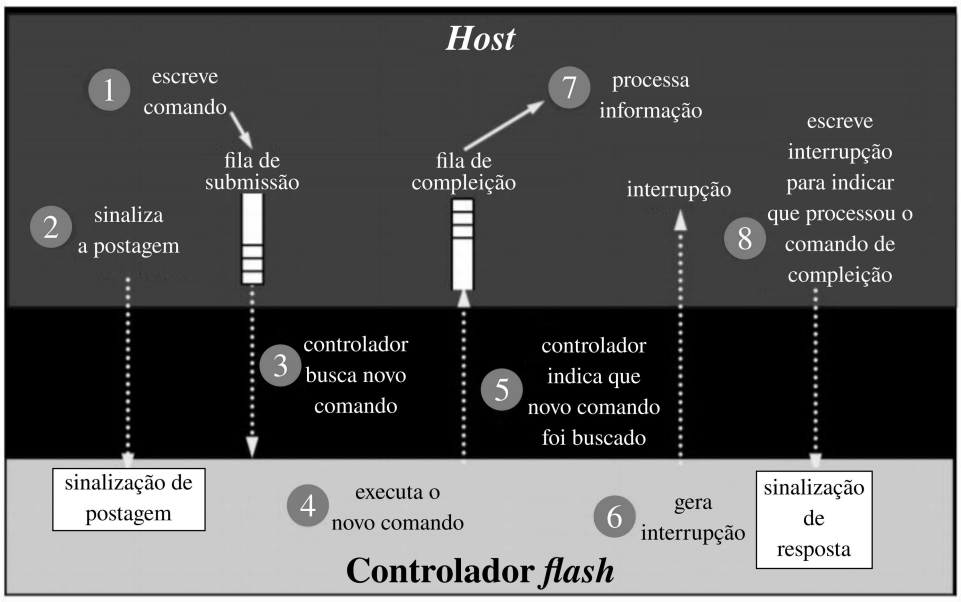
Figura 17 – Ilustração simplificada de *cores* do sistema NVMe



Fonte: Adaptado de NVM Express (2019).

Confere-se na Figura 18 os passos simplificados para postar e realizar um comando em NVMe.

Figura 18 – Processamento simplificado de comandos em NVMe



Fonte: Adaptado de NVM Express Incorporated (2019).

O *host* é capaz de postar comandos em filas com três sistemas de prioridade: *round-robin*, *round-robin* com classificação de prioridade e por último um esquema específico do fabricante (*vendor-specific*). O tamanho das filas são no máximo de 65535 comandos. NVMe pode ser implementada com no mínimo 13 comandos básicos (leitura, escrita, busca, ...) (NVM EXPRESS, 2019).

Recentemente em 2016, a especificação foi expandida para outras aplicações, chamando-se NVMe *over fabrics* - exemplos de aplicações são fibra óptica, Protocolo de Controle de Transmissão (*Transmission Control Protocol*), entre outros. Isto se deve ao reconhecimento da velocidade e eficiência do protocolo em questão. Outra vantagem deste sistema é que pode ser implementado em *hardware*, por exemplo em um FPGA.

2.7 *FIELD-PROGRAMMABLE GATE ARRAY*

Field-Programmable Gate Array (FPGA) é um dispositivo eletrônico de *hardware* programável e reconfigurável, baseado em uma arquitetura de Blocos Lógicos Reconfiguráveis, intra-conexões, Matrizes de Programáveis de Intra-conexões, primitivas e portas de IO. As primitivas são *hardwares* prontos de uso específico, por exemplo uma memória Bloco RAM ou mesmo microprocessadores integrados. Já os Blocos Lógicos Reconfiguráveis são compostos de *Look-Up Table*, multiplexadores, elementos de memória e registradores. As *Look-Up Tables* operam como portas lógicas reconfiguráveis.

Por ser um dispositivo semicondutor flexível e reconfigurável, apresenta um risco de insucesso em potencial menor que um ASIC. Por este motivo, muitas vezes o FPGA é utilizado para prototipação antes da implementação do projeto em um ASIC, podendo ser reaproveitado para projetos subsequentes. O emprego do FPGA está cada vez mais voltado para tópicos atuais, como *Machine Learning*, 5G, *Embedded Vision* e *Cloud Computing*. A tendência é de aumentar seu uso, já que o poder de processamento da CPU é insuficiente para muitas tarefas, conforme será observado na capítulo de Estado da Arte. A grande vantagem do FPGA está exatamente na sua capacidade de paralelismo. Ao contrário de um processador, o FPGA não opera como um sistema de processamento genérico, mas sim específico para o uso o qual foi projetado, permitindo a chamada “velocidade do fio” (*wirespeed*). A implementação de múltiplas máquinas de estado em paralelo é possível com o FPGA, assim como qualquer *hardware*. Entretanto, esta grande vantagem vem com um custo: a dificuldade de implementação em linguagem de descrição de *hardware*.

2.7.1 *VERY HIGH SPEED INTEGRATED CIRCUIT HARDWARE DESCRIPTION LANGUAGE*

Linguagem de descrição de *hardware* a alta frequência, ou *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) é uma linguagem popular de descrição de

hardware (*hardware description language*) originalmente desenvolvida pelo Departamento de Defesa dos Estados Unidos da América, em 1983 (DEPARTMENT OF DEFENSE, 1992). Originalmente utilizada para documentar circuitos digitais, é atualmente mantida e atualizada pelo Instituto de Engenheiros Elétricos e Eletrônicos (IEEE). VHDL não é uma linguagem de programação, como a Linguagem C ou a Linguagem *Python*, pois por linguagem de programação entende-se linguagem de *software*. Evidentemente então, VHDL é uma linguagem de extremo baixo nível, pois é possível descrever portas lógicas em nível real que deverão ser literalmente implementadas em *hardware*. Por este motivo, pode ser considerada mais baixo nível que todas as outras linguagens competidoras, consequentemente mais difícil de implementar. Este extremo baixo nível pode porém trazer uma vantagem, tudo deve ser descrito e nada é assumido, logo VHDL tem um comportamento preditivo. As outras duas linguagens competidoras mencionadas são: Verilog e SystemVerilog - comumente sendo mais fáceis de implementar.

Existe ainda uma ferramenta chamada Síntese de Abstração de Alto Nível, ou *High-Level Synthesis* (HLS), que permite o projetista escrever em linguagem de alto nível como Linguagem C ou Matlab e transformá-la para linguagem de descrição de *hardware*. Esta ferramenta pode ser oficial de um fabricante de FPGA, ou mesmo aberta da comunidade. A ferramenta HLS permite maior velocidade de implementação a um custo de um código automaticamente gerado com menor compreensão e potencialmente menos otimizado. Porém, estudos recentes permitem concluir que a ferramenta HLS não está em uma desvantagem significativa em comparação com código desenvolvido (SHARAFEDDIN et al., 2014). A descrição de *hardware* em uma linguagem de programação mais conhecida permite acesso aos projetistas menos experientes com linguagens como VHDL. A linguagem VHDL passou a ser empregada para síntese de circuitos para FPGAs nas ferramentas de fabricantes, como com a ferramenta da fabricante Xilinx® chamada de “Vivado”. Tais ferramentas de síntese permitem o uso de todas as linguagens de descrição de *hardware*. No próximo capítulo de Estado da Arte serão apresentados trabalhos correlatos com o projeto proposto, sendo o capítulo dividido em duas seções: Aplicação e Aceleração.

3 ESTADO DA ARTE

Este capítulo apresenta os trabalhos correlatos à pesquisa apresentada neste documento. Para uma melhor organização e clareza de informação ao leitor, o capítulo está dividida em duas seções. A primeira parte é de aplicação, que reúne os artigos relevantes aos estudos que tratam sobre a *flash* em si, ou controladores SSD, ou controladores *Redundant Array of Independent Disks* (RAID), ou sobre o FS, tudo sobre a camada de aplicação. Já a segunda seção, a de aceleração em *hardware*, reúne os artigos selecionados que tratam sobre aceleração em *hardware* ou sobre plataforma de testes de memória *flash*. Comenta-se em primeiro lugar, os artigos da seção de Aplicação.

3.1 APLICAÇÃO

Esta seção dedica-se a tratar de assuntos da camada de aplicação. A camada de aplicação muitas vezes é demasiadamente simples, deixando todo o trabalho laborioso ao *hardware*. Nos próximos parágrafos, ver-se-á quais são os esforços em vias melhor desenvolver a camada de aplicação.

Memórias *flash* são passíveis de perda de dado, sendo efetivamente úteis por um período mais curto comparado a outras memórias. Conforme explicado nos Conceitos Básicos, uma corrente de referência para a leitura de dados é aplicada sempre que se deseja ler da memória *flash*. Cai et al. (2015b) estudaram experimentalmente a leitura de dados em memória *flash* e chegam a duas conclusões principais: a corrente de leitura ideal muda conforme o gasto das células e; diferentes regiões possuem diferentes correntes de leitura ideais. Para remediar estes problemas, a equipe propõe duas técnicas: Leitura Otimizada de Retenção e Resgate de Falha de Retenção, respectivamente *Retention Optimized Reading* (ROR) e *Retention Failure Recovery* (RFR). A pesquisa foi desenvolvida em uma plataforma FPGA com memórias *flash* da tecnologia MLC de *2x-nm* do artigo Cai et al. (2011) (será tratado na seção de Aceleração). A técnica ROR é composta de dois componentes em série: um algoritmo de pré-otimização, rodado diariamente após cada inicialização da SSD e uma técnica otimizada de tentativa de leitura que usa a tensão de referência otimizada para cada bloco. A pré-otimização faz tentativas de leitura na última página de cada bloco, começando com a tensão de referência, armazenando assim o RBER respectivo. Em seguida faz tentativas de diminuir e aumentar, armazenando o valor de referência que corresponder ao menor RBER. Já o RFR consiste em quatro passos: identificação de células falhadas, identificação de células em risco de falha, identificação de células de perda vagarosa e rápida, por último inverter o valor do *bit*, caso necessário. Esta técnica pode reduzir RBER em até 50 %, em prática dobrando a capacidade de correção, que seria outrora impossível com ECC. Já a ROR pode estender a vida útil da *flash* por até 64 %, reduzindo a média de latência em até

10,1 %.

Relatado anteriormente, a programação de células não-SLC é tipicamente feita a múltiplos passos. O artigo Cai et al. (2017b) explora as vulnerabilidades de interferência, baseando o estudo em células MLC com tecnologia de fabricação de *1x-nm* para maximizá-las. A programação a dois passos (caso da MLC) acrescenta a probabilidade de corrupção de dados durante a escrita - tendo um período significativo entre os passos. A célula parcialmente programada possui uma tensão quiescente mais baixa comparada a células totalmente programadas, deixando-a exposta às interferências do tipo célula a célula (*cell-to-cell*) e de leitura. Para estas interferências, a equipe propõe três novas soluções: uma de eliminação de erros e duas de mitigação. A primeira proposta elimina totalmente erros relacionados à interferência de leitura, modificando a programação a dois passos para não ler uma célula parcialmente programada - mas sim um *buffer* DRAM que guarde o *bit* de interesse. Esta técnica possui um custo adicional em latência de 9,3 % a 24,2 % (para uma página de 16 kB). Caso este custo adicional não seja aceitável propõe-se outra técnica de tensão de leitura adaptativa, como proposto no artigo anterior (maior parte dos autores são compartilhados entre os dois artigos). Outra proposta é escolher três referências de tensão de leituras diferentes, uma para cada estágio da programação - pois quanto maior a diferença entre a tensão de leitura e a tensão da porta flutuante (referente ao estado), maior a interferência. Este conjunto de propostas mitigam ou removem totalmente as vulnerabilidades, estendendo a vida útil do SSD em até 16 %. A investigação se desenvolveu na plataforma FPGA do artigo Cai et al. (2011) (será tratado na seção de Aceleração).

Memórias *flash* possuem características muito desejáveis, como durações dos ciclos de leitura e escritas temporalmente mais curtos - no entanto a um custo. A tecnologia *flash* é baseada em transistores de porta flutuante (conforme o capítulo de Conceitos Básicos) e com isto, existe desgaste do isolante, obrigatoriamente reduzindo a vida útil em comparação com discos rígidos. Outrossim, bem como já comentado anteriormente, memórias *flash* são sensíveis à interferência externa, principalmente dos tipos não-SLC. A fim de compreender melhor sobre as interferências que a memória *flash* é susceptível, Zhang et al. (2017) desenvolveram uma plataforma para caracterização de erros em memórias *flash* MLC. A plataforma propõe explorar erros de interferências de programação na alocação de página baseado em esquema, ou *page allocation scheme-based program interference* (PASBPI), com o objetivo de ter um maior conhecimento sobre as causas e como evitá-las. A plataforma se baseia em um FPGA Spartan3E da Xilinx® com capacidade de 16 memórias NAND *flash*. O FPGA utiliza também um processador "programável" (*soft-processor*) MicroBlaze com um controlador DDR2, um buffer RAM de dupla porta de 32 kB e quatro controladores *flash* que permitem quatro memórias *flash* por canal. As memórias a serem testadas são todas *2x-nm* MLC NAND *flash*. Diferentes paradigmas de alocação são analisados. Por fim, é elaborada uma tabela para analisar os erros conforme o paradigma de alocação e o modo de acesso (aleatório ou sequencial). As conclusões do artigo deverão ajudar para projetar ECC mais adequados e esquemas de gerenciamento de erro melhores.

Controladores e FTL são um campo já bem explorado. Originalmente criados para apresentar uma transição suave entre HDDs e SSDs, acabaram incorporando características dos discos rígidos que não existem em SSDs. Isto abre espaço para aperfeiçoamento para incorporar as características das memórias *flash*. O controlador *Out-Of-Order* (fora de ordem) de Nam et al. (2011) propõe exatamente explorar o paralelismo dos SSDs - que contém múltiplos canais de comunicação com os invólucros individuais, propondo a comunicação paralela entre estes invólucros e o envio de pacotes fora de ordem. O artigo se tornou uma referência hoje e explica as diferenças entre algoritmos sequenciais, desacoplados e paralelos. O controlador implementa a arquitetura paralela em uma placa de desenvolvimento FPGA e compara com ambas as duas arquiteturas concorrentes. O artigo conclui que o controlador Ozone (O3) obtém entre 3 % a 100 % melhor desempenho em comparação as outras arquiteturas, que responde entre 46 % a 88 % em comparação à execução desacoplada, ambos para as cargas de trabalho testadas. A comparação do desempenho fica mais gritante quando a carga de trabalho contém mais apagamento/escrita, já que a leitura é a mais lenta entre todas as ações em memórias *flash*.

Deduplicação é um processo que reduz significativamente a ocupação da memória por dados de um armazenamento. Retendo somente uma única instância do arquivo, dados redundantes são substituídos por ponteiros ao dado original. É um algoritmo complexo que, deixado para uma CPU, consome muita energia e possui um considerável tempo de execução - além de ocupar a CPU. Para isto, Ajdari et al. (2018) se propuseram de implementar este algoritmo em FPGA que faz a deduplicação antes de armazenar no SSD (*inline*), aumentando a vida útil do SSD. Os pacotes são processados em 64 kB para minimizar o metadado de criptografia *Secure Hash Algorithm 256 bits* (SHA-256) do pacote. Primeiramente recebe um pedido de escrita e calcula-se o SHA-256 do pacote. Sem demora, compara com a tabela de SHA-256 de outros pacotes anteriormente armazenados. Por último, caso o SHA-256 de algum outro pacote seja uma comparação idêntica, o pacote é considerado idêntico. Após a identificação, o pacote é armazenado no SSD, junto com seu valor SHA-256 na tabela de comparação em NVRAM. O protótipo foi capaz de reduzir o uso da CPU em 93,6 % e em aproximadamente 20 % o consumo. Além de comprovar melhora no processo, é economicamente mais favorável para *data centers* que previamente empregavam várias CPUs em paralelo para a deduplicação.

Simulações para desenvolvimento de novos controladores SSD e arquiteturas para Sistema de Gerenciamento de Múltiplas Unidades de Armazenamento (*Redundant Array of Independent Disks*, RAID) aplicadas para SSD são fundamentais. A atual plataforma para pesquisa de memórias *flash*, a Plataforma de Pesquisa *Flash*, ou *Flash Research Platform* possui certas limitações. Ela é dependente de certas arquiteturas, não suporta conjuntos de SSDs (RAID), não permite investigar o não-determinismo de SSDs e exige que se compre as memórias em questão. Por estes motivos, Komsul, Mcewan e Mir (2014) propõem uma nova plataforma de co-simulação para SSD, inteiramente simulada em FPGA (mais barata), reconfigurável e que permite investigar topologias RAID e o não-determinismo intrínseco de SSDs. A plataforma proposta consiste em duas partes: computador e FPGA, respectivamente simulando o *host* e o

SSD ou RAID de SSDs. Para validação do FPGA, a equipe considera cinco canais *flash* (*Flash Bus*) e três memórias *flash* por canal, além de mais uma memória *flash* avulsa. Os resultados apresentam uma semelhança sobre todos os aspectos, inclusive no quesito do não-determinismo, entre a plataforma proposta e o sistema real.

Conforme discutido nos Conceitos Básicos sobre *multi-stream*, há um trabalho emergente para determinar os diferentes grupos de *StreamID* - tal que seja minimizado o tempo de processamento e aumentar a vida útil do dispositivo. Com esta finalidade, Bhimani et al. (2018) se atém a dar resposta a esta questão. O artigo propõe uma nova variável booleana chamada de coerência (*coherency*): verdadeira quando múltiplos blocos lógicos são atualizados em tempos similares. Variáveis tecnicamente objetivas e mensuráveis devem classificar os dados nos diferentes grupos de *StreamID*. Acesso adjacente, coerência, sequencialidade e frequência são variáveis analisadas. O artigo também propõe o uso do algoritmo de classificação de grupos *K-mean clustering* para treinar os dicionários base, que identificam os diferentes grupos. A equipe modificou um simulador tradicional chamado de DiskSim para implementar suporte a *multi-stream* e testar os resultados. A medição do sucesso do método é feita pelo WAF. Em conclusão, nenhuma das características é universalmente melhor que a outra. Em outra, *multi-stream* integra-se à latência, porém é considerada aceitável, já que o objetivo da proposta não é diminuí-la, porém de prolongar a vida útil da memória. A tecnologia de *multi-stream* é capaz de diminuir o WAF em no mínimo 20 % podendo chegar em até 70 %.

Diversas soluções propostas anteriormente buscam soluções para questões de armazenamento de dados no nível do *hardware*. Entretanto, existem cuidados especiais para o armazenamento também no nível do *software*. Estes cuidados visam que o armazenamento seja feito de maneira eficiente e garantida. Um grande exemplo disso em prática são os banco de dados que requerem a utilização de transações para que o armazenamento seja feito de maneira segura e confiável, em detrimento ao desempenho, sendo o mais comum deles o SQL. SQL é uma linguagem de gerenciamento e armazenamento de estrutura de dados. Sua biblioteca para o uso em linguagem C/C++ chama-se SQLite, estando presente em diversas aplicações, tais como o Facebook, Twitter, Gmail, entre muitas outras. SQLite funciona em dois modos de *journaling*: *rollback* e *write-ahead*. O modo *Rollback* substitui toda a base de dados em um outro arquivo (chamado de *rollback journal*), alterando-a aos novos valores. Já no modo *write-ahead*, a nova base de dados é concatenada à antiga. SQLite possui um problema: a atomicidade nativa de pacotes no envio é ruim e ineficiente, tendo que ser implementado em diversos casos pelo próprio desenvolvedor.

Kang et al. (2013) implementam uma FTL transacional (sinônimo para atomicidade), o X-FTL. Esta FTL foi projetada em especial para o SQLite, mas pode rodar com qualquer outro uso na camada superior, tal como o FS. O X-FTL libera o FS da incumbência da alta-redundância ao passar a atomicidade para o FTL, encontrado no controlador da própria memória *flash*. Outrossim, esta atomicidade não possui um custo adicional, devido ao sistema de *copy-on-write*,

que não apaga as páginas anteriores. Isto transforma um ponto fraco das memórias *flash* em um ponto forte: atomicidade inerente, diminuindo pela metade a quantidade de dados a serem escritos e consequentemente duplicando a vida útil da memória. Logo, as mudanças tanto no FS quanto no FTL são mínimas, modificando-se somente o comando abortar no SQLite. A biblioteca pode rodar com *journaling* desligado, pois o próprio X-FTL é responsável pela atomicidade. O protótipo foi desenvolvido na plataforma OpenSSD, com o FS Ext4 como base. Uma comparação é feita no mesmo *hardware* com o FTL e SQLite originais e modificados foi feita para quatro bases de dados SQL sintéticas e reais para telefones celulares. O SQLite original foi testado com ambos os modos de *journaling*, enquanto que no modificado foi desligado. SQLite rodou entre 2,4 a 3 vezes mais rápido com X-FTL, em comparação com o modo de *write-ahead* - que possui melhor resposta em comparação com *rollback*. Outra pilha de testes foi rodada para testar X-FTL sem SQLite, o *Flexible IO* (FIO) *benchmark*. Neste caso, o FS Ext4 desempenhou 67 % a 99 % e 240 % a 254 %, respectivamente em comparação com *journaling* ordenado e completo. Para todos os testes executados, o X-FTL desempenhou consistentemente melhor nas comparações.

Comumente aplicações devem implementar seu próprio protocolo de FS para serem toleráveis à falhas (*crash-consistent*). Pois, caso o sistema trave, dois blocos de dados podem ser atualizados em conjunto pelo FS, deixando um bloco de dado que não é tolerável à falhas, afetando o outro (fonte artigo). Isto requer algum tempo, retirando a atenção do projeto inicial - algo que poderia ser padrão e disponibilizado como qualquer outro FS. Por este motivo, Min et al. (2015) propõem uma modificação do FS previamente citado, o Ext4 utilizado em conjunto com X-FTL do artigo (KANG et al., 2013). O novo FS modificado chama-se CFS, que garante propagação atômica a fim de ser tolerável à falhas, propondo funções para declaração de atomicidade dos dados. Modificações mínimas da linha de código existente no Ext4 são necessárias (aproximadamente 5800 linhas alteradas). Aplicações anteriores são compatíveis sem modificações. Porém, caso o aplicativo precise ser tolerável à travamentos, deve ser modificado. Isto significa modificar em média menos de 0,01 % - para as aplicações modificadas no artigo. A carga de teste é representada por um total de cinco aplicações: SQLite, MariaDB, Kyoto Cabinet, APT e vim, que utilizam bases de dados distintas. A equipe roda este conjunto de teste em um computador com processador Intel Xeon E5606 de quatro *cores* e 4 GB de memória. Foram utilizadas a plataforma OpenSSD com 8 GB de memória e duas interfaces SATA. Em média, CFS desempenha duas a cinco vezes mais rápido em operações de escrita enquanto que reduz o número de escritas à memória em 1,9 a 4,1 vezes, garantindo atomicidade e compatibilidade a aplicações anteriores.

Memória Persistente (*Persistent Memory*, PM) é um novo tipo de memória que será utilizada entre a memória DRAM DDR e SSD nas futuras arquiteturas de computador. Este tipo emergente de memória recebe dados através de uma DRAM volátil para posteriormente armazenar na memória *flash*, trazendo assim os benefícios do SSD e a velocidade de banda das memórias DRAM. Outrossim, Sistemas de Arquivo (FS) modernos, como o Ext4, baseiam-se em blocos de metadados chamados *Inodes*. Quando o sistema de *journaling* está ativo,

estes blocos devem ser copiados, independentemente da localização de seus blocos físicos, indiretamente incrementando o WAF. Por este motivo, Yang et al. (2019) propõem um sistema de Agregação de Inode Baseada em Frequência de Atualização, *Updating Frequency based Inode Aggregation* (UFIA). Este sistema consiste na identificação de *Inodes* inválidos para suas realocações adjacentes no FS. Estas realocações reduzem o WAF, pois blocos apagados não são mais misturados com blocos inválidos — mais prováveis de serem reprogramados. Esta reorganização é então armazenada na PM. Três pilhas de teste diferentes (*benchmark*) foram aplicadas ao UFIA, resultando em uma redução entre 3,34 a 2,31 vezes no WAF. Este sistema também apresenta um aumento de operações por segundo em até 66 %, devido à redução do número de escritas. No Quadro 8 fez-se um consolidado dos trabalhos revisados nesta seção.

Quadro 8 – Artigos da seção de aplicação

Ano	**Autores**	**Trabalho Realizado**	**Absorção**
2015	Cai, Y. et al.	*Data Retention in MLC NAND Flash Memory Characterization, Optimization, and Recovery*	Características da tecnologia *flash*, em específico MLC NAND
2017	Cai, Y. et al.	*Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques*	Programações da *flash* não-SLC
2017	Zhang, M. et al.	*FPGA-Based Failure Mode Testing and Analysis for MLC NAND Flash Memory*	Padrões de erro na *flash* e uso de FPGA associado.
2011	Nam, E. H. et al.	*Ozone (O3): An Out-of-Order Flash Memory Controller Architecture*	Funcionamento de um controlador *Out-of-Order* de referência da literatura atual
2014	Ajdari, M. et al.	*A Scalable HW-Based Inline Deduplication for SSD Arrays*	Arquitetura para lógica de um deduplicador *inline* com SSDs
2014	Komsul, M. Z. et al	*An FPGA-based Development Platform for Real-time Solid State Devices*	Simulação de um SSD em lógica
2018	Bhimani, J. et al.	*FIOS: Feature Based I/O stream identification for improving endurance of multi-stream SSDs*	Sistema *multi-stream* e processo de determinação de *StreamID*
2013	Kang, W. et al.	*X-FTL: Transactional FTL for SQLite Databases*	FTL específico para uma aplicação, também servindo de referência da literatura atual
2015	Min, C. et al.	*Lightweight Application-Level Crash Consistency on Transactional Flash Storage*	FTL com consistência a travamentos
2019	Yang, C. et al.	*Reducing Write Amplification for Inodes of Journaling File System using Persistent Memory*	Algoritmos para extensão do tempo de vida da *flash*

Fonte: Autor.

Os artigos citados (Nam et al., 2011) e (KANG et al., 2013) são mais antigos, mas sua importância justifica a referência. O controlador desenvolvido no artigo (Nam et al., 2011) é frequentemente utilizado como comparação para outros artigos, além de ter sido o precursor

de controladores do tipo “fora de ordem” (*Out-Of-Order*). Já o (KANG et al., 2013) demostra como um FTL pode ser feito especificamente para uma aplicação, sem deixar de contemplar outras aplicações, trazendo benefícios para estas. Este FTL também é frequentemente citado na literatura. A realocação de blocos é realizada em (YANG et al., 2019), trabalhando na redução de WAF.

A seção de Aplicação apresentada integra trabalhos relacionados à tecnologia *flash*, às camadas de abstração e aos controladores SSD ou RAID. A camada de abstração é igualmente relevante em vista do projeto proposto quanto a otimização de acesso, sendo necessária sua compreensão em detalhes para o objetivo do projeto. Concluída esta seção, atentar-se-á àqueles artigos que estão mais diretamente relacionados ao trabalho proposto - na seção de Aceleração.

3.2 ACELERAÇÃO

A presente seção de Aceleração contempla artigos que utilizam no mínimo um FPGA para acelerar processamentos específicos corriqueiros. Este FPGA seguidamente é acompanhado de uma memória *flash* e/ou outro sistema de processamento, frequentemente processamento por CPU. Existem algoritmos especificamente susceptíveis à aceleração, que podem ser acelerados por um processamento em FPGA ou por um ASIC - como poderá ser observado. Para isto, neste momento faz-se importante esclarecer algumas noções:

- **Processamento *in-datapath***: processamento que fica “no caminho” do fluxo de dados. Vantajoso em comparação ao seu antagônico, o processamento *off-datapath*, já que este deixa o dado ser armazenado para depois processá-lo. Porém é mais custoso/difícil que este, já que a capacidade de fluxo deve corresponder aos dados - vide o uso de lógica programável. Apresenta uma vantagem única aplicado à *flash*: não possui múltiplos ciclos de programação, estendendo a vida útil desta.
- **Computação de borda**: Conhecido em inglês como *edge-computing*, trata-se de aplicar processamento perto de onde o dado é consumido. Exemplo: processamento perto da memória.
- **Algoritmo *greedy***: Qualquer algoritmo que busca uma solução ideal (máximo global), contudo por questões de tempo e recurso, permite entregar uma solução sub ideal (máximo local).

Esclarecidas estas definições, a seguir são comentados os artigos sobre aceleração empregando lógica programável. As plataformas FPGA para pesquisa em tecnologia *flash* são bem-vindas. Song et al. (2014) criam uma plataforma chamada OpenSSD Cosmos+ para desenvolvimento relacionado a memórias *flash* e SSD. O protótipo Cosmos+ é o terceiro protótipo do projeto OpenSSD, sendo os dois precedentes as placas Jasmine e a Cosmos, em ordem. Os

protótipos foram desenvolvidos por um grupo de pesquisa na Coreia do Sul e são comercializados mundialmente. Alguns dos artigos anteriores e próximos que serão citados utilizaram a plataforma OpenSSD, que é conveniente pois possui suporte amplo e comunidade para desenvolvimento. A placa Cosmos+ possui como centro um FPGA SoC Xilinx® Zynq-7000®, pronto com um processador embutido ARM Cortex-A9. A placa também conta com memória DDR3, PCIe de terceira geração com quatro vias, quatro canais de comunicação de invólucro *flash* e memórias *flash* MLC. Este projeto é o mais próximo do que é proposto por este trabalho, porém fundamentalmente diferente: OpenSSD é uma plataforma genérica com o objetivo de estudar a tecnologia *flash* por si só e não possui um FPGA com lógica suficiente para implementações mais complexas, além de não possuir conector para SSDs comerciais.

Caracterizar erros de tempo de vida é importante para planejar ECC e *Wear-leveling* efetivos. Cai et al. (2011) propõem uma plataforma FPGA de acesso à memória *flash* que permita acelerar o gasto do SSD ou manter em uso normal. O modo de aceleração de geração de erros tem a finalidade de conhecer o que ocorre por toda a vida útil do SSD em um curto período de tempo. Com estes resultados, erros são caracterizados e um novo GC é proposto que desconsidera páginas ruins, ao invés de blocos ruins. Blocos ruins passam a ser marcados a partir de um gatilho configurável. No caso do gatilho configurado para 32 páginas (metade), estende-se a vida útil em 51 %.

Alguns algoritmos são próprios para a aceleração em FPGA, um deles é a ferramenta de procura de comparações de informações sequenciais biológicas de aminoácidos, conhecida em inglês por *Basic Local Alignment Search Tool* (BLAST). BLAST é um algoritmo comum em biologia para sequenciamento de genomas. É computacionalmente intenso, pois compara uma sequência de aminoácidos a ser analisada a uma base de dados de grande proporção. O algoritmo é essencial para descobertas no campo da biologia e medicina. Yoshimi, Wu e Yoshinaga (2016) propõem uma rede de CPUs acelerada por nós FPGAs de pré-processamento de dados. O sistema implementado no FPGA é chamado de Motor de Processamento de Fluxo de Dados, ou *Data Stream Processing Engine* e comporta duas funções do BLAST: particionamento da base de dados e uma pré-identificação das sequências de interesse. O artigo propõe um *In-DataPath-BLAST*, que computa todo o dado que passar pelo FPGA. *In-DataPath-BLAST* anota a posição do fluxo de dado dentro da memória SSD. Há um melhoramento de 20 vezes no tempo de computação do particionamento da base de dados e em um fator de 321,4 % no tempo de computação na pré-identificação. Por fim, o *In-DataPath-BLAST* comprovou que o gargalo se encontra na SSD, pois o fluxo de dados obtido é a capacidade da SSD de fornecer dados.

Ainda no tema de algoritmos computacionalmente complexos, encontras-se os classificadores (*sorting*). Podem ser utilizados dois padrões para classificação: Joulesort e Terasort, respectivamente potência consumida e itens organizados *por segundo* - sendo este último uma quantidade de itens na casa do 1 TB. Com isto, o artigo de Jun e and (2017) se propõe a implementar um acelerador de classificação em FPGA. O acelerador não se encaixa nos critérios

de ambos para concorrer ao recorde, porém são utilizados os recordistas como parâmetro. O acelerador é capaz de classificar 1 TB com itens de 16 B, em 5000 s, com o consumo energético de 140 W. Isto significa que ultrapassou o campeão do Joulesort em mais de 200 % (na data de publicação), enquanto que Terasort desempenhou mais de 50 % acima do campeão (na data de publicação) - que utilizou uma rede de computadores de 21 nós. Histórico de campeões dos diversos padrões, outros padrões e explicações sobre os padrões podem ser acessados em (SORT BENCHMARK PÁGINA PRINCIPAL, 2019).

Análise de grafos é um tema relevante para muitas áreas, como: rede Internet, pontuação para páginas da Internet, redes sociais, análise de neurônios no cérebro, redes neurais, rede pública comutada e até mesmo detecção de ataques terroristas virtuais, para citar algumas aplicações. Porém, é computacionalmente intensa, tendo grafos de até 3,5 bilhões de vértices, ou seja 2 TB em formato de texto. Já existem vários sistemas de processamento como X-Stream, FlashGraph, GraphChi, GraphLab e vários outros - boa parte deles somente com CPU. Jun et al. (2018) se propõem a implementar um sistema híbrido de CPU com acelerador FPGA. O artigo promove uma nova arquitetura chamada de *Ordena-Reduz* (*Sort-Reduce*), que consiste em uma parte CPU e outra parte acelerador. O acelerador FPGA se encarrega de pré-processar os dados de forma a economizar a CPU, deixando-a a se dedicar para outras tarefas específicas do algoritmo a ser implementado. Existem *n* algoritmos para análise destes grafos, resultando em análises relevantes. O acelerador basicamente trabalha para ordenar cada pacote de dado (vértice ou aresta) que é enviado para o pré-processamento. Cada pacote atômico possui tamanho fixo, de 512 kB armazenados em DRAM para acesso rápido. O acesso à memória *flash* é sequencial, sendo mais rápido que acesso aleatório. Com isto é proposto um novo FTL, eliminando a capacidade de acesso aleatório e reduzindo o atraso que se teria com esta função adicional. Para fins de comparação, a equipe elaborou o mesmo algoritmo (chamado de GraFSoft) para concluir sobre a vantagem que traz o FPGA para o sistema proposto. Para comparação, a equipe rodou os algoritmos baseados em CPU em um único servidor com processador Intel Xeon E5-2690 de 32 *cores* à frequência de 2,90 GHz e 128 GB de DRAM. Como armazenamento, equipou o servidor com cinco SSD PCIe, totalizando a capacidade de 6 GBps de banda com leitura sequencial. Já para GraFBoost, o sistema utilizou um servidor Xeon X5670 de 24 *cores* e 48 GB de memória. A plataforma FPGA usada foi VC707 da Xilinx® com 1 GB de cartão de memória DDR3 DRAM, complementada por duas memórias *flash* de 512 GB. O gargalo mais proeminente do GraFBoost foi a memória DRAM, sendo evidenciado quando disponibilizado uma DRAM com banda de 20 GBps. Em média, GraFBoost desempenhou entre duas a quatro vezes melhor. Outrossim, foi capaz de trabalhar com grafos os quais boa parte dos sistemas de processamentos não foram. Para o consumo de energia, consumiu 160 W de energia, sendo 68,75 % consumidos pela CPU em baixa carga.

Ainda no tópico de grafos, um problema recorrente é deduzir a árvore de extensão mínima, *Minimum Spanning Tree* (MST). A MST é um problema o qual, dado um grafo, deve-se achar uma solução para conectar todos os pontos, com a soma dos pesos das arestas minimalizada.

O algoritmo comumente empregado para solucionar este problema é o Kruskal, que consiste em primeiro ordenar os pesos das arestas para formar a MST. Para acelerar a solução de uma MST, Li et al. (2018) criaram um sistema chamado de Coordenação para SSD inteligente e CPU ou, em inglês *Coordinating Intelligent SSD and CPU* (CISC). O CISC é composto de uma parte CPU e outra parte FPGA. O FPGA é encarregado de ordenar as arestas de acordo com seu peso. Já a CPU se encarrega de processar a MST, baseada na informação pré-processada pelo FPGA. O FPGA é diretamente integrado à memória SSD, fornecendo assim pesos já pré-ordenados à CPU. Com uma filosofia similar ao previamente relatado, o sistema proposto Jun e and (2017): *Terabyte Sort*, o FPGA processa pacotes de tamanho fixo, porém desta vez, pacotes que cabem na memória interna do FPGA (BRAM). Para os testes comparativos, um servidor equipado com processador Intel Xeon com 96 *cores*, rodando a frequência de 2,5 GHz e um sistema operacional Linux (*kernel* versão 4.14) foi utilizado. O sistema conta com PCIe geração 3 de 4 *lanes* que conectam com o CISC, implementado na plataforma Open-SSD. O sistema possui desempenho em média 220 % a 270 % na implementação do algoritmo CISC em série e 1170 % a 1720 % na implementação paralela - comparados a algoritmos tradicionais.

Algoritmos *Sketching* procuram selecionar uma amostra representativa de uma base de dados maior. Por representatividade entende-se que há uma característica de interesse da base de dados e esta amostra possui a aproximadamente a mesma característica (por exemplo valor médio). Este algoritmo é útil quando se trabalha com bases de dados muito grandes, tentando aplicar algum processamento, porém com recursos limitados. Outro algoritmo interessante para *Big data* é *Orthogonal Matching Pursuit* (OMP). OMP é uma técnica de decomposição de dados em múltiplas funções chamadas de átomos a fim de representar a função original como uma combinação linear destes átomos - sendo que cada átomo é ortogonal. A técnica gera duas matrizes, a primeira de dicionário de átomos e a segunda matriz de pesos. Pode-se imaginar a Transformada Complexa de Fourier como uma OMP, sendo átomos funções senoidais e os pesos as amplitudes.

O artigo de Rouhani et al. (2015) propõe um novo protótipo para computação de matrizes densas chamado SSketch. Matrizes de dados são consideradas densas quando possuem um alto grau de correlação, ou seja elementos não-zeros. SSketch se utiliza de múltiplos *kernels* OMPs para gerar a matriz de dicionário esparsa e a matriz de pesos. A matriz de pesos é composta por blocos, sendo cada bloco idealmente independente. Cada bloco independente reduz o acesso à memória, consequentemente o atraso, e pode ser processado pelo processador independentemente. O protótipo é construído por um acelerador FPGA e um processador de dados, cujo objetivo é processamento em fluxo (*streaming processing*). A única função do acelerador é implementar o algoritmo de *sketching* para simplificar o processamento posterior em *software*. Para isto, o acelerador FPGA implementa múltiplos *kernels* OMPs de forma paralela para possuir boa vazão de dados, sendo interfaceado por uma conexão Ethernet de 1 Gbps. Outrossim, a equipe fornece uma interface de programação de aplicativo (*Application Programming interface*) para o desenvolvimento de processamento com aprendizado do dicionário adaptativo em tempo real. O

acelerador é baseado na placa de desenvolvimento ML605 da Xilinx®, que contém um FPGA Virtex-6 e o processamento é feito por um computador com processador Intel core i7-2600K rodando Windows OS. Para o teste, são rodadas três bases de dados, sendo comprovado que SSketch pode rodar até 200 vezes mais rápido comparado a somente processamento em CPU à frequência de 3,4 GHz para o protótipo.

Big data trata-se de um grande conjunto de dados com informações relevantes a serem extraídas. Estes dados podem vir das mais diversas áreas como redes sociais, supermercados, governo, fábricas, Internet das Coisas e muitas outras áreas. É um tema contemporâneo sendo alvo de pesquisa e otimização, com ênfase em aceleração do processamento e armazenamento. Hoje, existem empresas dedicadas ao processamento de *big data* para terceiros. No artigo Jun et al. (2014) é proposto uma nova arquitetura de armazenamento chamada de *Blue Database Machine* (BlueDBM) especificamente para *big data*. A arquitetura BlueDBM consiste em utilizar nós de FPGA *in-datapath* para possibilitar a criação de aceleradores em uma rede distribuída, com acréscimo mínimo em atraso. Cada nó se comunica com um computador via PCIe, tendo a possibilidade do PC mandar processar dados diretamente no FPGA. Os nós são compostos de blocos de interface com o cliente, acelerador, mapeador de endereços, comunicação entre FPGAs e controlador *flash* - sendo este último implementando uma simples FTL. Os diferentes nós se comunicam em um protocolo proprietário baseado em serializador-desserializador. A rede BlueDBM é flexível no sentido que se pode adicionar ou remover nós à rede enquanto o sistema estiver ligado, graças a um mecanismo de reconhecimento automático de nós. Para o teste de arquitetura, foi construída uma rede com quatro nós baseados na plataforma FPGA ML605 com aceleradores que fazem uma contagem de palavras específicas (por exemplo, contar quantas vezes “teste” aparece em determinado arquivo). Cada ML605 é ligada com uma placa customizada com conector FMC contendo quatro canais paralelos com o total de oito memórias de 512 GB em tecnologia SLC, cada qual com 27 µs de atraso média. Como o conector utilizado contém praticamente todos os *transceivers* da placa ML605, a conexão entre FPGAs precisou ser feita por meio de conectores SMA. Empregaram-se adicionalmente mais duas placas ML605 para realizar a parte de comunicação em rede, utilizadas como *hub*s. Cada comunicação entre FPGAs ponto a ponto dura em média 0,5 µs, representando assim menos de 1,85 % do atraso médio de uma única *flash*. A comunicação com o computador por PCIe provou ser o fator gargalo no teste, podendo acelerar ainda mais, caso melhor *hardware* seja empregado.

Baidu (2019) é um provedor de serviços digitais *online* chinês, sendo efetivamente concorrente do Google®. Seu sítio de pesquisas é o mais popular da China, contando também com vários serviços como armazenamento em centros maciços de dados, Inteligência Artificial e loja de aplicativos, entre outros. A equipe de engenheiros da Baidu® Ouyang et al. (2014) publicaram o artigo, no qual propuseram a *software-defined flash* (SDF), essencialmente uma SSD adaptada a seu banco de dados de Internet. A SDF conhece sua carga de trabalho e respectivas peculiaridades, como proporção de escrita/acesso aleatório ou sequencial, podendo assim melhor se adaptar. O protótipo conta unidades de escrita do tamanho de blocos, fazendo

assim que a concorrência de acessos seja párea ao paralelismo do *hardware*. O controle da SDF exige que as operações de GC aconteçam antes da escrita. Os *bits* extras para proteção de dados também são removidos, logo o protótipo fia-se no mecanismo nativo de replicação de dados. Tudo isto faz com que seja eliminado por completo a necessidade de *over-provisioning*. Percebe-se que SDF não é um simples substituto para SSD, pois é adaptado a grandes dados sequenciais, sendo vagaroso para múltiplas pequenas escritas. Para isto, Baidu só emprega SDFs em sua grande base de dados especificamente para páginas da Internet, como exemplo o índice de páginas Web. O resto do tráfego é desviado a outro servidor especializado em pequenos dados, para não ocupar a banda. O controlador *flash* é desenvolvido em um FPGA da Xilinx®, o Spartan-6® implementando um controlador e FTL no mesmo invólucro. Já um segundo FPGA, o Virtex-5 é empregado para se comunicar com os computadores via PCIe. Cada controlador *flash* baseado em Spartan-6 gerencia de 11 canais *flash*; e cada Virtex-5 faz a conexão de quatro Spartan-6 com o servidor - no total cada SDF possui cinco FPGAs. As memórias *flash* são da tecnologia MLC com 25 nm, cada qual com 8 GB de capacidade, totalizando ao SDF uma capacidade de 704 GB. Como *buffer* utilizou-se dois cartões DDR3 de 512 MB à frequência de 533 MHz. O tamanho mínimo de armazenamento é de 8 MB. O atraso de apagar/programar do SDF média em torno de 383 ms, com pouca variação - em comparação à uma SSD com as mesmas especificações (quantidade de canais e memória iguais) o atraso de programação (somente programação) fica entre 7 ms a 650 ms, com grande variação instantânea. Na data de publicação do artigo, mais de 3000 SDF estão rodando no centro de armazenamento de dados de Internet. As medições mostram que são entregadas 95 % da banda de frequência original da *flash* com 99 % da capacidade total, enquanto que o custo é reduzido pela metade, comparado aos SSDs comerciais. Baidu planeja aumentar o número de SDFs em seus centros de armazenamentos.

A tendência de integrar FPGA paralelamente com processamento é universal: um FPGA é capaz de automaticamente adequar-se, caso seja flexível, a uma arquitetura que auxilie em acelerar o processamento. Telefones inteligentes (*smartphones*) não estão distantes desta tendência. Atualmente, os aplicativos possuem uma única parte em *software*, porém, caso seja integrado uma lógica programável, aplicativos também poderão ter uma parte em *hardware*. Projetistas de produto não terão mais restrições a muito (custo) ou pouco (sub-processamento) *hardware*, passando a decisão de funções de *hardware* para o usuário final. Apesar das vantagens, há uma complicação óbvia com esta proposta: aplicativos maliciosos utilizando *hardware* de maneira prejudicial para burlar sistemas de segurança para garantir acesso privilegiado ao aplicativo, acessando inteligência indevida ou até mesmo partes sensíveis do celular.

Com a proposta de integrar lógica programável em celulares, Coughlin, Ismail e Keller (2016) propõem uma junção de FPGA e telefone celular. Para distribuição de aplicativos de forma segura, uma loja de aplicativos híbridos (*software* e *hardware*) é idealizada e projetada, com restrição de recursos, a fim de evitar que aplicativos maliciosos prejudiquem o telefone. A loja de aplicativos gerencia a separação e utilização de recursos do FPGA, para que cada aplicativo

tenha seu espaço independente no aparelho. Isto permitirá aos usuários de fazerem escolhas com a tecnologia que adquiriram e também desenvolvedores poderão simplesmente atualizar o *hardware* após ter sido comprado, por exemplo atualizar a tecnologia de telefonia celular de 3G a 4G - o que hoje não é possível. A loja de aplicativos proposta deverá reconfigurar a lógica programável para múltiplos tipos e fabricantes FPGAs. Por este motivo também é apresentado uma reconfiguração integrada na nuvem: a Cloud RTR, sendo RTR (*run-time reconfiguration*) a habilidade de reconfigurar o FPGA integrado em tempo real. O compartilhamento do FPGA entre diferentes aplicações sugerido é baseado em reserva de recursos lógicos. Os desenvolvedores de aplicativos poderão trabalhar com uma ferramenta de abstração de alto nível (*High-level Synsthesis Tool*) para desenvolver a parte do *hardware* (já que programadores hoje tipicamente não possuem conhecimento de HDL), que gerará *bitstreams* parciais a serem disponibilizados na nuvem. O protótipo foi desenvolvido na placa FPGA SoC Zedboard, que conta com um FPGA Zynq 7020® da Xilinx®, integrado no mesmo pacote com um processador *dual-core* Cortex A9 (similar ao processador do iPhone 4). Como teste, foram desenvolvidas em C++ HLS um aplicativo de rádio de modulação QAM (*Quadrature Amplitude Modulation*) e um módulo de criptografia AES (*Advanced Encryption Standard*) de 128 b aplicado no projeto *open source*, o navegador Orbot (Internet Tor) para Android. Ambos provaram ser respectivamente quarenta vezes e doze vezes mais rápido em comparação com o padrão Android.

O sistema operacional Linux possui dois tipos de acesso à memória: através do FS ou do *Direct IO*. Ambos os tipos de acesso comunicam-se com a camada *Block IO*, parte integrante do *kernel*. Esta camada, obrigatoriamente assume que o periférico será mais lento que o processador. Este paradigma, no entanto, foi recentemente rompido com o advento do SSD NVMe, fazendo com que esta abstração prejudique o desempenho do sistema. Vistos os recentes avanços em termos de aceleração em FPGA para aprendizado de máquina e *big data*, pode ser questionado, se esta própria camada do sistema operacional não pode ser acelerada. Stratikopoulos et al. (2018) tratam esta questão, buscando responder à este questionamento, ao introduzir "*FastPath*". Esta solução transporta integralmente ao *FPGA* as camadas de acesso à memória do sistema operacional, colocando efetivamente o FPGA em uma configuração *in-datapath*. Através deste sistema, todas as operações de acesso à memória passam pelo FPGA — sendo assim aceleradas, em virtude da implementação em lógica. Um protótipo foi implementado com o FPGA Zynq 7000 Xilinx®e testado através do *Flexible IO* (FIO) *benchmark*. O atraso de leitura é reduzido em 67 %; e escrita, em 71 %, implicando que *FastPath* está em até 40 % da banda máxima teórica do SSD utilizado.

FastPath aumentou a banda ao transportar camadas de acesso à memória para o FPGA. No entanto, este não transplantou o próprio *driver* NVMe, presente no *host* em lógica programável. Com o intuito de melhorar a banda além do proposto em FastPath, Zhang et al. (2019) implementaram em um Xilinx® Zynq 7z045® o *driver* NVMe em uma plataforma customizada. Esta plataforma conta com dois canais independentes de fibra óptica para acesso aos SSDs e sistema de arrefecimento próprio — efetivamente configurando-a assim como uma memória de

acesso óptico. Esta memória acelerada por FPGA desempenhou leituras e escritas sequenciais dez vezes melhores que o original. Também houve uma redução de atraso em processamentos de requisições do usuário em 80 % contra o *driver* original. No Quadro 9 é disponibilizado um consolidado dos artigos apresentados nesta seção.

Quadro 9 – Artigos da seção de aceleração

Ano	Autores	Trabalho Realizado	Absorção
2014	Song, Y. H. et al.	*Cosmos OpenSSD: A PCIe-based Open Source SSD Platform*	Plataforma de testes FPGA para testes em invólucros *flash*
2011	Cai, Y. et al.	*FPGA-Based Solid-State Drive Prototyping Platform*	Plataforma de testes FPGA para aceleração de erros em invólucros *flash*
2016	Yoshimi, M. et al.	*Accelerating BLAST Computation on an FPGA-enhanced PC Cluster*	Aceleração em FPGA para algoritmo de processamento intenso no campo da biologia/medicina
2017	Jun, S. et al.	*Terabyte Sort on FPGA-Accelerated Flash Storage*	Aceleração em FPGA para algoritmo de *sorting*
2018	Jun, S. et al.	*GraFBoost: Using Accelerated Flash Storage for External Graph Analytics*	Aceleração em FPGA para algoritmo de processamento de grafos
2018	Li, D. et al.	*CISC: Coordinating Intelligent SSD and CPU to Speedup Graph Processing*	Aceleração em FPGA coordenada com CPU para processamento de *Minimum Spanning Tree*
2015	Rouhani, B. D. et al.	*SSketch An Automated Framework for Streaming Sketch-Based Analysis of Big Data on FPGA*	Aceleração em FPGA híbrida para computação de matrizes de grande ordem
2014	Jun, S. et al.	*Scalable Multi-Access Flash Store for Big Data Analytics*	Rede de FPGAs para sistema de aceleração para *Big Data*
2014	Ouyang, J. et al.	*SDF: software-defined Flash for Web-Scale Internet Storage Systems*	SSD específico para site de busca *online*
2016	Coughlin, M. et al.	*Apps with hardware: enabling run-time architectural customization in smart phones*	Conceito FPGA embarcado em celular
2018	Stratikopoulos, A. et al.	*FastPath: Towards Wire-speed NVMe SSDs*	Implementação em *hardware* de módulos do *kernel* para aceleração
2019	Zhang, J. et al.	*Design and Implementation of Optical Fiber SSD Exploiting FPGA Accelerated NVMe*	Implementação do *driver* NVMe em FPGA

Fonte: Autor.

O artigo (Cai et al., 2011) de 2011 ainda é relevante, pois este caracteriza erros da *flash*, propõe um novo FTL e faz uma demonstração do emprego de um FPGA junto com esta tecnologia de memória - aproximando-se do projeto proposto. Já os artigos (Cai et al., 2015b), (Cai et al., 2017b) e (Zhang et al., 2017) estudam os erros que podem acontecer nas memórias *flash*, tentando mitigar os efeitos. Outrossim, os artigos (Nam et al., 2011), (JUN et al., 2014), (KOMSUL; MCEWAN; MIR, 2014) e (BHIMANI et al., 2018) implementam controladores de SSD ou RAID e testam em plataforma genérica a base de FPGA. (YANG et al., 2019) cria um

sistema de reorganização de blocos para reduzir o WAF. Por último, os artigos (KANG et al., 2013) e (MIN et al., 2015) procuram propor novos FS e FTL para usos genéricos. Ambos os artigos (SONG et al., 2014) e (Cai et al., 2011) propõem plataformas de prototipação para *flash*, sendo esta última mais específica para aceleração de testes e exploração de algoritmos FTL e da topologia *flash* - ainda propondo um de exemplo de uso. Já os artigos (Yoshimi; Wu; Yoshinaga, 2016), (Jun; AND, 2017), (JUN et al., 2018), (LI et al., 2018), (Rouhani et al., 2015), (JUN et al., 2014), (STRATIKOPOULOS et al., 2018) e (Zhang et al., 2019) provam que um FPGA pode acelerar e/ou auxiliar a CPU com computação *in-datapath* de algoritmos de alto consumo de recursos (exemplo, algoritmos *greedy*). (OUYANG et al., 2014) propõe uma alternativa específica à SSD, tendo em vista um banco de dados, em parte eliminando gerenciamentos da *flash* desnecessários para este objetivo.

(SONG et al., 2014) é uma plataforma genérica e versátil para integração de FPGA e *flash* aplicado em vários artigos apresentados, como em (KANG et al., 2013), (MIN et al., 2015) e (LI et al., 2018). (SONG et al., 2014) é a plataforma que mais se semelha ao projeto futuramente proposto, porém carece em certos aspectos, como a especialização em aceleração de acesso, entre outros (mais detalhes no capítulo Metodologia). Como pode-se observar, (SONG et al., 2014) é utilizado em múltiplos artigos para implementar as provas de conceitos, sendo bem sucedido ao servir como referência - o que sugere o sucesso da proposta apresentada neste documento. Esta plataforma é composta somente do *hardware* além de ser paga, existindo uma vasta comunidade de pesquisadores que adotaram esta tecnologia. Outros recursos são disponíveis como orientações genéricas, casos de uso e até mesmo tutoriais genéricos. Outrossim, (Cai et al., 2011) é utilizado em (Cai et al., 2015b), sendo menos adotado que o artigo mencionado anteriormente.

Memórias *flash* podem ser operadas por FPGAs, conforme visto nos artigos (Cai et al., 2015b), (Cai et al., 2017b), (Zhang et al., 2017), (Nam et al., 2011), (JUN et al., 2014), (KOMSUL; MCEWAN; MIR, 2014), (KANG et al., 2013), (MIN et al., 2015), (SONG et al., 2014), (Cai et al., 2011), (Yoshimi; Wu; Yoshinaga, 2016), (Jun; AND, 2017), (JUN et al., 2018), (LI et al., 2018), (Rouhani et al., 2015), (JUN et al., 2014), (OUYANG et al., 2014) e (COUGHLIN; ISMAIL; KELLER, 2016)[1], proporcionando versatilidade e poder computacional de borda, especializado ao seu emprego. Faz-se menção especial ao artigos (STRATIKOPOULOS et al., 2018) e (Zhang et al., 2019), que obtiveram uma aceleração de acesso à memória. No entanto, (STRATIKOPOULOS et al., 2018) obteve esta aceleração por implementar parte do *kernel* em *hardware*; enquanto que (Zhang et al., 2019) implementou o *driver* do NVMe em FPGA. Outrossim, FPGA em muitos casos é a implementação mais barata, considerando o custo-benefício.

Como pode ser observado, tecnologias *flash* e suas camadas de abstração são uma linha de pesquisa muito atual. Igualmente observável, FPGAs são utilizados cada vez mais para resolver problemas complexos de processamento, sendo frequentemente associados à memória

[1] Todos os artigos descritos anteriormente, exceto por um único.

flash para testar e/ou implementar diversas funções, tais como controladores SSD, RAID ou mesmo FTLs. Nesta tendência, o projeto proposto se situa e será apresentado no capítulo de Metodologia. Dando-se por terminada este Estado da Arte, passa-se ao capítulo de Metodologia. Neste será detalhada a metodologia empregada para o desenvolvimento, implementação e teste do projeto proposto.

4 METODOLOGIA

O presente capítulo apresenta itens que compõem a metodologia do sistema proposto. Conceitos, métodos, arquitetura, topologia e fluxogramas serão descritos para a especificação do sistema. Conforme mencionado no capítulo de Introdução, este projeto trata majoritariamente da descrição de lógica programável (HDL). Sendo assim, mesmo que o *hardware* não seja o enfoque do projeto, este precisará ser especificado para o sucesso. Detalhes da metodologia empregada estão na seção de Método de Desenvolvimento.

4.1 MÉTODO DE DESENVOLVIMENTO

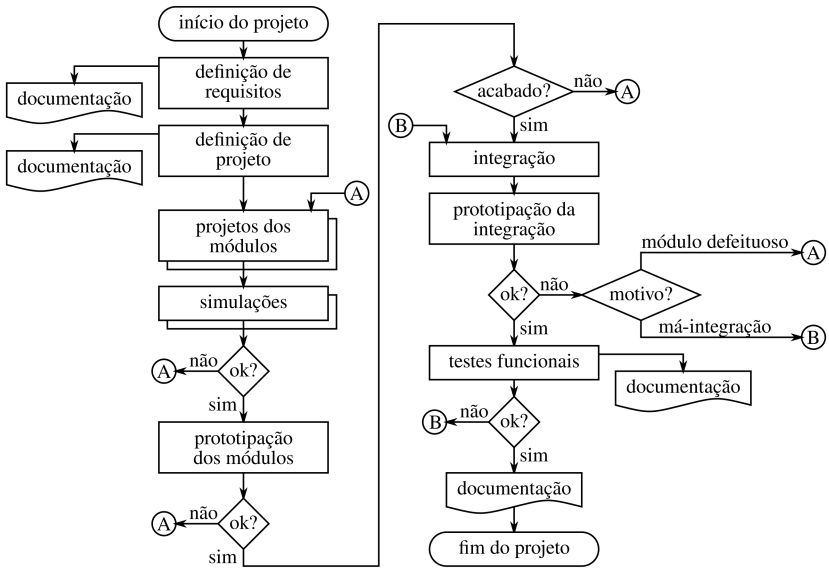
Nesta seção será apresentada a metodologia empregada em vistas do desenvolvimento do projeto proposto, que também será referido por projeto final. Segundo (Cohen, 1999), uma boa metodologia de codificação VHDL deve: respeitar as regras da linguagem, especificadas pelos padrões IEEE; gerar códigos de uma aparência comum e similar a outros códigos, com o objetivo de aumentar a familiaridade entre módulos diferentes; ser facilmente legível e mantido, a fim de aumentar a compreensão não somente do autor, mas como qualquer pessoa que leia; ter resultados dentro do esperado, tanto para a geração de códigos de descrição comportamental como para códigos sintetizáveis; evitar estruturas obsoletas, utilizando preferencialmente estruturas contemporâneas e recomendáveis; e coeso, significando que funções, procedimentos e declarações de tipos recorrentes devem ser definidos em pacotes. Adicionalmente, este livro complementa que códigos: simuláveis devem ser eficientes em simulação; e sintetizáveis, obedecer as regras de síntese do fabricante. Entende-se que nesta metodologia de codificação proposta pelo livro, fundamentalmente residem as premissas de portabilidade, modularidade e flexibilidade. Estas premissas significam que o código deve: funcionar para qualquer tecnologia; operar com qualquer fabricante; e ser construído por partes independentes, com interfaces padrões e consequentemente substituíveis. Reconhecendo estas premissas como de bom valor, a linguagem de descrição de *hardware* VHDL será empregada para este projeto proposto.

Para qualquer desenvolvimento de projeto de lógica programável, é necessário inicialmente ser feita a escolha do fabricante de FPGA. Pois cada fabricante de FPGA possui sua própria ferramenta de desenvolvimento. Para o projeto proposto foi escolhido o fabricante Xilinx®, pois este é o atual líder do mercado de FPGAs e possui ferramentas muito bem conceituadas e tradicionais, tais como Xilinx Vivado®, simplesmente referida aqui como Vivado. Esta ferramenta faz parte de um pacote nomeado *Vivado HLx Editions*®, que contém também um ambiente de Síntese de Abstração de Alto Nível (*High-Level Synthesis*, HLS), que permite a geração de códigos de HDL em linguagens de alto nível. HLS é voltada para o projetista com menos experiência em elaborar códigos HDL, porém gera códigos menos otimizados (SHARA-

FEDDIN et al., 2014) e por este motivo não será utilizada neste trabalho. O Vivado também inclui Propriedades Intelectuais padrões - em inglês *Intellectual Proprieties* (IP). O diagrama de blocos, sendo a interface gráfica do Vivado, não é necessária para o uso destes IPs, estas são também instanciáveis em código. No contexto de IPs, *soft* significa um *hardware* implementado utilizando a lógica programável do FPGA; e *hard* um bloco de primitiva implementado no projeto do *hardware* FPGA. Alguns blocos *hard* são exigências para certos IPs, como o IP de PCIe que acompanha o FPGA da Xilinx® - isto não significa a inexistência de soluções *soft* de PCIe, mas simplesmente não são acompanhadas com o FPGA em questão. Certas empresas focam parcialmente ou exclusivamente no fornecimento de IPs criptografados (INTELLIPROP, 2019a). Soluções de IP *soft* de terceiros são caras e voltadas para o mercado de grande porte - consequentemente não serão utilizadas neste projeto. Faz-se importante frisar que, para descrições de *hardware* ligeiramente complexas, a síntese LTR, a implementação LTR e geração de *bitstreams* são intensas e demoradas, podendo levar em alguns casos até mais de um mês. Por isto, é importante que os módulos sejam testados individualmente, para ter-se uma ideia dos seus funcionamentos.

Partes do projeto que funcionam podem ser integradas ao projeto proposto, sendo assim mais certeiro quanto a funcionalidade deste. Por este motivo, a metodologia se baseia em: determinação de requisitos; projetos dos módulos, para testar pequenas partes do sistema para poder integrar gradativamente; simulações e testes reais destes módulos; integração dos módulos, a fim de formar um projeto macro, que será o projeto proposto; e aplicação das métricas, medindo o sucesso do projeto. A Figura 19 apresenta a metodologia de projeto elaborada que será empregada para o desenvolvimento do projeto.

Figura 19 – Metodologia empregada no projeto



Fonte: Autor.

Na Figura 19 é possível compreender melhor o fluxo dos trabalhos. O projeto inicia-se pela definição dos requisitos e definições de projeto. Após todas as definições serem feitas, os módulos são elaborados e testados individualmente. Os testes são feitos em simulação, antes de serem prototipados na placa de avaliação. Caso um destes testes falharem, voltar-se-á ao projeto do módulo em questão. Assim, se constrói gradativamente o projeto proposto, por módulos, até que este possa ser integrado e testado na mesma placa de avaliação. Caso o projeto proposto não funcionar é estimado o motivo, seja por má integração dos módulos ou por módulos "defeituosos", significando aqueles módulos que não cumprem a função a qual foram projetados. Caso os módulos estejam mal integrados nas instanciações, a solução é corrigir o problema no projeto macro; caso seja o módulo esteja "defeituoso", deverá ser retornado ao projeto do módulo, passando por todo o fluxo novamente. Com o projeto proposto funcionando, medir-se-á o sucesso conforme os requisitos e as métricas. Todas as simulações, sejam de estímulos simples, de *test bench*, de temporização ou de qualquer outra natureza, serão feitas no próprio ambiente fornecido junto com o Vivado. Para módulos pequenos são feitas simulações simples de estímulos manuais de entrada, sendo para módulos grandes são feitas simulações complexas de *test bench*, que geram estímulos de um grande número de combinações de entrada e em alguns casos a verificação automática de resultados. Ao longo de todo o processo será gerado documentação com: textos sobre as definições de requisitos e projetos; textos sobre os módulos, suas especificações, funcionalidades, interfaces e como operá-los, similarmente como é feito para primitivas; e textos sobre o projeto, sobre o funcionamento, operação e como implementá-lo na placa de avaliação. Toda esta documentação busca relatar o andamento, as dificuldades encontradas e as conclusões técnicas para projetos futuros e manutenção deste. Exemplo de preenchimento de especificação para um módulo se encontra nos apêndices. A metodologia para a determinação das especificações para estes módulos será indutiva (*top-down*). Dando-se por concluída esta seção de Metodologia de Desenvolvimento, definir-se-á os requisitos na próxima seção.

4.2 REQUISITOS

A razão de realizar este projeto de pesquisa, é fundamentalmente a aplicação. Porém, afirma-se que a fim de garantir melhoras sensíveis nesta, focar-se-á na lógica programável e métodos de habilitação tecnológica por *hardware*, para a possibilidade de melhoria do acesso à memória e novas linhas de pesquisa para o estado da arte. A metodologia será aplicada para uma solução específica, servindo simultaneamente para a validação do conceito, feito a partir de métricas. A partir do contexto de pesquisa na área do projeto, feito no capítulo de Estado da Arte, se posiciona o projeto proposto. As premissas de portabilidade, modularidade e flexibilidade serão importadas como requisitos deste projeto. A flexibilidade permite a expansão da arquitetura proposta, tendo em vista projetos futuros. O requisito de portabilidade serve como justificativa para o não-uso do diagrama de blocos do Vivado. Mesmo que este gere código VHDL, a parte

gráfica é compatível unicamente com o fabricante Xilinx®, não sendo compatível com nenhuma outra ferramenta de qualquer outro fabricante. Já o requisito de modularidade serve para justificar a construção do projeto proposto por módulos. Trocando o FPGA ou a ferramenta, o módulo que deixe de funcionar pode ser substituído por outro módulo de interface e funcionamento idênticos, sendo ajustado a parte incompatível - exemplo de tais módulos são instanciações de primitivas, que variam conforme o FPGA.

O uso de placas de desenvolvimento dá a liberdade da não-preocupação com aspectos que não pertencem ao escopo do projeto, além da disponibilidade de suporte do fabricante. No entanto, caso as escolhas destas placas prontas sejam mal feitas, o projeto completo poderá estar comprometido - como exemplo de uma destas escolhas críticas, uma placa de avaliação cujo FPGA não tenha lógica o suficiente para suportar o projeto. Por este motivo, é descartado o uso da placa OpenSSD Cosmos+, cujo artigo foi apresentado no capítulo de Estado da Arte. A Cosmos+ não possui lógica suficiente para a implementação do projeto proposto, nem outros requisitos essenciais que estão definidos nesta seção. No projeto proposto serão utilizadas duas placas: uma placa FPGA de avaliação e outra de compatibilização para SSD PCIe NVMe. Para esta última, a FPGA Drive cumpre o ofício, sendo uma placa de compatibilização do conector FMC e conector M.2 PCIe, a fim de permitir a ligação de dois SSDs com uma placa de avaliação. Tal placa é perfeita para o projeto proposto e será utilizada. Nem todas as placas de avaliações possuem compatibilidade com a FPGA Drive, algumas podendo acessar somente um SSD e outras sendo completamente incompatíveis. A Figura 20 apresenta esta placa de compatibilização.

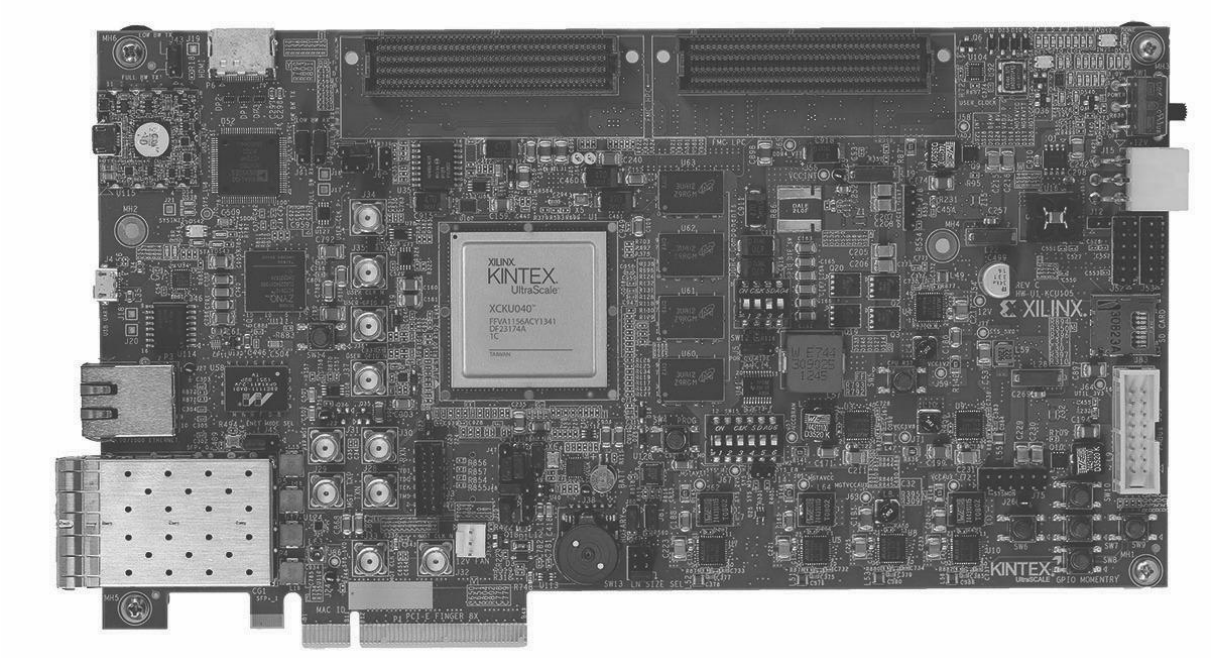
Figura 20 – Placa auxiliar de desenvolvimento FPGA Drive

Fonte: Adaptado de Opsero Electronic Design (2019).

Para a realização do projeto proposto será utilizada uma placa de avaliação pronta. Para isto deve ser previsto uma quantidade mínima de lógica disponível para implementar o projeto proposto. A estimativa de ocupação da lógica no FPGA é feita preliminarmente baseado em módulos que deverão ser empregados e informações recolhidas na Internet sobre a ocupação da lógica destes módulos, ou de módulos similares, em FPGAs similares. A placa de avaliação deve possuir *hardware* suficiente para suportar o projeto proposto e os projetos posteriores. O FPGA deve ter menos de 60 % de ocupação, pois acima disto, a ferramenta de compilação demora

demasiadamente para gerar o *bitstream* para prototipação. Caso fosse considerada uma ocupação de 100 %, estima-se que a prototipação completa levaria até mais de um mês para ser compilada. Estas estimativas estão presentes nos Apêndice A e Apêndice B — considerando uma ocupação de 60 % e margem de erro de 20 %. A especificação mínima para cumprir os requisitos está listada no Apêndice C. A placa de avaliação que melhor encaixou-se com esta especificação mínima foi a KCU105 (XILINX, 2019e). A KCU105 contém: um FPGA da família Kintex e tecnologia UltraScale, o XCKU040; dois conectores FMC, um de baixa densidade (LPC) e outro de alta densidade (HPC); quatro CIs de memória DRAM DDR, totalizando aproximadamente 2 GB; e conector PCIe *edge connector*, compatível com a geração 3 de oito vias. Faz-se referência a seção 2.2 *Peripheral Component Interconnect Express*, que explicita a compatibilidade reversa (*backward compatibility*), significando que a geração 3 da KCU105 poderá funcionar somente com versões acima ou idêntica desta geração de PCIe. A especificação completa da lógica do FPGA contido na KCU105 (XCKU040-2FFVA1156E) está disponível em (XILINX, 2019d) e no Apêndice D. A placa FPGA Drive é compatível com a KCU105, estando disponível ambos os SSDs para o acesso. Na Figura 21 é apresentada esta placa de avaliação.

Figura 21 – Placa de avaliação FPGA KCU105



Fonte: Adaptado de Samtec (2019).

Neste projeto de pesquisa, o Ext4 será o único FS de compatibilidade obrigatória. O Ext4 é moderno, com uma grande comunidade para o suporte, de uso mais difuso e principalmente relevante para um projeto de pesquisa: com o código aberto. Também neste contexto, será levada em consideração a especificação base 1.3d de NVM Express™ (NVM EXPRESS INCORPORATED, 2019) para a realização. No Quadro 10 consta um agrupamento dos requisitos aqui especificados, seguidos de suas breves justificativas.

Quadro 10 – Requisitos do projeto

Requisito	Justificativa
Uso de placas prontas com recursos de *hardware* suficientes	Despreocupação consciente relativo ao *hardware*
Uso de FPGA com recursos lógicos suficientes	Não permitirá a implementação caso não houver recursos suficientes
Codificação por VHDL	Emprego das premissas de flexibilidade, portabilidade e modularidade em codificação
Flexibilidade	Possibilidade de expansão para projetos futuros
Portabilidade	Compatibilidade com diferentes fabricantes
Modularidade	Facilidade de implementação e compreensão simplificadas
Compatibilidade com Ext4	Necessário o conhecimento do esquema de mapeamento do FS
Compatibilidade com NVM Express™ 1.3d	Necessária a escolha da versão da especificação, sendo esta a mais recente

Fonte: Autor.

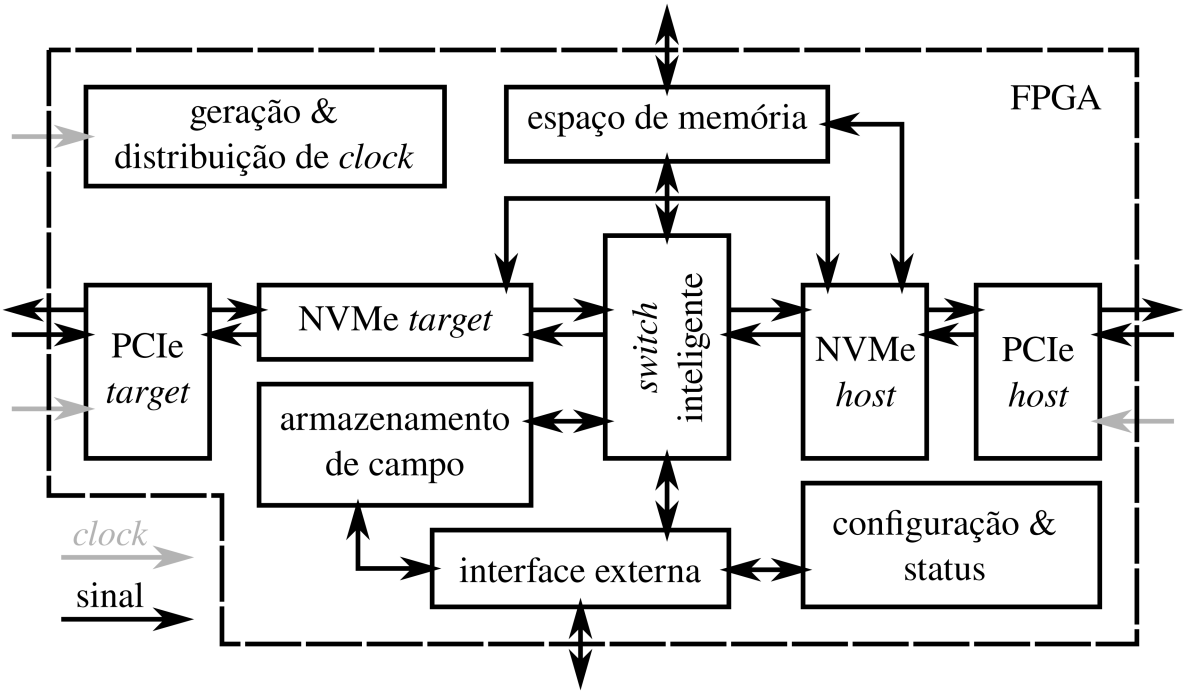
Concluída esta definição de requisitos, o projeto proposto será explanado na próxima seção de Sistema Proposto. Nesta seção será definido em detalhes o conteúdo e teor previstos deste.

4.3 SISTEMA PROPOSTO

No capítulo de Introdução, foi especificado o objetivo deste projeto de pesquisa, que consiste em propor um conceito de habilitação de acesso à memória. Para isto, uma *bridge* será implementada como prova do conceito de habilitação de acesso. Durante a pesquisa relacionada ao estado da arte, não foi encontrado um artigo que propôs uma solução para gerar estatísticas sobre campos NVMe, nem tecnologia de *switch* NVMe[1]. Conforme demonstrado no capítulo de Estado da Arte, diversas aplicações são aceleradas com implementações em *hardware*, dado que este fornece uma velocidade de banda próxima à capacidade teórica (*wirespeed*). Logo, percebeu-se a possibilidade de interceptação dos comandos em *hardware* para a interpretação do conteúdo deste. O comando interpretado poderá então ser disponibilizado para a otimização de acesso e estatística, com comunicação externa. Na Figura 22 abaixo, é apresentado o planejamento da arquitetura desta *bridge*.

[1] Apêndice H trata da publicação feita, especificamente do *switch* NVMe, tópico do capítulo de Análise de Resultados e deste sistema proposta.

Figura 22 – Arquitetura para a *bridge*



Fonte: Autor.

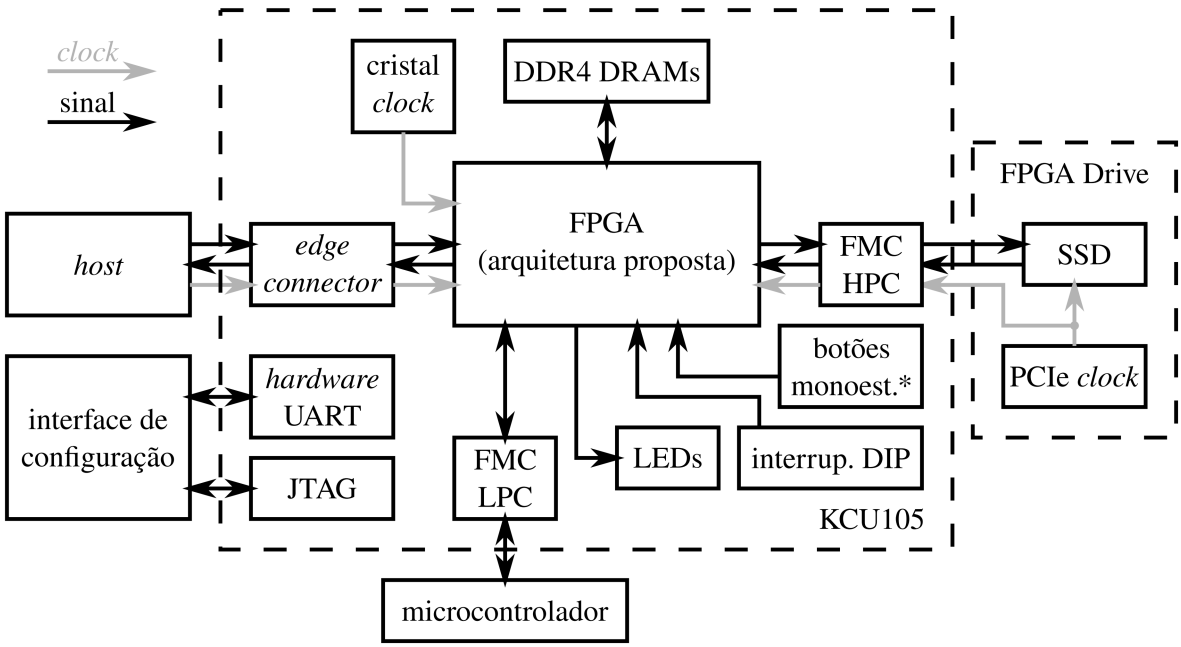
Primeiramente, esclarece-se o fluxo dos comandos NVMe dentro da lógica programável, em seguida, faz-se apontamentos sobre os módulos em questão. O comando de dado (requisição do *host*) entra no FPGA através do módulo da PCIe *target*, expondo a instrução NVMe em um formato intermediário, ao próximo módulo: o NVMe *target*. Este converte deste formato para pacotes NVMe, a ser definido no capítulo de Análise de Resultados, com o objetivo de simplificar a complexidade do protocolo NVMe. Após serem processados pelo *switch* inteligente, estes pacotes são enviados ao módulo de NVMe[2] *host*, que converte novamente ao formato intermediário, readaptando à especificação NVMe. A partir deste formato, o módulo de PCIe *host* adequa ao protocolo PCIe, fornecendo os dados à SSD. Entende-se então, que o *host* comunica-se com o *target*; e vice-versa. Logo, por *target* compreende-se qualquer módulo que emule parcelas do SSD, ou dispositivo NVMe. Outrossim, os módulos de PCIe poderão ser formados por primitivas configuradas especificamente para estas atribuições. Uma vez entendido o fluxo de comandos NVMe dentro da lógica passa-se à dissertar sobre a funcionalidade dos módulos.

O módulo de *switch* inteligente é o principal em toda a arquitetura, pois este cumpre com as tarefas de: captura de campo, contador de comandos; e arbitração. Captura de campos e contador de comandos NVMe servem para gerar estatísticas tanto do protocolo NVMe, como do padrão de uso do FS. A arbitração de comandos é função primária que define qualquer *switch*, e define-se pela discriminação de comandos, neste caso: os comandos que são enviados ao

[2] Módulos NVMe podem ser implementados através de IP comerciais IntelliProp (2019d)

host e os comandos que são enviados ao *target* alternativo. Todas estas funcionalidades são configuráveis através da interface externa em tempo real. O pacote NVMe discriminado em seus diferentes campos pode ter seu conteúdo interpretado, baseando-se no leiaute do FS Ext4 - em acordo com a premissa deste projeto. Sendo assim, é identificado o tipo de bloco de dados sendo lido ou escrito pelo *host*. Pode-se, então, elaborar diversos processamentos diferentes para estes comandos discriminados, contudo, propõe-se aqui somente um de exemplo a seguir. A captura de campo, juntamente com o contador de comandos, gera estatísticas de comandos, podendo recriar um histograma dos dados frequentemente acessados, assim habilitando a área de pesquisa. Esta mesma estatística pode ser utilizada para o processamento de comandos discriminados: um armazenamento de blocos da memória frequentemente acessados - em um *cache* de blocos na memória DRAM. Os blocos armazenados podem ser reutilizados, uma vez que o *host* faça uma nova requisição destes, sendo respondida imediatamente pelo próprio *switch* inteligente. Algumas das consequências imediatas são: a extensão da vida útil da SSD - por estar sendo menos requisitada; a aceleração da SSD, pois esta ficaria mais tempo ociosa, deixando o sistema de GC ser executado neste ócio; e a aceleração na resposta, por não precisar retornar da SSD, sendo mais rápida pelo tipo da memória utilizada. O módulo de armazenamento de campo armazena os campos capturados pelo *switch* inteligente, fornecendo assim os mesmos à interface externa. Um módulo de controle e estado do sistema é previsto para configuração e gerenciamento do sistema. Este bloco de controle comunica-se unicamente com a interface externa, que possui comunicações tanto com os GPIOs da placa KCU105, tanto com o microcontrolador externo. O módulo de interface externa implementa um protocolo de comunicação projetado especificamente para a tarefa de configuração e recuperação de dados. Esta interface é responsável pela a comunicação entre FPGA e microcontrolador, sendo que a programação do microcontrolador está fora do escopo deste trabalho. Aproveitando a RAM existente, o módulo de espaço de memória contém o controlador de DRAM, podendo assim fornecer memória de acesso rápido aos módulos de *switch* inteligente e NVMe *host*. Prevê-se unicamente a necessidade do mesmo se comunicar com o NVMe *host*, pois este irá emular um *host* - sendo que todas as filas do NVMe são mantidas no *host*. Outrossim, um módulo de gerenciamento e distribuição de *clock* é previsto, contendo uma primitiva de divisão de *clock* e de *buffers* de distribuição global. Esta arquitetura pode ser implementada dentro do FPGA, na placa KCU105. A arquitetura do *hardware* a ser utilizado para a implementação da *bridge* é apresentado na Figura 23.

Figura 23 – Arquitetura do *hardware* da *bridge*



Fonte: Autor.

* monoestáveis

No teste da *bridge*, um computador de testes incorpora a função de *host*, feita através do *edge connector* da PCIe. Outro computador sere como interface de configuração do sistema proposto, através da UART e JTAG. Os recursos do usuário disponíveis da KCU105 estão listados no Apêndice E. Estes recursos são aproveitados para comunicação com o sistema e configuração do sistema. Do outro lado existe a placa FPGA Drive, que adequa a SSD através do conector FMC HPC disponível na KCU105. A FPGA Drive é transparente para este projeto, servindo meramente como um adaptador. Os invólucros de memória DRAM de tecnologia DDR4 soldados na placa KCU105 servem como memória volátil, totalizando 2 GB de armazenamento. Um cristal de geração de *clock* fixo em 300 MHz alimenta o bloco de gerenciamento de distribuição de *clock*. O conector FMC LPC é utilizado para comunicação com o microcontrolador externo. A placa microprocessadora utilizada será a STM32H743ZIT6U[3,4], incorporando a função de microcontrolador. Terminada esta seção de Sistema Proposto, passar-se-á ao Método de Avaliação, que proporá critérios para a medição do sucesso do projeto.

4.4 MÉTODO DE AVALIAÇÃO

Esta seção apresentará o método de geração de banco de dados para testes e os critérios de avaliação do projeto. Tais critérios irão variar conforme o que está sendo avaliado. Para os módulos, cada simulação será bem-sucedida se este cumprir a função ao qual foi projetado.

[3] Fabricado pela STMicroelectronics™, com microcontrolador ARM® Cortex®.
[4] Manual do usuário disponível em (ST, 2019).

Isto inclui, por exemplo para um módulo de gerenciador de *clock* com restrições de *jitter*, uma simulação adicional de cumprimento destas restrições na distribuição interna no FPGA. Estas restrições deverão constar na especificação do módulo. Na prototipação destes, deverá ser feita uma adaptação para poder processar na placa KCU105. Esta adaptação não é objeto de avaliação, consequentemente não será avaliada. Caso o módulo seja simples, esta etapa não será necessária. O grau de simplicidade do módulo deverá ser declarado em sua especificação. Nesta prototipação poderão ser testados múltiplos módulos em conjunto. Alguns módulos não podem ser testados separadamente, tal como o módulo de comunicação por PCIe, necessitando o módulo de instanciação do *buffer* específico do *clock*, evidenciando um sequenciamento de testes. Uma vez que todos os módulos foram testados e prototipados na placa KCU105, o projeto poderá ser integrado e prototipado. Identicamente à simulação dos módulos, o protótipo do projeto proposto passará o teste se este cumprir a função ao qual foi projetado. Percebe-se que a função dos módulos e do projeto servem como base dos critérios de avaliação. Consequentemente, estas funções devem ser bem definidas. As funções dos módulos são especificadas em suas documentações individualmente, vide a importância da documentação. Exemplo de especificação de módulo consta no Apêndice G.

No Apêndice F é apresentado o método de geração de comandos pseudo-aleatórios. O intuito deste método é obter um conjunto de comandos de submissão (direção *host* para SSD) de padrão NVMe, cujos conteúdos são conhecidos, havendo a possibilidade de gerar comandos específicos - com conteúdos administráveis. O resultado dos comandos gerados seriam arquivos de texto em conjunto com sua documentação. Arquivos de texto (“.txt”) podem ser introduzidos em simulações do tipo *test bench*. O processo começa definindo-se um número N arbitrário de comandos. Para iniciar-se a contagem, é introduzida uma variável n que conta o número de iterações realizadas. Um bloco de função de comando pode ser operado pseudo-aleatoriamente, ou por instrução direta. Por função de comando entende-se: de leitura, de escrita, ou de qualquer outra definida na especificação NVMe. Outras opções peculiares de cada tipo de comando é operada de forma similar, gerando ao mesmo tempo a documentação. Caso a função do comando necessite conteúdo adicional, por exemplo de escrita, um conjunto de *bits* pseudo-aleatórios poderão ser fornecidos. Isto se repete a cada iteração, incrementando o contador a cada vez, até que se chegue ao valor determinado de N.

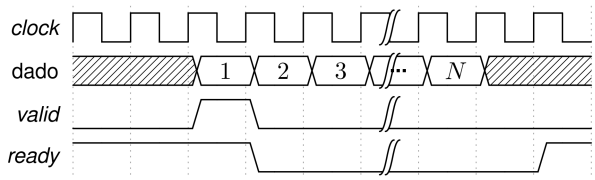
Dado que o sistema de arquivos que será utilizado é o Ext4, o GNU/Linux deverá ser o sistema operacional para testes, por ser aberto e difundido. Adicionalmente, o GNU/Linux é compatível com todas as ferramentas necessárias. Terminado este capítulo de Metodologia, inicia-se o capítulo de Análise de Resultados.

5 ANÁLISE DE RESULTADOS

O presente capítulo aborda resultados do projeto, sendo estes simulações de lógicas desenvolvidas. Em respeito ao requisito de flexibilidade estipulado, o planejamento destes módulos é facilmente expansível e modificável. Conforme estabelecido no Sistema Proposto, existem dois principais aspectos: sistema gerador de dados sobre o padrão de uso da SSD, para habilitar a pesquisa; e sistema habilitador de otimização de acesso à memória. Deste modo, serão analisados três grandes sistemas implementados: sistema de captura de campo; sistema de contagem de *opcode*; e sistema de *switch* NVMe, único sistema dedicado à habilitar a otimização de acesso à memória. Este capítulo começará com conceitos bases, nominalmente a definição do pacote NVMe; descrição dos arquivos para teste; como é feita em tempo real a introdução de configurações. Após esta exposição de conceitos bases, apresentar-se-á uma visão global dos diferentes sistemas citados; em seguida será apresentado o Módulo de Armazenamento de Campo. Resultados de simulações serão apresentados após a apresentação dos conceitos bases.

Para compreensão de toda a arquitetura, introduz-se o conceito de pacote NVMe. Esta definição de pacote facilita o fluxo de comandos, utilizada em todo o projeto. Tal pacote trata somente do comando NVMe, seja de submissão (SC) ou compleição (CC), serializado por *byte*, juntamente com um sistema de *handshake* — sinais *valid* e *ready*. Por não serem de interesse nem do ponto de vista de pesquisa, nem de otimização de memória, os sinais de controle do NVMe e os sinais de configuração do cabeçalho do dispositivo PCIe são tratados separadamente. A forma de onda de um pacote NVMe pode ser visualizada em Figura 24. Para ilustração, os números dentro do sinal "dado" indicam os *bytes* considerados. O sinal *valid*, provido pelo remetente, indica que o *byte* é o primeiro e é válido; já o sinal *ready* indica que o destinatário está apto a receber pacotes. A transação (*stream*) começa quando *valid* e *ready* reconhecem o primeiro *byte*. A transação continua até o final do pacote (N). Em comandos de submissão, $N = 64$; e compleição, $N = 16$.

Figura 24 – Transmissão de um pacote NVMe



Fonte: Autor.

Considerando pacotes, é necessário a elaboração de alguns padrões para as simulações. Tais pacotes são descritos em arquivos, conforme o Quadro 11. Seis pacotes diferentes representam todo o conjunto de casos necessários para os sistemas apresentados neste capítulo. Cada

arquivo é um pacote NVMe único, sendo dois arquivos com comandos de leitura. Informações relevantes destes arquivos constam no Quadro 11. Estes arquivos foram gerados segundo o método apresentado na subseção de Método de Avaliação, podendo ser manipulados para facilidade de análise.

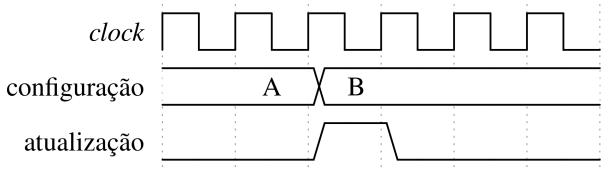
Quadro 11 – Informações relevantes sobre os arquivos de teste

Arquivo	**Função (*opcode*)**	**ID**	**SLBA**	**DSM**	**Resultado esperado AC_WORKLOAD**[1]
read1	leitura	0xB296	0x92DC3F96_11BDFCD6	0x82	SSD
read2	leitura	0xB397	0x07060504_03020100	0xFF	NVMe genérico
flush	*flush*[2]	0xF3C7	não existe	não existe	SSD
write	escrita	0xB690	0x07060504_030200FE	não existe	SSD
NVMe_SQ_B_01	proprietária	0xBB9e	não existe	não existe	SSD
NVMe_SQ_B_02	proprietária	0x2F15	não existe	não existe	SSD

1 Foco do sistema de *switch* NVMe.
2 Comando *flush* força que dados armazenados no *buffer* sejam escritos na memória não-volátil.
Fonte: Autor.

Outrossim, faz-se um esclarecimento sobre o Módulo de Interface Externa. Conforme a seção de Sistema Proposto, o Módulo de Interface Externa é utilizado para se comunicar com um microcontrolador, podendo assim receber configurações. Estas configurações se dão por um *byte* seguido de um sinal para sincronia, sintetizadas através do Módulo de Registrador de Configuração, interno à interface. A Figura 25 expõe o caso único de atualização de uma configuração genérica “A” para outra genérica “B”.

Figura 25 – Atualização de uma configuração



Fonte: Autor.

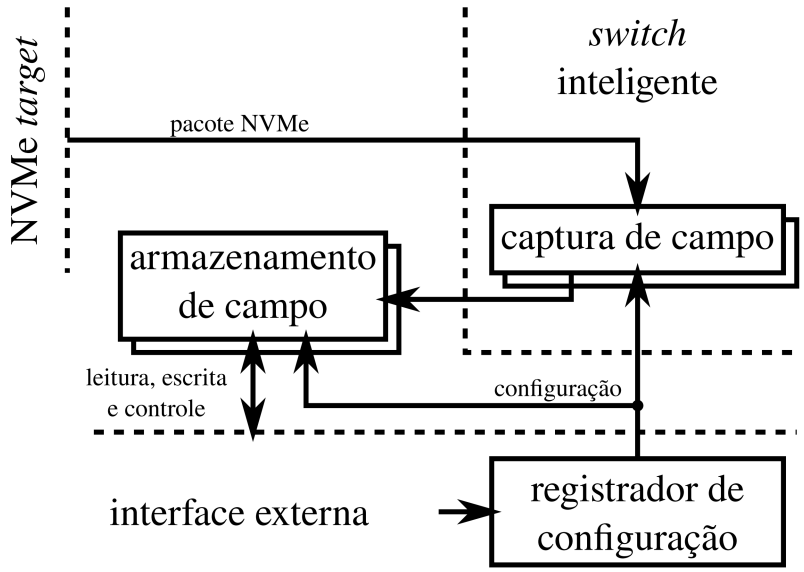
Em todos os sistemas, a forma de onda para configurações serão as mesmas. No entanto, os valores do sinal de configuração *config* irá variar entre sistemas, juntamente com seu significado. Concluído este último esclarecimento base, passa-se à análise do sistema de captura de campo.

5.1 SISTEMA DE CAPTURA DE CAMPO

O mecanismo para captura de campos NVMe e processamento posterior é uma das atribuições do *switch* inteligente — sendo o foco inicial do desenvolvimento. Para simulações

realistas deste sistema, estes campos capturados serão recebidos em tempo real pelo Armazenamento de Campo, que disponibilizará ao Módulo de Interface Externa. Este o Módulo de Captura de Campo aceita somente pacotes NVMe, conforme defino na Figura 24. Um campo capturado corresponde no máximo a 8 *bytes*, suficiente para armazenar qualquer campo, segundo a especificação NVMe. Já o Módulo de Armazenamento de Campo armazena os campos capturados, sendo sua a principal atribuição a instanciação e administração da memória volátil (BRAM) do FPGA. A Figura 26 ilustra simplificadamente como este sistema opera. Nota-se que as caixas representando os módulos de Armazenamento de Campo e Captura de Campo estão multiplicadas, evidenciando que este sistema poderá rodar em paralelo um número pré-configurado de captura de campos em um dado instante, em acordo com a especificação do módulo (Apêndice G).

Figura 26 – Sistema de captura de campo

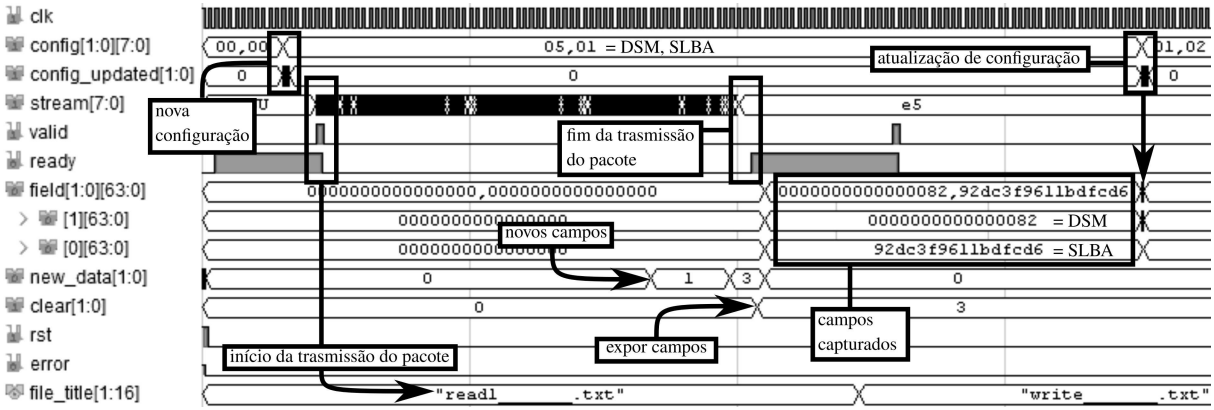


Fonte: Autor.

Com base na Figura 26, propõe-se o seguinte cenário de captura de campo seguido de seu armazenamento bem-sucedido. A Interface Externa disponibiliza uma nova configuração para a captura de um campo específico, indicando ao módulo em questão, que houve uma atualização. Esta nova configuração síncrona será enviada paralelamente aos módulos de Captura de Campo e Armazenamento de Campo, juntamente com o sinal de atualização. Este sinal de atualização implicará mudanças nas máquinas de estados de ambos os módulos interfaceados. Em um primeiro instante, o Módulo de Captura de Campo indica disponibilidade após chegar em seu estado de ócio; após alguns ciclos de *clock*, o Módulo de NVMe *Target* começa a enviar um novo pacote NVMe, indicando da mesma forma o começo do pacote. Supondo que o pacote enviado contenha o campo configurado, este é capturado e armazenado temporariamente em um registrador interno ao módulo. Simultaneamente, uma sinalização indica ao Módulo de Armazenamento de Campo o novo campo está disponível. Estando em ócio, o Módulo

de Registradores manda sinal para liberar o campo em questão. Uma vez que há campos armazenados no registrador, a Interface Externa consegue acessá-los por endereço. Neste cenário são percebidos alguns detalhes que podem não ocorrer como planejado, chamados de exceções. São exceções: O Módulo de NVMe *Target* envia comando enquanto que o Módulo de Captura de Campo está ocupado; O Módulo de Armazenamento de Campo não guarda a tempo o campo capturado; O Módulo de Armazenamento de Campo não possui mais endereços para o retenção. Não obstante, o sistema deve administrar tais exceções. Por administrar entende-se gerar sinais de erro, perda de campo armazenado ou indicação de memória cheia. Na Figura 27 é possível visualizar o Módulo de Captura de Campo funcionando.

Figura 27 – Módulo de Captura de Campo funcionando



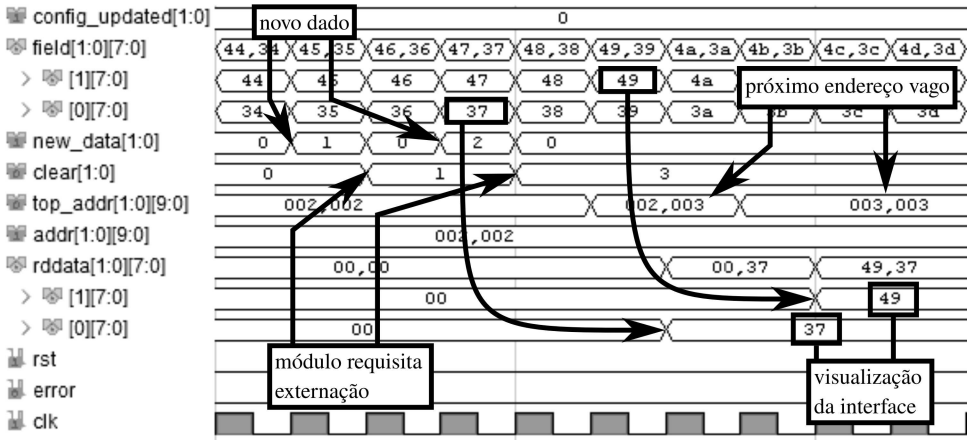
Fonte: Autor.

A Figura 27 apresenta o funcionamento de dois Módulos de Captura de Campo em paralelo, configurados de maneira diferente para o mesmo pacote NVMe. O módulo reconhece o sinal *config*, cujo número significa uma configuração diferente. Neste caso, 0x01 equivale à captura do campo Endereço Inicial de Bloco Lógico (*Start of Logical Block Address*, SLBA); e 0x05 capturar o *Dataset Management* (DSM). Os campos foram capturados no momento da indicação do sinal *new_data*; externados através do signal *clear* e externados pelo sinal *field*. O valor do sinal *field* corresponde ao valor esperado, conforme a configuração e a Quadro 11. Graças aos requisitos de flexibilidade e modularidade, este módulo é utilizado em outras partes do projeto, incluindo o sistema de contagem de *opcode*.

Devido ao funcionamento do mecanismo de armazenamento temporário do campo em registrador interno, o Módulo Armazenamento de Campo deverá coletar automaticamente o valor salvo em seu tempo de ócio. Foram previstos quatro aspectos chaves neste módulo, são estes: instanciações de BRAM; uma máquina de estados para manter a organização; uma máquina de estados para a coleta do novo campo a ser armazenado; e uma interface direta da BRAM à Interface Externa. Duas BRAMs de 36 *bits* são instanciadas em paralelo para interfacear os 8 *bytes* do Módulo de Captura de Campo, sendo 32 *bits* para dado e 8 *bits* para metadado. As memórias RAM são do tipo *true-dual port*, possibilitando duas portas independentes, sendo uma

porta utilizada exclusivamente para a Interface Externa. A outra porta é compartilhada entre o mecanismo organizador e o mecanismo coletor. A Figura 28 exibe o comportamento ideal do módulo ao armazenar um novo campo.

Figura 28 – Módulo de Armazenamento de campo, processando sinal de novo campo



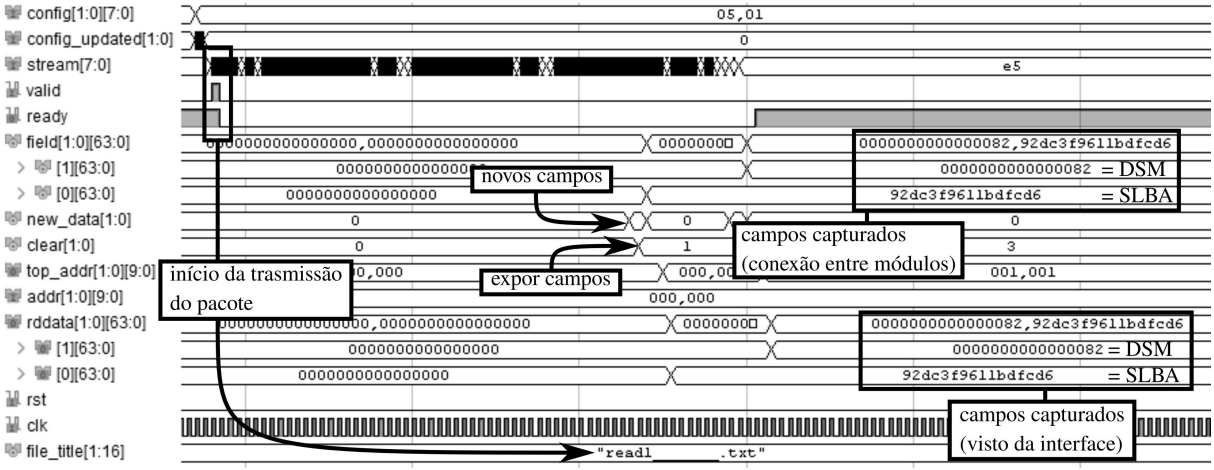
Fonte: Autor.

Na Figura 28 foram omitidas, por não serem usadas nesta simulação, as portas de leitura e escrita de metadado e escrita de dado. Nesta simulação, dados são gerados sequencialmente pelo sinal *field*, simulando o recebimento de novos campos. O sinal *field* foi resumido em um *byte* para facilitar a leitura da simulação. Os sinais de interação com a Captura de Campo, isto é *new_data* e *clear*, são operados pelo mecanismo coletor de campo. A figura apresenta dois módulos operando simultaneamente em paralelo, sendo possível ver que o dado capturado é armazenado e visualizado pela porta de leitura de dado (*rddata*), sinal pertencente à Interface Externa. O módulo também expõe a porta *top_addr*, cujo valor aponta para o próximo endereço vago. Uma vez comprovado o funcionamento deste módulo, passa-se à análise da integração entre os módulos referente ao sistema de captura de campo apresentados.

A integração consiste na instanciação dos módulos detalhados anteriormente, com as conexões realizadas conforme a Figura 26, apresentada no início deste capítulo. Um *test bench* similar ao do Módulo de Captura de Campo foi utilizado para teste de integração. O objetivo específico desta análise é observar a compatibilidade entre os módulos e dar prova que a interface está funcional. É relevante ressaltar, que para cada Módulo de Captura de Campo, haverá um correspondente de Armazenamento de Campo. Este objetivo será atingido se o campo selecionado for capturado pelo Módulo de Captura de Campo, subsequentemente estocado no Módulo de Armazenamento de Campo, podendo ser observado pela Interface Externa. Os nomes e significados dos sinais são os mesmos dos módulos anteriormente detalhados, porém faz-se necessário apontar alguns trocas. Os sinais de novo campo (*new_data*) e expor campo (*clear*) passam a ser internos, pois são comunicações respectivamente dos módulos de Captura de Campo ao Armazenamento de Campo e do Armazenamento de Campo ao Captura de Campo.

Feita esta precisão, analisa-se o cenário ideal de captura de um campo qualquer na Figura 29.

Figura 29 – Integração do sistema de captura de campo com posterior armazenamento



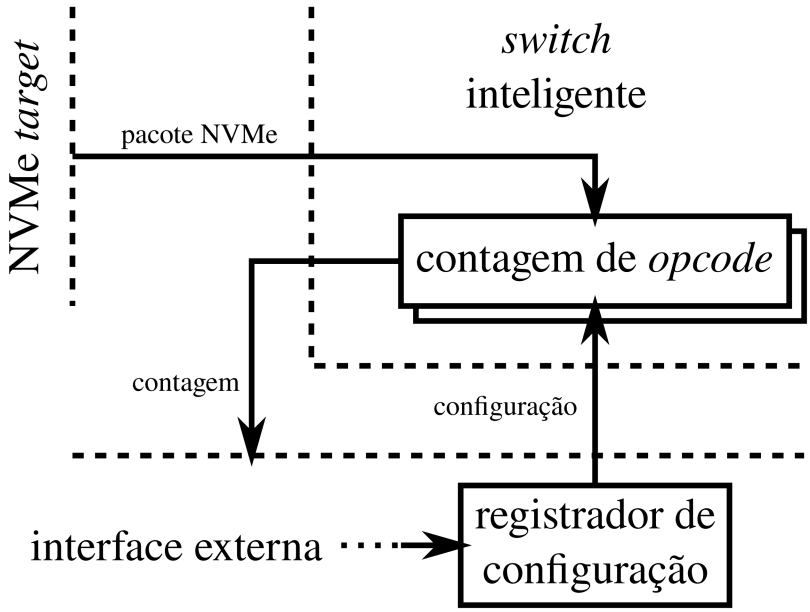
Fonte: Autor.

Na Figura 29 observa-se que a interação entre um par de módulos de captura e armazenamento de campo. Cada qual está configurado para DSM e SLBA. Estes campos são capturados e disponibilizados à Interface Externa. Feita esta última análise do sistema de captura de campo, passa-se ao segundo viabilizador de estatísticas, tratando da contagem de *opcode*s.

5.2 SISTEMA DE CONTAGEM DE *OPCODE*

O mecanismo contador de pacotes NVMe possui a atribuição de somar o número de pacotes recebidos, discriminando pelo *opcode* do comando NVMe. Por *opcode*, entende-se a função do comando, seja escrita, leitura, *flush*, entre outras listadas na especificação NVM Express Incorporated (2019). Algumas funções, como de leitura e escrita, possuem campos com valores numéricos, ao exemplo do SLBA, que também são interessantes para a discriminação. Pode-se configurar este sistema então, para capturar escritas, cujos blocos correspondem a alguma área de interesse no FS. Estas informações são essenciais para formar gráficos como histogramas e padrões de escrita para diferentes tipos de cargas de trabalho (*workload*). Este mecanismo supre, juntamente com o de captura de campo, a parcela de geração de estatísticas para este projeto de pesquisa. A Figura 30 apresenta o interfaceamento deste sistema com os módulos externos ao *switch* inteligente.

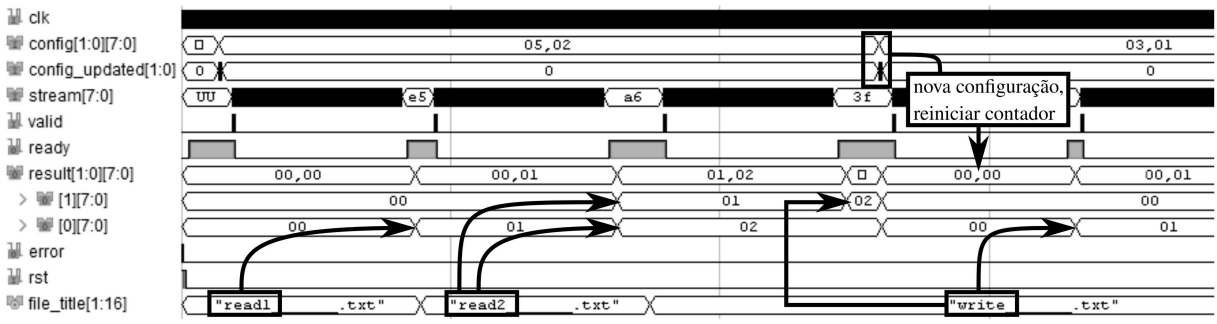
Figura 30 – Sistema de contagem de *opcode*



Fonte: Autor.

Tendo a Figura 30 como base, explana-se um cenário de contagem bem-sucedido. A Interface Externa disponibiliza uma nova configuração para a contagem de *opcode*, indicando ao módulo em questão, que a nova configuração está disponível. Em paralelo à configuração, o sinal indicativo de atualização reinicia o contador. Com o Módulo de Contagem de *Opcode* disponível, o Módulo de NVMe *Target* começa a enviar um novo pacote NVMe. O contador então compara dois critérios: a faixa em que o campo numérico se encontra[1] e o *opcode* - por exemplo, para uma configuração de escrita, cujo SLBA é superior a 0xAAAA e inferior a 0xEEEE. Caso ambos os critérios concordarem, o contador é incrementado e o resultado é imediatamente disponibilizado na Interface Externa. A operação do sistema descrito pode ser observada na Figura 31.

Figura 31 – Sistema de Captura de Campo funcionando



Fonte: Autor.

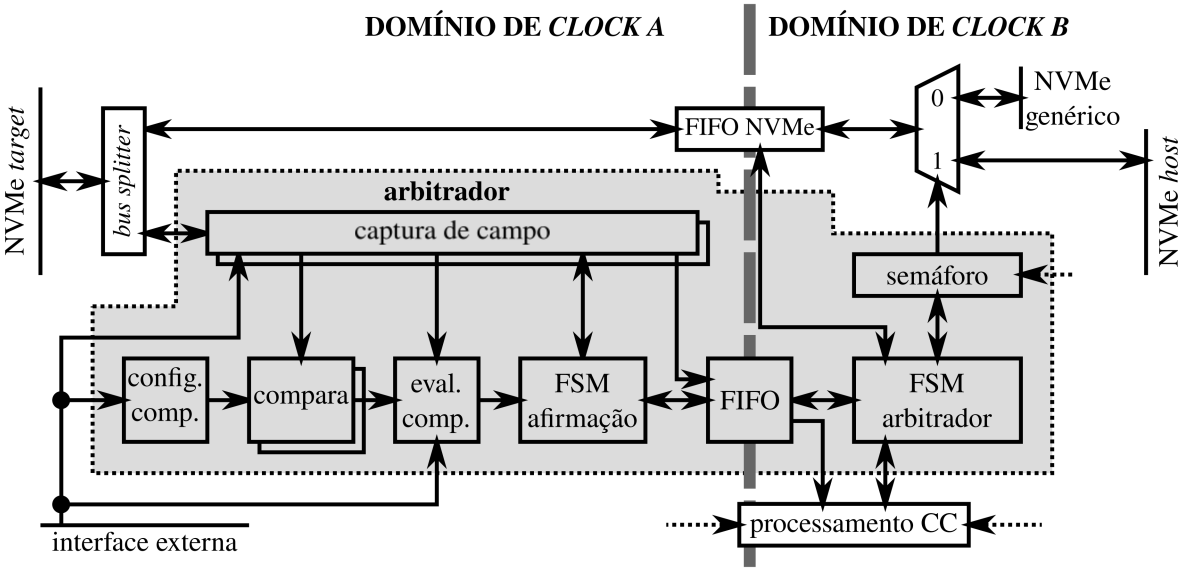
[1] Neste caso, o Módulo de Captura de Campo é empregado

Similarmente à simulação do sistema de captura de campo apresentado, a Figura 31 possui dois Módulos Contadores de *Opcode* em paralelo, processando o mesmo pacote. Neste sistema, o sinal *config* implica em diferentes configurações para os valores, sendo: 0x01, contagem de comandos de escrita; 0x02, comandos de leitura; 0x03, comandos proprietários[2]; e 0x05 contagem dos comandos de escrita e leitura, cujo campo SLBA está entre 0x07060504030200FE e 0x0706050403020102[3]. É possível observar que a contagem corresponde ao arquivo de teste, sendo apresentada no término do recebimento do pacote. A configuração de valor 0x05 contabiliza somente os arquivos de teste dentro da faixa esperada, conforme o Quadro 11. Concluída esta visão geral do sistema de contagem, passa-se ao sistema de *switch* NVMe.

5.3 SISTEMA DE *SWITCH* NVME

O sistema de *switch* NVMe possui a atribuição de selecionar algum aspecto do pacote e arbitrar entre dois destinatários: um NVMe genérico ou para a SSD conectada, através do módulo NVME *host*. Por NVMe genérico, entende-se qualquer dispositivo, físico ou virtual, que possa se comunicar no formato de pacote NVMe. Este dispositivo genérico pode desempenhar um número de funções, ao exemplo de uma memória *cache* para campos do FS que são acessados em maior frequência. Contudo, o sistema apresentado intencionalmente não limita a função do NVMe genérico — por consequência, este não será tratado, fazendo parte de trabalhos futuros. A Figura 32 apresenta o sistema, com o módulo responsável pela arbitração expandido.

Figura 32 – Simplificação do sistema de *Switch* NVMe, com Módulo Arbitrador expandido



Fonte: Autor.

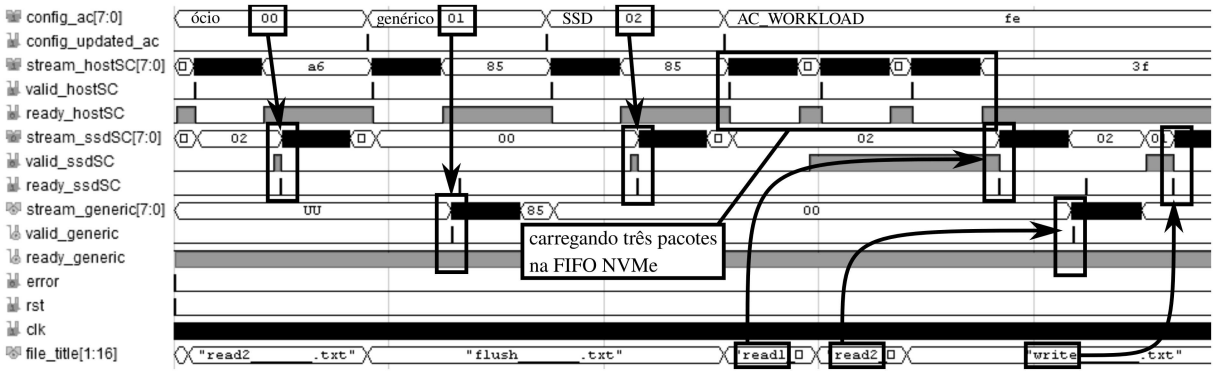
A Figura 32 apresenta o Módulo Arbitrador expandido, sendo este o conjunto de módulos

[2] Comando proprietário é todo o comando não definido na especificação NVM Express Incorporated (2019)

[3] Faixa correspondente aos arquivos de teste *read2* e *write*, porém descarta o *read1*.

que arbitram sobre o destino do pacote. O resultado do Módulo Arbitrador é um único *bit* que controla o demultilplexador específico para pacotes NVMe. Visto que este sistema é por definição obrigado a receber todo o pacote para arbitrar o destino, é feita a separação do domínio de *clock*. Consequentemente, uma FIFO feita especificamente para pacotes NVMe é posta para reter os pacotes e não prejudicar o fluxo normal do NVMe. Esta FIFO NVMe possui uma terceira porta de controle, cujo intuito é controlar o descargamento dos pacotes. O Módulo Arbitrador faz a configuração do demultiplexador e ativa a descarga da FIFO NVMe. Observa-se também, a necessidade de um bifurcador de canal (*bus splitter*), possibilitando o recebimento de um pacote NVMe para mais que um destinatário simultaneamente. Propõe-se então um cenário de direcionamento inteligente de um pacote. A Interface Externa configura simultaneamente o Módulo de Captura de Campo e os Módulos de Comparação[4]. Um novo pacote é recebido pelo NVMe *target*, sendo paralelamente armazenado na FIFO NVMe e analisado pelo Módulo de Captura de Campo. Este módulo, ao finalizar a analise, emite um sinal de fim do recebimento do pacote. Uma vez que o pacote tenha sido finalizado, o Módulo de Avaliação de Comparação combina logicamente as saídas dos Módulos de Comparação. Esta saída então é sincronizada por uma Máquina de Estados Finita (*Finite State Machine*, FSM), sendo introduzida na FIFO interna ao arbitrador, completando o desacoplamento dos domínios de *clock*. No domínio de *clock* B (à direita), outra máquina de estados recupera o valor, configura o demultiplexador em acordo com este valor; e então descarrega a FIFO NVMe. Também foi elaborado um Módulo de Semáforo, que gerencia possíveis conflitos entre o NVMe genérico e o arbitrador. Outrossim, o identificador do comando (ID) é enviado ao processamento de CC, como auxílio na implementação do NVMe genérico. A Figura 33 apresenta o funcionamento nominal do sistema, com diferentes configurações e arquivos de teste[5].

Figura 33 – Sistema de *Switch* NVMe em operação nominal



Fonte: Autor.

A Figura 33 apresenta três configurações distintas: 0x00, em ócio; 0x01, fixado no NVMe genérico; 0x02, fixado ao NVMe *host*; e 0xFE, configuração para teste, também chamada de

[4] Dependendo da configuração, mais que uma comparação deverá ser efetuada.

[5] Código disponível em https://gitlab.com/storagesystemslaboratory/nvme-switch

AC_WORKLOAD. Diferentemente dos módulos previamente analisados, o *switch* deve processar em seu ócio, fixando-se a um destino padrão. O destino padrão pode ser tanto o NVMe *host* quanto o NVMe genérico — fato que o distingue das configurações fixadas em NVMe genérico e NVMe *host*. No entanto, este sistema de *switch* mostra sua inteligência na configuração de teste 0xFE, pois define automaticamente o destino do pacote. Esta configuração foi elaborada para simular o caso real, em que o SLBA é monitorado para ser enviado ao NVMe genérico, caso corresponda à uma faixa específica. Esta faixa é definida entre 0x07060504030200FE e 0x0706050403020102 — idêntica ao Módulo Contador de *Opcode*. O resultado esperado pode ser observado no Quadro 11. As portas, cujo nome é terminado por "*_hostSC*" advém do NVMe *target*; e aqueles terminados por "*_ssdSC*" são destinadas ao NVMe *host*. O sinais internos terminados por "*_generic*" são destinado ao NVMe genérico. O resultado pode ser avaliado corretamente quando a porta *stream_hostSC* transfere um pacote e, após o fim desta transferência, este pacote é transferido ao destinatário conforme o nome do arquivo e a configuração.

Este capítulo desenvolveu, em um grau de detalhamento adequado, uma descrição dos resultados obtidos da parte da simulação lógica. Este porém, é somente parte do projeto. Houve uma pesquisa intensa nos recursos bibliográficos, com decisões a serem tomadas, como as decisões das arquiteturas, tanto dos módulos, como as referentes a todo o projeto. Foram apresentados os funcionamentos ideais dos principais módulos, optando por não dissertar sobre as exceções consideradas, nem outros módulos menores já implementados, considerados não relevantes a este texto. Também foram planejados os módulos de PCIe *target* e *host*, juntamente com seus respectivos módulos NVMe. Igualmente, foi planejado um módulo de NVMe genérico, conhecido como NVMe *cache*, que realizaria um armazenamento rápido dos comandos mais acessados. Contudo, estes módulos contam como trabalhos futuros, por não terem sido implementados completamente. Finalizado este capítulo de Análise de Resultados, passa-se à Conclusão, que tratá reflexões finais sobre o projeto e apresentará uma continuidade ao mesmo.

6 CONCLUSÃO

O mais recente protocolo de interfaceamento para SSDs acarretou em uma grande melhoria para o dispositivo. A criação da especificação NVMe não somente desbloqueou parcialmente a banda intrínseca da memória *flash* como também abstraiu controle do *host*, deixando-o menos capaz de controlar da SSD em certas funcionalidades. Por este motivo, o protocolo é aperfeiçoado a cada nova versão, integrando mais controles e mais funcionalidades. Contudo, este aperfeiçoamento é comumente liderado pela indústria detentora da tecnologia, o que dificulta ou até mesmo impossibilita o acesso livre ao pesquisador acadêmico. Assim sendo, a especificação NVMe é aberta ao público, porém sua implementação na SSD é dependente do fabricante. Para o estudo de qualquer tecnologia de memória, é necessária uma plataforma versátil para a análise e implementação de novas propostas. Até o momento, nenhuma plataforma versátil e aberta (*open source*) visando SSDs e NVMe foi elaborada. Com isto, o presente trabalho dedica-se a suprir academia e indústria com uma ferramenta inédita para o estudo do protocolo NVMe, utilizando SSDs comerciais, que também possibilite a otimização de acesso à memória.

Respeitando o objetivo de habilitar a otimização de acesso, faz-se necessário o máximo de banda possível. Isto implica no processamento de algoritmos em lógica, em contraste aos seus equivalentes em processadores de arquitetura fixa, de menor banda para a mesma frequência. Do mesmo modo, a lógica programável é flexível, pois permite a reconfiguração, sendo utilizada em diversas áreas na indústria e pesquisa. Por este motivo, faz-se então a combinação de lógica programável com tecnologia *flash*, especificamente para o estudo de SSDs e do protocolo NVMe. Logo, este projeto utiliza a placa de avaliação KCU105 por conta de sua replicabilidade e independência tecnológica. Outros *hardwares* prontos também são utilizados, como a placa adaptadora FPGA Drive e SSD comercial. Deste modo, outros pesquisadores poderão adquirir os mesmos materiais utilizados, dos mesmos fabricantes e implementar este projeto.

Tendo em vista a totalidade do projeto, os objetivos propostos foram completamente atendidos e os resultados obtidos estão em acordo com as métricas previamente estabelecidas na seção de Método de Avaliação. A documentação descritiva sobre cada módulo foi gerada, incluindo diagramas. Também foram gerados arquivos de teste formadores da base de dados, sendo utilizados conforme o capítulo de Análise de Resultados. Por meio destes arquivos, foi comprovado o bom funcionamento de cada módulo, em acordo com as premissas de codificação proposta na seção de Método de Desenvolvimento. Dessa maneira, é considerado que os objetivos propostos foram completamente atingidos.

Como forma de validação deste projeto, uma publicação de artigo, apresentada no Apêndice H, foi realizada, cujo título é "*Enabling Fast Decision Criteria in NVMe SSDs through*

NVMe-based Switch". O artigo tratou da proposta de habilitação de otimização da memória, marcando a primeira proposta de NVMe *switch* na academia. Este artigo foi publicado no Jornal de Engenharia Elétrica e Eletrônica (*Journal of Electrical and Electronics Engineering*), pertencente à Organização Internacional de Pesquisa Científica (*International Organization of Scientific Research, IOSR-JEEE*). O código fonte desenvolvido para este artigo se encontra em repositório *online*[1].

Trabalhos futuros poderão se dedicar à implementação total em *hardware*, finalizando os módulos de PCIe e NVMe. Outrossim, poderá ser tratada a implementação de um conceito de NVMe genérico, conforme apresentado no capítulo de Análise de Resultados. Como sugestão de NVMe genérico, propor-se-ia a execução de uma região de acesso rápido, similar à memória RAM (*cache*). Esta região cobriria algum campo de interesse no Sistema de Arquivo (FS), acelerando seu acesso e estendendo o tempo de vida da SSD. Com o projeto proposto funcional, sugere-se como trabalho futuro a análise de padrões de uso do FS, trazendo melhorias em codificação dos mesmos, considerando esta nova tecnologia *flash*. Estas sugestões poderão ser incorporadas para trabalhos de doutorandos que estudem FS, o protocolo NVMe, memorias *flash*.

[1] https://gitlab.com/storagesystemslaboratory/nvme-switch

REFERÊNCIAS BIBLIOGRÁFICAS

Ajdari, M. et al. A scalable hw-based inline deduplication for ssd arrays. *IEEE Computer Architecture Letters*, v. 17, n. 1, p. 47–50, Jan 2018. ISSN 1556-6056. Citado na página 45.

AQUA COMPUTER. *kryoM.2*. 2019. Disponível em: <https://shop.aquacomputer.de/product_info.php?language=en&products_id=3400>. Acesso em: 08 mar 2019. Citado na página 20.

BAIDU. 2019. Disponível em: <https://www.baidu.com/>. Acesso em: 28 fev 2019. Citado na página 53.

BHIMANI, J. et al. Fios: Feature based i/o stream identification for improving endurance of multi-stream ssds. In: . [S.l.: s.n.], 2018. Citado 3 vezes nas páginas 37, 46 e 56.

BREWER, J.; GILL, M. *Nonvolatile Memory Technologies with Emphasis on Flash: A Comprehensive Guide to Understanding and Using NVM Devices*. [S.l.: s.n.], 2007. 1-758 p. Citado 3 vezes nas páginas 28, 29 e 34.

Cai, Y. et al. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, v. 105, n. 9, p. 1666–1704, Sep. 2017. ISSN 0018-9219. Citado na página 29.

Cai, Y. et al. Vulnerabilities in mlc nand flash memory programming: Experimental analysis, exploits, and mitigation techniques. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. [S.l.: s.n.], 2017. p. 49–60. ISSN 2378-203X. Citado 5 vezes nas páginas 34, 35, 44, 56 e 57.

Cai, Y. et al. Fpga-based solid-state drive prototyping platform. In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. [S.l.: s.n.], 2011. p. 101–104. Citado 5 vezes nas páginas 43, 44, 50, 56 e 57.

Cai, Y. et al. Read disturb errors in mlc nand flash memory: Characterization, mitigation, and recovery. In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. [S.l.: s.n.], 2015. p. 438–449. ISSN 1530-0889. Citado na página 35.

Cai, Y. et al. Data retention in mlc nand flash memory: Characterization, optimization, and recovery. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. [S.l.: s.n.], 2015. p. 551–563. ISSN 1530-0897. Citado 4 vezes nas páginas 35, 43, 56 e 57.

Cohen, B. *VHDL Coding Styles and Methodologies*. 2. ed. [S.l.]: Springer, 1999. 455 p. Citado na página 59.

COUGHLIN, M.; ISMAIL, A.; KELLER, E. Apps with hardware: Enabling run-time architectural customization in smart phones. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016. p. 621–634. ISBN 978-1-931971-30-0. Disponível em: <https://www.usenix.org/conference/atc16/technical-sessions/presentation/coughlin>. Citado 2 vezes nas páginas 54 e 57.

DEPARTMENT OF DEFENSE. *Standard General Requirements for Electronic Equipment*. Washington DC, EUA, 1992. Acesso em: 27 abr 2019. Citado na página 42.

DICIONÁRIO PRIBERAM. *Aplicativo*. 2019. Disponível em: <https://dicionario.priberam.org/aplicativo>. Acesso em: 22 mar 2019. Citado na página 15.

DICIONÁRIO PRIBERAM. *Metadado*. 2019. Disponível em: <https://dicionario.priberam.org/metadado>. Acesso em: 07 jun 2019. Citado na página 15.

EXT4 WIKI. *Ext4 (and Ext2/Ext3) Wiki*. 2019. Disponível em: <https://ext4.wiki.kernel.org/>. Acesso em: 11 mar 2019. Citado na página 23.

FLASHDBA. *Understanding Flash: The Flash Translation Layer*. 2019. Disponível em: <https://flashdba.com/2014/09/17/understanding-flash-the-flash-translation-layer/>. Acesso em: 11 mar 2019. Citado na página 23.

GITHUB. *Repositório oficial para Ext4 versão Linux*. 2019. Disponível em: <https://github.com/torvalds/linux/tree/master/fs/ext4>. Acesso em: 20 mar 2019. Citado na página 23.

IBM DEVELOPPER. *Network file systems and Linux*: Nfs: As useful as ever and still evolving. 2019. Disponível em: <https://www.ibm.com/developerworks/library/l-flash-filesystems/>. Acesso em: 09 mar 2019. Citado na página 21.

INTELLIPROP. 2019. Disponível em: <http://www.intelliprop.com/>. Acesso em: 19 mar 2019. Citado na página 60.

INTELLIPROP. *AFCI IP Core with BCH NAND Flash Controller*. 2019. Xilinx. Disponível em: <https://www.xilinx.com/products/intellectual-property/1-fod2wk.html>. Acesso em: 02 mai 2019. Citado na página 88.

INTELLIPROP. *NVMe Target Core*. 2019. Xilinx. Disponível em: <https://www.xilinx.com/products/intellectual-property/1-6eqslp.html>. Acesso em: 02 mai 2019. Citado na página 88.

INTELLIPROP. *NVMe to NVMe Bridge*. 2019. Xilinx. Disponível em: <https://www.xilinx.com/products/intellectual-property/1-mpymke.html>. Acesso em: 02 mai 2019. Citado 3 vezes nas páginas 65, 87 e 88.

INTERNATIONAL COMMITTEE FOR INFORMATION TECHNOLOGY STANDARDS. *Página principal*. 2019. Disponível em: <http://www.incits.org/>. Acesso em: 03 mar 2019. Citado na página 18.

Jun, S.; AND, S. X. Terabyte sort on fpga-accelerated flash storage. In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. [S.l.: s.n.], 2017. p. 17–24. Citado 3 vezes nas páginas 50, 52 e 57.

JUN, S.-W. et al. Scalable multi-access flash store for big data analytics. In: *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. New York, NY, USA: ACM, 2014. (FPGA '14), p. 55–64. ISBN 978-1-4503-2671-1. Disponível em: <http://doi.acm.org/10.1145/2554688.2554789>. Citado 3 vezes nas páginas 53, 56 e 57.

JUN, S.-W. et al. Grafboost: Using accelerated flash storage for external graph analytics. In: . [S.l.: s.n.], 2018. p. 411–424. Citado 2 vezes nas páginas 51 e 57.

KANG, J.-U. et al. The multi-streamed solid-state drive. In: *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*. Philadelphia, PA: USENIX Association, 2014. Disponível em: <https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/kang>. Citado na página 36.

KANG, W.-H. et al. X-ftl: Transactional ftl for sqlite databases. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2013. (SIGMOD '13), p. 97–108. ISBN 978-1-4503-2037-5. Disponível em: <http://doi.acm.org/10.1145/2463676.2465326>. Citado 6 vezes nas páginas 23, 46, 47, 48, 49 e 57.

KEYSIGHT TECHNOLOGIES. *NE5412E Serial Attached SCSI*: 4 (sas-4) transmitter test application. 2019. Disponível em: <https://www.keysight.com/en/pd-2853599-pn-N5412E/serial-attached-scsi-4-sas-4-transmitter-test-application?cc=BR&lc=por>. Acesso em: 29 mar 2019. Citado na página 18.

KOMSUL, M.; MCEWAN, A.; MIR, I. An fpga-based development platform for real-time solid state devices. In: . [S.l.: s.n.], 2014. v. 2, p. 1198–1203. Citado 3 vezes nas páginas 45, 56 e 57.

Lee, J.-D.; Hur, S.-H.; Choi, J.-D. Effects of floating-gate interference on nand flash memory cell operation. *IEEE Electron Device Letters*, v. 23, n. 5, p. 264–266, May 2002. ISSN 0741-3106. Citado na página 35.

LI, D. et al. Cisc: Coordinating intelligent ssd and cpu to speedup graph processing. In: . [S.l.: s.n.], 2018. p. 149–156. Citado 2 vezes nas páginas 52 e 57.

MASUOKA, F.; IIZUKA, H. *Semiconductor memory device and method for manufacturing the same*. Google Patents, 1980. US Patent 4,531,203. Disponível em: <https://patents.google.com/patent/US4531203A/>. Citado na página 27.

MCMILLAN, R. *Microsoft supercharges bing search with programmable chips*. 2014. Disponível em: <https://www.wired.com/2014/06/microsoft-fpga/>. Acesso em: 08 abr 2019. Citado na página 14.

Meliolla, G. et al. Study of tunneling gate oxide and floating gate thickness variation effects to the performance of split gate flash memory. In: *2017 International Symposium on Electronics and Smart Devices (ISESD)*. [S.l.: s.n.], 2017. p. 256–259. Citado na página 27.

MIN, C. et al. Lightweight application-level crash consistency on transactional flash storage. In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, 2015. p. 221–234. ISBN 978-1-931971-225. Disponível em: <https://www.usenix.org/conference/atc15/technical-session/presentation/min>. Citado 2 vezes nas páginas 47 e 57.

Mohan, V. et al. Modeling power consumption of nand flash memories using flashpower. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 32, n. 7, p. 1031–1044, July 2013. ISSN 0278-0070. Citado na página 33.

Nam, E. H. et al. Ozone (o3): An out-of-order flash memory controller architecture. *IEEE Transactions on Computers*, v. 60, n. 5, p. 653–666, May 2011. ISSN 0018-9340. Citado 4 vezes nas páginas 45, 48, 56 e 57.

Novotný, R.; Kadlec, J.; Kuchta, R. Overview of the nand flash high-speed interfacing and controller architecture. In: *Journal of Information Technology & Software Engineering*. [S.l.: s.n.], 2015. v. 5, n. 1. Citado 3 vezes nas páginas 25, 30 e 31.

NVM EXPRESS. 2019. Disponível em: <https://nvmexpress.org/>. Acesso em: 14 mar 2019. Citado 3 vezes nas páginas 39, 40 e 41.

NVM EXPRESS INCORPORATED. *NVM Express Base Specification*. [S.l.], 2019. Acesso em: 27 abr 2019. Citado 5 vezes nas páginas 40, 63, 74, 76 e 87.

OPSERO ELECTRONIC DESIGN. *FPGA Drive FMC*. 2019. Disponível em: <https://opsero.com/product/fpga-drive-fmc-dual/>. Acesso em: 16 abr 2019. Citado na página 62.

OUYANG, J. et al. Sdf: Software-defined flash for web-scale internet storage systems. In: . [S.l.: s.n.], 2014. v. 49, p. 471–484. Citado 3 vezes nas páginas 38, 53 e 57.

PERIPHERAL COMPONENT INTERCONNECT SPECIAL INTEREST GROUP. *Doubling Bandwidth in Under Two Years*: Pci express base specification revision 5.0, version 0.9 is now available to members. 2019. Disponível em: <https://pcisig.com/doubling-bandwidth-under-two-years-pci-express%C2%AE-base-specification-revision-50-version-09-now>. Acesso em: 06 mar 2019. Citado na página 19.

PERIPHERAL COMPONENT INTERCONNECT SPECIAL INTEREST GROUP. *Página principal*. 2019. Disponível em: <https://pcisig.com/>. Acesso em: 06 mar 2019. Citado na página 19.

Rouhani, B. D. et al. Ssketch: An automated framework for streaming sketch-based analysis of big data on fpga. In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. [S.l.: s.n.], 2015. p. 187–194. Citado 2 vezes nas páginas 52 e 57.

SAMSUNG. *Data Center SSD*. 2019. Disponível em: <https://www.samsung.com/semiconductor/minisite/ssd/product/data-center/overview/>. Acesso em: 01 mai 2019. Citado na página 38.

SAMTEC. *VITA 57.1 FMC Standard Products and Support*. 2019. Disponível em: <https://www.samtec.com/standards/vita/fmc>. Acesso em: 29 abr 2019. Citado na página 63.

SCHROEDER, B.; LAGISETTY, R.; MERCHANT, A. Flash reliability in production: The expected and the unexpected. In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, 2016. p. 67–80. ISBN 978-1-931971-28-7. Disponível em: <https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder>. Citado na página 38.

SERIAL ATA. *New SATA spec will double data transfer speeds to 6Gbps*: Sata-io will demonstrate the speed and functionality of the third-generation specification at the intel developer forum. 2019. Wayback Machine. Disponível em: <https://web.archive.org/web/20100923003722/http://www.sata-io.org/documents/SATA_6gbphy_pressrls_finalrv2.pdf>. Acesso em: 29 mar 2019. Citado na página 18.

SERIAL ATA. *Página principal*. 2019. Disponível em: <https://sata-io.org/>. Acesso em: 06 mar 2019. Citado na página 18.

SHARAFEDDIN, M. et al. On the efficiency of automatically generated accelerators for reconfigurable active ssds. In: . [S.l.: s.n.], 2014. v. 2015. Citado 2 vezes nas páginas 42 e 60.

SILICON POWER BLOG. *NAND Flash Memory Technology: The Basics of a Flash Memory Cell*. 2019. Disponível em: <https://blog.silicon-power.com/index.php/guides/nand-flash-memory-technology-basics/>. Acesso em: 12 mar 2019. Citado na página 32.

SOLOMON, R. *PCI Express Basics & Background*. [S.l.]: PCI-SIG, 2019. Disponível em: https://pcisig.com/sites/default/files/files/PCI_Express_Basics_Background.pdf Acessado em 08/03/2019. Citado na página 19.

SONG, Y. H. et al. Cosmos openssd: A pcie-based open source ssd platform. *Proc. Flash Memory Summit*, 2014. Citado 2 vezes nas páginas 49 e 57.

SORT BENCHMARK PÁGINA PRINCIPAL. 2019. Disponível em: <https://sortbenchmark.org/>. Acesso em: 20 fev 2019. Citado na página 51.

STMICROELECTRONICS. *UM2407 User Manual*: Stm32h7 nucleo-144 boards (mb1364). 2019. Disponível em: <https://www.st.com/content/ccc/resource/technical/document/user_manual/group1/95/1a/9a/89/87/6a/45/70/DM00499160/files/DM00499160.pdf/jcr:content/translations/en.DM00499160.pdf>. Acesso em: 23 mai 2019. Citado na página 67.

STORAGE REVIEW. 2019. Disponível em: <https://www.storagereview.com/>. Acesso em: 12 mar 2019. Citado 2 vezes nas páginas 35 e 37.

STORAGE REVIEW. *SSD vs HDD*. 2019. Disponível em: <https://www.storagereview.com/ssd_vs_hdd>. Acesso em: 12 mar 2019. Citado na página 38.

STRATIKOPOULOS, A. et al. Fastpath: Towards wire-speed nvme ssds. 07 2018. Citado 2 vezes nas páginas 55 e 57.

XILINX. *MicroBlaze Processor Reference Guide*. 2019. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug984-vivado-microblaze-ref.pdf>. Acesso em: 28 abr 2019. Citado 2 vezes nas páginas 87 e 88.

XILINX. *Resource Utilization for AXI Bridge for PCI Express Gen3 Subsystem v3.0*. 2019. Disponível em: <https://www.xilinx.com/support/documentation/ip_documentation/ru/axi-pcie3.html>. Acesso em: 02 mai 2019. Citado 2 vezes nas páginas 87 e 88.

XILINX. *Resource Utilization for DDR4 SDRAM (MIG) v2.2*. 2019. Disponível em: <https://www.xilinx.com/support/documentation/ip_documentation/ru/ddr4.html>. Acesso em: 02 mai 2019. Citado 2 vezes nas páginas 87 e 88.

XILINX. *UltraScale FPGA*: Product tables and product selection guide. 2019. Disponível em: <https://www.xilinx.com/support/documentation/selection-guides/ultrascale-fpga-product-selection-guide.pdf>. Acesso em: 02 mai 2019. Citado 2 vezes nas páginas 63 e 90.

XILINX. *Xilinx Kintex UltraScale FPGA KCU105 Evaluation Kit*. 2019. Disponível em: <https://www.xilinx.com/products/boards-and-kits/kcu105.html>. Acesso em: 16 abr 2019. Citado 2 vezes nas páginas 63 e 91.

YANG, C. et al. Reducing write amplification for inodes of journaling file system using persistent memory. In: . [S.l.: s.n.], 2019. p. 866–871. Citado 3 vezes nas páginas 48, 49 e 56.

Yoshimi, M.; Wu, C.; Yoshinaga, T. Accelerating blast computation on an fpga-enhanced pc cluster. In: *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. [S.l.: s.n.], 2016. p. 67–76. ISSN 2379-1896. Citado 2 vezes nas páginas 50 e 57.

ZEEIS. *Flash Translation Layer - FTL for NAND*. 2019. Disponível em: <http://www.zeeis.com/flash-translation-layer/>. Acesso em: 11 mar 2019. Citado na página 23.

Zhang, J. et al. Design and implementation of optical fiber ssd exploiting fpga accelerated nvme. *IEEE Access*, v. 7, p. 152944–152952, 2019. Citado 2 vezes nas páginas 55 e 57.

Zhang, M. et al. Fpga-based failure mode testing and analysis for mlc nand flash memory. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. [S.l.: s.n.], 2017. p. 434–439. ISSN 1558-1101. Citado 3 vezes nas páginas 44, 56 e 57.

APÊNDICE A – ESTIMATIVA PRELIMINAR PARA O PROJETO DE PESQUISA

Quadro 12 – Estimativa preliminar dos módulos para o projeto de pesquisa

	Quantia	***Slices****	***Look-Up Table****	**Células Lógicas***	**BRAM** (36 kb)	**BRAM** (kb)	**Freq. máxima** (MHz)	**Obs.**
NVMe *target*, NVMe *host*, *Switch* inteligente	1	5374	15114	15114	60	2160	250	1
PCIe Gen3	2	1204	4819	4819	19	684	250	2
Controlador DRAM	1	18755	22795	24509	25	900	250	3
Interface externa	1	757	989	989	12	432	350	4
Soma	1	27294	48536	50250	135	4860	350	
Subtotal	60 %	32753	58243	60300	162	5832	350	Margem de 20 %
Total	100 %	54588	97072	100500	270	9720	350	

* Termo válido somente para comparação de FPGAs da mesma família.
1 (INTELLIPROP, 2019d), 2 (XILINX, 2019b), 3 (XILINX, 2019c), 4 (XILINX, 2019a).
Fonte: Autor.

O Quadro 13 apresenta a estimativa de uso da DRAM, considerando somente filas e os comandos da especificação base NVMe™.

Quadro 13 – Estimativa preliminar do uso da DRAM para o projeto de pesquisa

	Quantia	**Máximo**[1]
Filas	2000	65535
Comandos	2000	64000
Largura dos comandos (b)	512	
Subtotal (b)	2048000000	
Subtotal (Gb)	2,048	
Subtotal (GB)	0,256	
Margem[2] **(%)**	30	
Total (GB)	0,333	

1 (NVM EXPRESS INCORPORATED, 2019), 2 O controlador da DRAM pode possuir *bits* de controle.
Fonte: Autor.

APÊNDICE B – ESTIMATIVA PRELIMINAR PARA O PROJETO KRATUS

O projeto Kratus é um projeto paralelo ao de pesquisa, que visa um objetivo similar com maior complexidade. Este utiliza uma placa de avaliação FPGA. O projeto proposto não utilizará certos módulos presentes no projeto da placa Kratus, porém estes devem ser levados em consideração. A placa de avaliação é custosa e ela deverá servir tanto para o projeto de pesquisa, como para o projeto Kratus. No Quadro 14 abaixo, é feito uma listagem da estimativa de ocupação do FPGA por módulos. Observa-se que alguns módulos ficarão em "paralelo", significando que não serão implementados ao mesmo tempo, mas sim em configurações diferentes. A explicação desta decisão de projeto foge do tema desta pesquisa.

Quadro 14 – Estimativa preliminar dos módulos para o projeto Kratus

	Quantia	***Slices****	***Look-Up Table****	**Células Lógicas***	**BRAM** (36 kb)	**BRAM** (kb)	**Freq. máxima** (MHz)	**Obs.**
NVMe *bridge*	1	5374	15114	15114	60	2160	100	1
NVMe alvo	1	1316	5268	5268	22	792	100	2. Paralelo
PCIe Gen3	3	1204	4819	4819	19	684	250	3
Control. ONFI	1	15567	18920	18920	43	1548	200	4. Paralelo
Control. JEDEC	1	15567	18920	18920	43	1548	200	Fonte inexistente, assumindo ocupação da ONFI. Paralelo
Interface DDR4	1	18755	22795	24509	25	900	250	6
Micro-Blaze	1	757	989	989	12	432	350	7
Soma	1	44065	72275	73989	197	7092	350	Pior caso
Subtotal	60 %	52878	86730	88787	236	8510	350	Margem de 20 %
Total	100 %	88129	144550	147978	394	14184	350	

* Termo válido somente para comparação de FPGAs da mesma família.
1 (INTELLIPROP, 2019d), 2 (INTELLIPROP, 2019c), 3 (XILINX, 2019b), 4 (INTELLIPROP, 2019b), 5 (XILINX, 2019c), 6 (XILINX, 2019a).
Fonte: Autor.

APÊNDICE C – LISTA DE REQUISITOS DO SISTEMA PROPOSTO

O sistema proposto possui como requisito sua implementação em uma placa de avaliação FPGA, cuja as especificações mínimas são:

- Fabricada pela Xilinx®;
- Contenha um FPGA, cuja a especificação mínima:
  - Atenda aos requisitos do projeto proposto e do projeto Kratus constados nos Apêndice B e Apêndice A respectivamente;
  - Possua primitivas para três PCIe *hard*;
  - Capaz de gerenciar frequências na ordem de 350 MHz, conforme o Quadro 12;
  - Ser compatível com o microcontrolador *soft MicroBlaze*.
- Contenha memória DRAM suficiente, conforme a estimativa da Quadro 13;
- Seja compatível com FPGA Drive, com ambos os SSDs disponíveis ao acesso;
- Possua conectores extras para placas de expansão;
- Possua comunicação UART.

APÊNDICE D – RECURSOS DO FPGA XCKU040-2FFVA1156E

Quadro 15 – Recursos do FPGA XCKU040-2FFVA1156E

	FPGA	**XCKU040-2FFVA1156E**
Recursos Lógicos	Células lógicas	530000
	Registradores	484800
	Look-Up Tables	242400
Recursos de Memória	Máxima quantidade de RAM distribuída (kb)	7050
	BRAM/FIFO com ECC (36 kb)	600
	BRAM/FIFO (18 kb)	1200
	Total de BRAM (MB)	21,1
	Gerenciamento de *clock* (1 MMCM, 2 PLLs)	10
	IO DLL	40
IP integrados	*Slices* DSP	1920
	System Monitor	1
	PCIe Gen3	3
	Transceivers GTH de 16,3 Gbps	20
Recursos de IO relacionados ao *Footprint*	Quantidade máxima de *transceivers* HR *sigle-ended*	104
	Quantidade máxima de pares de *transceivers* HP diferenciais	192
	Quantidade máxima de *transceivers* HP *sigle-ended*	416
	Quantidade máxima de pares de *transceivers* HR diferenciais	48
	GTH/GTY	20

Fonte: Adaptado de Xilinx (2019d).

APÊNDICE E – *HARDWARE* DA KCU105

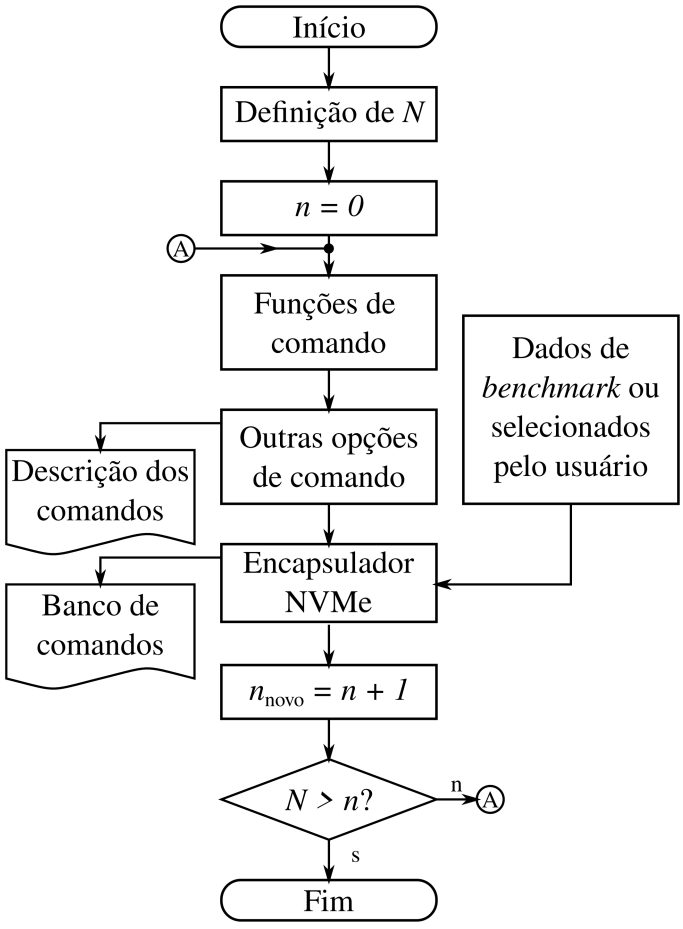
Quadro 16 – Recursos da KCU105

Grupo	Interface
Configuração	JTAG para configuração e programação
Memória	64 MB *Flash* SPI para programação do FPGA
	2 GB DRAM DDR4
	8 kb IIC EEPROM
	Micro SD para programação do FPGA por cartão externo
Comunicação	Ethernet Gigabit
	Dois soquetes SFP/SFP+
	Portas GTX com quatro conectores SMA
	UART
	PCIe x8 *edge connector*
Conectores de expansão	FMC HPC parcialmente populado
	FMC LPC
	Dois PMOD *headers*
	IIC
Visual	Saída HDMI
	Oito LEDs
Clock	Oito *clocks* programáveis
	Clock do sistema (*system clock*)
	EMC *clock*
	Clock do usuário (*user clock*)
	Clocks de *jitter* atenuados
IO	Cinco botões monoestáveis
	Quatro interruptores DIP
	Encoder bidirecional com botão monoestável integrado
	Par diferencial por conector SMA
Alimentação	Fonte 12V
	Conector ATX

Fonte: Adaptado de (XILINX, 2019e).

APÊNDICE F – MÉTODO DE CRIAÇÃO DA BASE DE COMANDOS NVME

Figura 34 – Método de criação de comandos NVMe pseudo-aleatórios



Fonte: Autor.

APÊNDICE G – ESPECIFICAÇÃO DO MÓDULO DE CAPTURA DE CAMPO

Este apêndice especifica o Módulo de Captura de Campo, seu funcionamento e operação.

Descrição

O módulo deverá ser capaz de capturar campos em comandos NVMe de submissão ou de compleição, separando as partes dos campos configurados em um registrador interno. Este módulo receberá o comando em formato de *byte*, em sincronia com uma indicação por borda de uma porta que aponte o começo do recebimento. A configuração de qual campo capturar destes será adquirida através de suas portas. Esta configuração deverá seguir uma convenção estipulada, possuindo uma configuração de ócio. O comando capturado deverá ficar temporariamente armazenado neste registrador interno, até que um módulo externo gere um sinal para expor o conteúdo. Um sinal indicando conteúdo válido no registrador deverá ser externado. Caso um novo comando tenha sido capturado antes do precedente ter sido armazenado, o precedente será perdido e um sinal externado indicará esta perda, para tomada de conhecimento do projetista. Esta funcionalidade deverá ser implementada em paralelo, em um número pré-determinado de vezes, a fim de que se possa capturar múltiplos campos de uma só vez.

Configuração

Existirão configurações para:

- A borda ativa;
- Quantidade de capturas de campo;
- A externalização do campo por borda ativa ou ambas as bordas;
- Desabilitação do mecanismo de envio.

APÊNDICE H – ARTIGO PUBLICADO

Este apêndice apresenta o artigo realizado durante a pesquisa de mestrado, cujo título é *Enabling Fast Decision Criteria in NVMe SSDs through NVMe-based Switch*, publicado na *International Organization of Scientific Research Journals of Electrical and Electronics Engineering* (IOSR-JEEE). O artigo foi publicado em Março de 2020, data que o local de publicação obteve um fator de impacto de 3,26. O artigo contempla de forma resumida e introdutória, a parte de arbitragem do pesquisa de mestrado.

IOSR Journal of Electrical and Electronics Engineering (IOSR-JEEE)
e-ISSN: 2278-1676,p-ISSN: 2320-3331, Volume 15, Issue 2 Ser. I (Mar – Apr 2020), PP 54-58
www.iosrjournals.org

Enabling Fast Decision Criteria in NVMe SSDs through NVMe-based Switch

Vinícius G. Linden[1], Rodrigo M. Figueiredo[1], Cassiano S. Campes[1,2], Lúcio R. Prade[1]
[1](Polytechnic School, Unisinos University, Brazil)
[2](College of Information and Communication Engineering, Sungkyunkwan University, South Korea)
Corresponding Author: Vinícius G. Linden

Abstract:
Background*: Solid-state drives (SSDs) have increased throughput and bandwidth drastically with the introduction of Non-Volatile Memory Express (NVMe) protocol, profiting from the state-of-the-art Peripheral Component Interconnect Express (PCIe) interface. Moreover, SSDs are perceived by the host as a black-box where they simply service the write and read requests. This inhibits the host to acknowledge the device's internal architecture. Additionally, the SSD does not distinguish between data and metadata requests. Thus, the host still lacks opportunities to get the best performance from SSDs due to the data semantic gap.*
Materials and Methods*: This paper proposes a hardware solution to enable fast decision criteria in NVMe SSDs. Our proposed model is flexible enough to be configured for specific workloads by the host. The solution enables switching decisions to be transferred into the hardware, minimizing CPU usage.*
Conclusion:*The data semantic gap is shortened in the hardware level, consequently reducing the write requests to flash and widening the life-span of the device.*
Key Word*: NVMe; Switch; SSD; HDL.*

Date of Submission: 12-03-2020 Date of Acceptance: 27-03-2020

I. Introduction

The Non-Volatile Memory Express (NVMe) specification has allowed higher bandwidth through the use of Peripheral Component Express (PCIe). It creates an interface for the host to communicate with the Solid-State Drive (SSD). Like any other storage interface, it takes addresses and data to either write or read, making the device to be perceived as a black-box. As an attempt to reduce the data semantic gap in NVMe version 1.3, the multi-stream concept has been introduced[1], enabling the separation of data inside the SSD. Nevertheless, stream separation is not enough to transfer the data semantic to the storage, leaving room for improvements. These enhancements would greatly benefit systems that are characterized by a massive use of SSDs.

State-of-the-art data-centers are increasingly employing SSDs for storage [2, 3]. They require a reliable File System (FS) that is able to handle a massive storage space, such as the *ext4*. This FS is prevalent in data-centers [4] and Linux systems in general. It separates the storage space into multiple block groups segments, reserving the first 1024 bytes for the bootloader. Each block group is further divided in superblock, group description table, block bitmap, inode bitmap, and data blocks. These regions are updated according to specific workloads and, consequently, metadata may have a much higher update frequency than data blocks. Because SSDs do not allow in-place-updates, a Flash Translation Layer (FTL) is employed to make the logical to physical addressing transparent to the host. However, excessive physical writes into the storage shorten the life-span of the flash cells due to wearout [5, 6]. This feature is characteristic of flash technology, so an SSD array system will still lose flash cells due to wearout caused by repetitive writes.

A system that has an array of NVMe-based SSDs is always routed through a PCIe switch, but since PCIe packets are not the same as NVMe commands, NVMe fields are not accessible to the PCIe switch in this manner. Thus, a PCIe switch does not allow command separation in NVMe. Therefore, a switch for NVMe commands is beneficial, because it allows multiple implementations to specific workloads [7]. The NVMe-based switch can be configured to make decisions for a given workload, in a given system. It is more advantageous to process data in- or near-storage with hardware for reduced latency and reduced CPU utilization in I/O processing [8, 9, 10]. To the best of our knowledge, there is no proposal for an NVMe switch solution that allows routing criteria based on a configurable aspect of the NVMe command itself. This paper proposes a novel packet-based NVMe switch architecture, capable of separating different NVMe commands based on the content of one or more of NVMe fields. This can be accomplished in Hardware Description Language (HDL), enabling the implementation on a Field-Programmable Gate Array (FPGA) [11]. The main contributions are: a

shortening of the data semantic gap between host and SSD by allowing in-hardware NVMe-based command switching; a way to generate designer-defined NVMe packets out of NVMe commands; a basic platform for NVMe system manipulation; and the NVMe switch flexible architecture.

II. Proposed System

A top-level architecture view is presented in **Figure 1**. There, the switch logic is between the PCIe and NVMe interfaces for communication with a host and the SSD. The NVMe command can be sent either to the SSD or to a *generic NVMe* device. The last being the main reason for this architecture. As discussed in the Introduction, the designer may choose the *generic NVMe* to implement as any possibility to enable near-storage processing. The switch's configuration is asynchronously introduced via an external interface. This configuration port is used to adapt the switch to specific workloads.

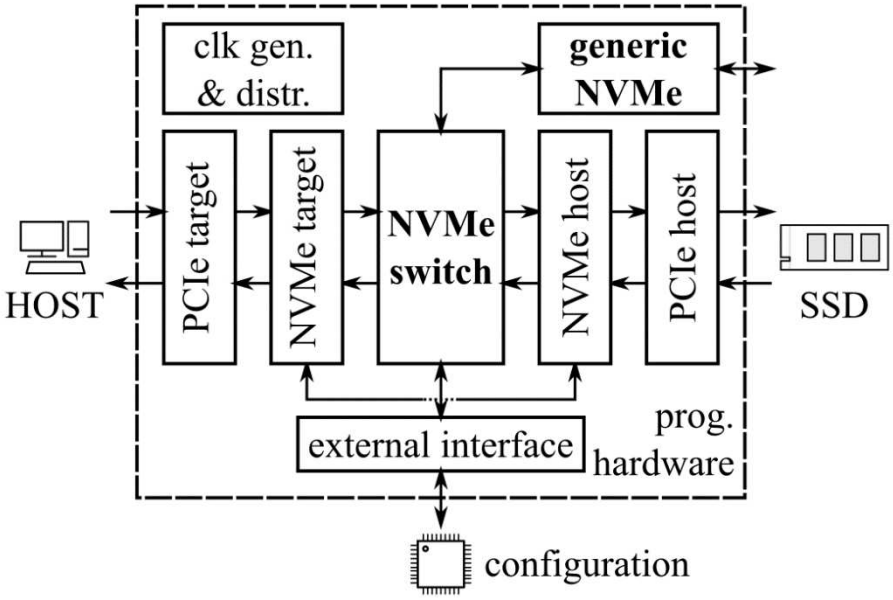


Fig. 1. Top-level architecture: it shows the layout of the modules with the NVMe switch in the middle

The *PCIe target* communicates with the host, and the *PCIe host* with the SSD. Similarly, the *NVMe target* communicates with the host, and the *NVMe host* with the SSD. The words “target” and “host” convey the sense of what interface is perceived by the host and SSD, respectively. The PCIe is required by the NVMe specification. NVMe modules deal with NVMe control, data flow, and PCI configuration. They translate the data to and from the NVMe streaming standard. This allows a flexible bridge for NVMe, which may be expanded later as needed. By virtue of the packet stream standard, the switch is presented in the data-path configuration.

The NVMe streaming standard uses byte-oriented frames. The first byte is acknowledged with the handshake, using the*valid* and *ready* signals. The rest of the bytes are automatically transferred. The burst length will be known according to the type of command: Submission Command (SC), 64 bytes; or Completion Commands (CC), 16 bytes. The *valid* signal comes from the source, indicating that the first byte is valid; and the *ready* signal is from the sink, indicating that the sink is ready to receive a full NVMe packet. The data transfer can only be peer-to-peer without a bus splitter module. This standard is the basis for the whole architecture. **Figure 2** shows a detailed view of the proposed switch architecture. The emphasis is on SC. The CC processing changes depending on the specific implementation of the *generic NVMe*. The relevant modules are explained in more detail below.

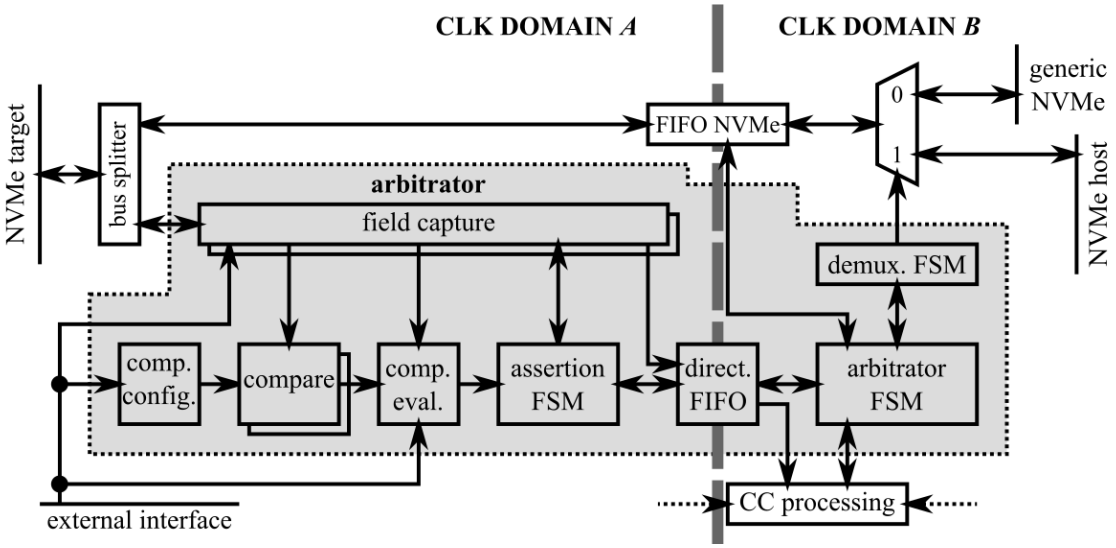


Fig. 2. Switch's internal architecture: it presents the SC packet flow side (above) and the CC processing side (below)

The *field capture* module observes the packets and cherry-picks NVMe fields based on a particular configuration and opcode. This module also separates the opcode of the command for use in other modules. It can accept multiple predefined configurations for arbitrator.

A *compare* module outputs a logical '1' or '0' depending on the evaluation. Comparisons are configurable for the type of evaluation (i.e. equality, inequality, boolean, etc) and comparison values. The module takes two evaluations and one evaluated value. Provided a configuration, the module will compare if the evaluated value is within a certain range, which is defined by one or two evaluation values. This is useful, for example, in taking a Start of Logical Block (SLBA) field and checking if this field hits a certain range in the address space. The designer may want to add more *compare* modules in parallel to check for more conditions. In this case, a *comparison evaluator* module will be required to logically combine the outputs. Moreover, the *comparison configurator* module is necessary to correctly translate between the arbitration configuration to the *compare* module. This arbitration configuration is acquired from the central processing unit (CPU), through the external interface.

The *FIFO NVMe* and *direction FIFO* are employed to completely separate clock domains. These clock domains are necessary in order to allow any halt by the SSD or *generic NVMe* without packet loss. There are two Finite State Machines (FSMs) in each clock domain: the *assertion* and the *arbitrator*. They are responsible for system management and synchronization. The main output of the *assertion* FSM is the direction of the command, alongside the FIFO's storage control signals. Because of the NVMe stream standard, the demultiplexer cannot change direction in the middle of a packet. Therefore, a *demultiplexer* FSM is a system to lock control over a module. It is used to resolve the dispute over the demultiplexer between the *generic NVMe* and *arbitrator* modules. Note that the packets are being stored in the *FIFO NVMe* while simultaneously being processed by the *arbitrator*, allowing for reduced latency. With this system in mind, one may proceed to analyze the simulation results presented in the next section.

III. Simulation Results

The proposed architecture shown in **Figure 2** has been developed in Very High Speed Integrated Circuit Hardware Description Language (VHDL). The implemented proof-of-concept is the switch, containing only one *compare* module. The SSD is linked to a logical '1' and the *generic NVMe* to a logical '0' (as shown in **Figure 2**) in the simulations below. The simulation was run in Vivado® Simulator 2019.1 and is available at GitLab (https://gitlab.com/storagesystemslaboratory/nvme-switch). The test configurations are shown in **Table no 1**. The AC_WORKLOAD configuration tests if the command's SLBA field access a certain address range, namely an *ext4*'s superblock.

Table no 1: Tested arbitrator configurations.

Configuration name	**Value**	***Generic NVMe*?**
AC_IDLE	0x00	FALSE
AC_SEND_GENERIC	0x01	TRUE
AC_SEND_SSD	0x02	FALSE
AC_WORKLOAD	0xFE	$v1 \leq SLBA \leq v2$

The testbench itself is a VHDL code with an automatic file consulting system - with each file being a NVMe packet. Each one is described in a separated text file, having a title and an activation signal. The title corresponds to the command's opcode, for our understanding. Once the reading of these packets is activated, the testbench will send the whole packet. Note that, if a packet does not contain the SLBA field used in AC_WORKLOAD the system will send it to the default destination - configured as the SSD. There are two packets that contain the SLBA field: *read1* and *read2*. Only the *read2* is to be sent to the *generic NVMe*. There are also fixed direction configurations, to send any packet independently of its fields. **Figure 3** shows the whole system working in the AC_WORKLOAD configuration.

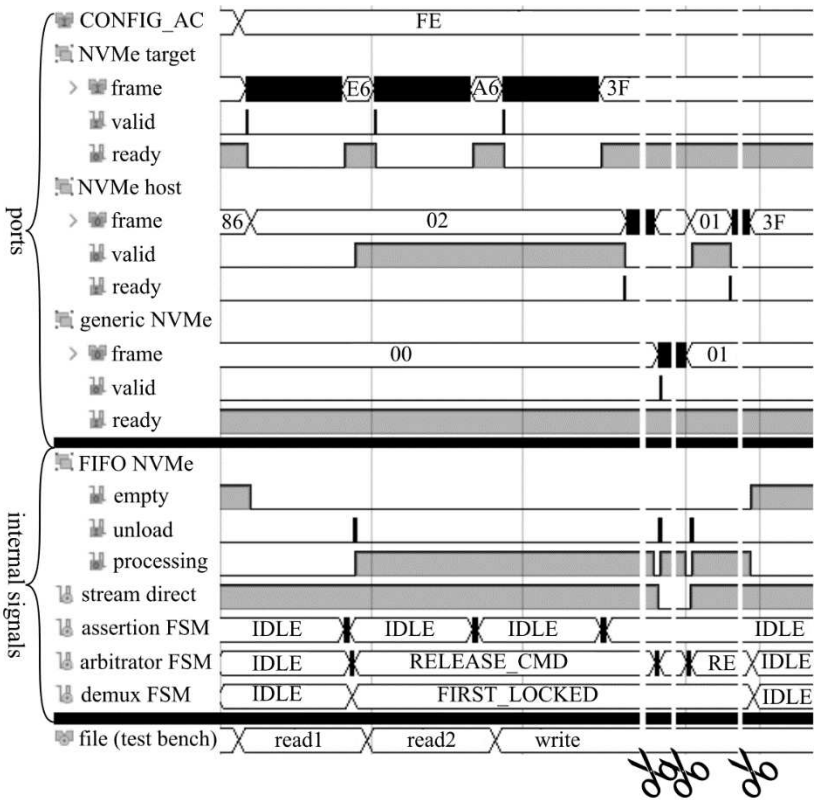


Fig. 3.AC_WORKLOAD configuration results for *read1*, *read2* and *write*. The three commands are buffered in the *FIFO NVMe*, displaying the switch's capability to hold the stream. The *read1* and *write* commands are sent to the SSD, whereas *read2* is sent to the *generic NVMe*, as expected. As indicated by the scissors, the figure was cut to present the most relevant parts of the simulation

IV. Conclusion

In this paper, we have proposed a solution for shortening the data semantic gap between host and SSD by allowing the host to gain more control over the SSD, without the burden of software processing. This has been accomplished through an NVMe-based switch with criteria configuration. This criterion can be configurated for different workloads at run-time via the external interface. The switch's proof-of-concept has been implemented in VHDL and validated in Vivado® Simulator 2019.1. The NVMe streaming standard facilitates the flow of packets throughout the design and it is the basis for the whole architecture. The results portrayed that the proposed design provides a flexible and expandable architecture that can be easily altered and/or increased in its functionality. The architecture also allows for other NVMe-based systems manipulation to run in parallel. Furthermore, in this design, any path may block the flow of packets without data loss. Thus, we have proved possible to arbitrate the flow of NVMe commands in hardware. The arbitration is configured for different workloads by the CPU, without in-datapath software intervention.

V. Future Work

In the future, we will run this design in a FPGA, alongside others functionalities such as NVMe *field capturing* and *opcode counting* for analysis purposes. This will allow us to make statistics out of the SSD. These statistics will serve as a tool for future research. A *generic NVMe* will be created to contain a cacheable address region with an automatic content flush, thus, extending the life-span of the SSD by reducing the number of writes in the flash cells. This will extend the life-span of the SSD by reducing the number of writes in the flash

cells. This implementation will be tested with the *ext4* FS in journaling mode. This work is a subset of an ongoing project that targets a complete solution for NVMe-based SSD devices for industry and research.

References

[1]. NVMe Express Workgroup: 'NVM Express Revision 1.3' https://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf, accessed December 2019

[2]. Schroeder, B., *et al*.: 'Flash Reliability in Production: The Expected and the Unexpected', 15th USENIX Conference on File and Storage Technologies, Santa Clara, USA, February 2017, pp. 67-80, isbn: 978-1-931971-28-7

[3]. Stratikopoulos, A., *et al*.: 'FastPath: Toward Wire-speed NVMe SSDs', *International Conference on Field-Programmable Logic and Applications*, July 2018, pp. 170-177, doi: 10.1109/FPL.2018.00036

[4]. Yang, C. *et al*.: 'Reducing Write Amplification for Inodes of Journaling File System using Persistent Memory', *Design, Automation & Test in Europe Conference & Exhibition*, March 2019, pp. 866-871, doi: 10.23919/DATE.2019.8715068

[5]. Cai, Y., *et al*.: 'Error characterization, mitigation, and recovery in flash-memory-based solid-state drives', *Proceedings of the IEEE*, 2017, **105**, (9), pp. 1666-1704, doi: 10.1109/JPROC.2017.2713127

[6]. Kim, K., *et al*.: 'Effect of field oxide structure on endurance characteristics of NAND flash memory', *Electronic Letters*, 2014, **50**, (10), pp; 739-741, doi: 10.1049/el.2014.0522

[7]. Intelliprop Inc.: 'NVMe to NVMe Bridge' https://www.xilinx.com/products/intellectual-property/1-mpymke.html, accessed March 2020

[8]. Adjari, M., *et al.*: 'A Scalable HW-based Inline Deduplication for SSD Array', *IEEE Computer Architecture Letters*, September 2018, **17**, (1), pp. 47-50, doi: 10.1109/LCA.2017.2753258

[9]. Li, D., *et al*.: 'CISC: Coordinating Intelligent SSD and CPU to Speedup Graph Processing', *17th International Symposium on Parallel and Distributed Computing*, June 2018, pp. 149-156, doi: 10.1109/ISPDC2018.2018.00029

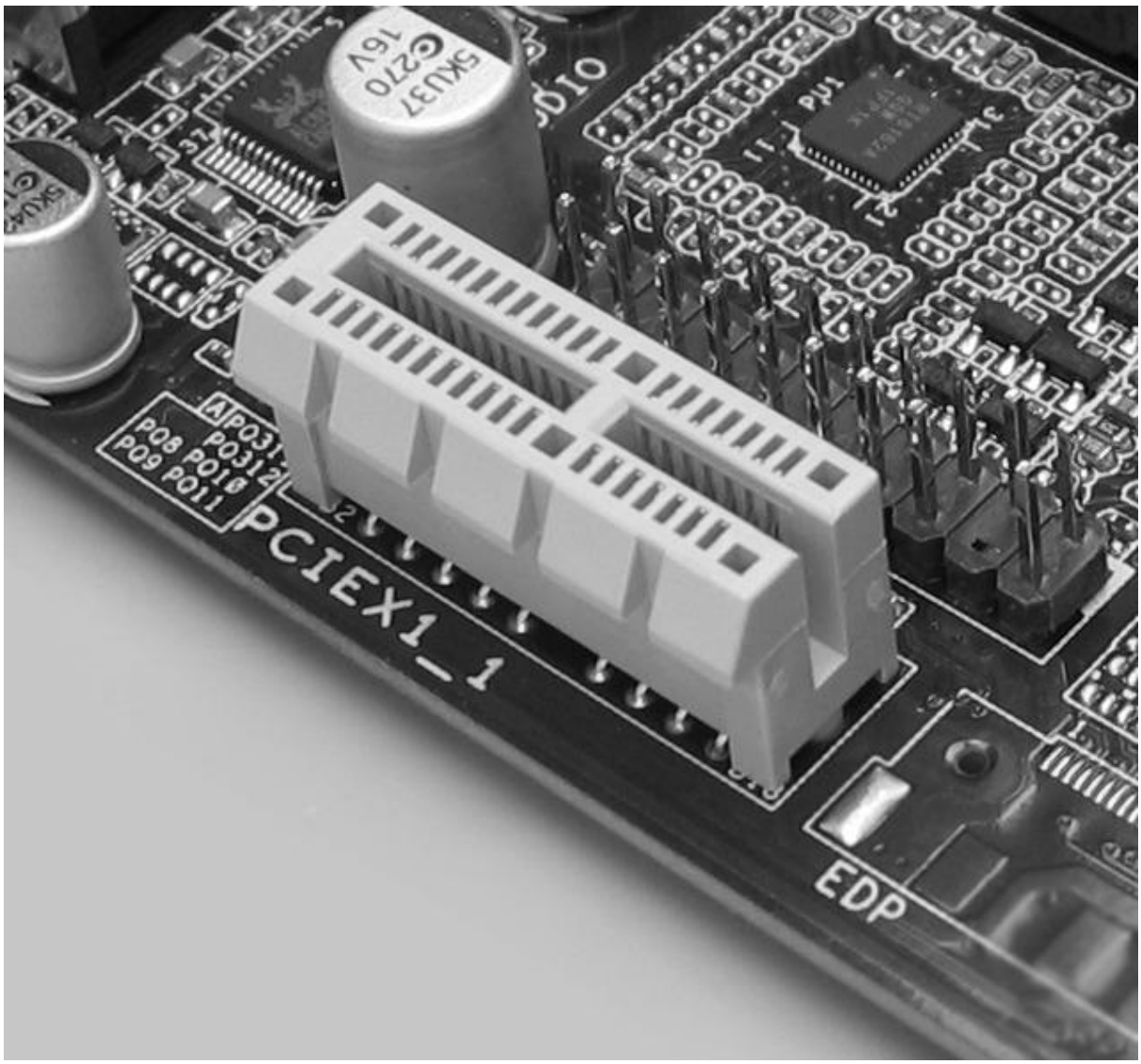
[10]. Jun, S., *et al*.: 'GraFBoost: Using Accelerated Flash Storage for External Graph Analytics', *ACM/IEEE 45th Annual International Symposium on Computer Architecture*, June 2018, pp. 411-424, doi: 10.1109/ISCA.2018.00042

[11]. Zhang, J., *et al*.: 'Design and Implementation of Optical Fiber SSD Exploiting FPGA Accelerated NVMe', *IEEE Access*, October 2019, **7**, pp. 152944-152952, doi: 10.1109/ACCESS.2019.2947181

Vinícius G. Linden,etal."Enabling Fast Decision Criteria in NVMe SSDs through NVMe-based Switch." *IOSR Journal of Electrical and Electronics Engineering (IOSR-JEEE)*, 15(2), (2020): pp. 54-58.

ANEXO A – CONECTOR PCIE FÊMEA ABERTO

Figura 35 – Conector padrão fêmea PCIe (x1) aberto



Fonte: Adaptado de Hans Haase.